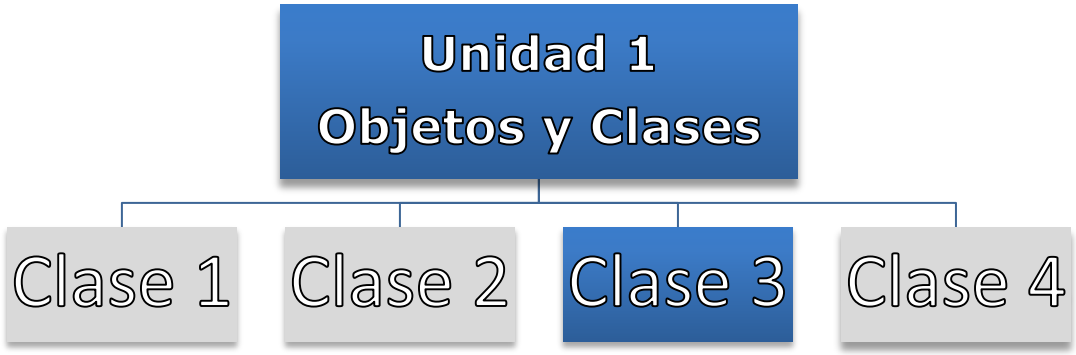
PROGRAMACIÓN ORIENTADA A OBJETOS



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

Presentación

La clase 3 corresponde a la unidad 1 de la asignatura. En esta unidad presentaremos la forma de utilizar adecuadamente propiedades y métodos, así como sus características más relevantes.

Lo invitamos a incursionar en los temas propuestos, ya que le brindarán no sólo un aporte tecnológico a su formación, sino que podrá mejorar su perspectiva profesional sobre la visión que posee acerca de *cómo desarrollar software*.

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 3.

- El modelo orientado a objetos.
- Jerarquías "Es - Un" y "Todo - Parte".
- Concepto de Clase y Objeto.
- Características básicas de un objeto: estado, comportamiento e identidad.
- Ciclo de vida de un objeto.
- Modelos. Modelo estático. Modelo dinámico. Modelo lógico. Modelo físico.
- Concepto de análisis diseño y programación orientada a objetos.
- Conceptos de encapsulado, abstracción, modularidad y jerarquía.
- Concurrencia y persistencia.
- Concepto de clase.
- Definición e implementación de una clase
- Campos y Constantes
- **Propiedades. Concepto de Getter() y Setter(). Propiedades de solo lectura. Propiedades de solo escritura. Propiedades de lectura-escritura. Propiedades con indizadores. Propiedades autoimplementadas. Propiedades de acceso diferenciado.**
- **Métodos. Métodos sin parámetros. Métodos con parámetros por valor. Métodos con parámetros por referencia. Valores de retorno de referencia.**
- **Sobrecarga de métodos.**
- Constructores. Constructores predeterminados. Constructores con argumentos.
- Finalizadores.

- Clases anidadas.

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.

1. Propiedades y métodos

Lecturas requeridas

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/methods

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters

2. Sobrecarga

Lecturas requeridas

Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XI. Punto 11.8

1. Propiedades y métodos

Propiedades.

Una **propiedad** es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados *descriptores de acceso*. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

Las **propiedades** se utilizan para acceder (lectura o escritura) a los campos privados de las clases. Estos campos privados representan las características cualitativas y cuantitativas que posee la clase.

Para devolver el valor de la **propiedad** se usa un descriptor de acceso de propiedad **get**, mientras que para asignar un nuevo valor se emplea un descriptor de acceso de propiedad **set**. Estos descriptores de acceso pueden tener diferentes niveles de acceso.

Las **propiedades** pueden ser *de lectura y escritura*, *de solo lectura* (tienen un descriptor de acceso get, pero no set) o *de solo escritura* (tienen un descriptor de acceso set, pero no get).

Las propiedades simples que no necesitan ningún código de descriptor de acceso personalizado se pueden implementar como como propiedades implementadas automáticamente o implícitas.

El lenguaje define la sintaxis con la cual se programan las **propiedades.**

```
public class Cliente

{
    0 referencias
    public string Nombre { get; set; }
    0 referencias
    public string Apellido { get; set; }
}
```

Ej0005

El el Ej0005 se puede observar como la clase Cliente define dos **propiedades** denominadas Nombre y Apellido respectivamente. A gravés de ellas podremos mantener el nombre y apellido de un cliente. Para utilizarlo debemos previamente instanciar un objeto como se observa a continuación.

```
public Cliente MiCliente;
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    MiCliente = new Cliente();
}
```

Ej0005

Luego cargamos el nombre y apellido del objeto que es apuntado por la variable MiCliente a través de las **propiedades**.

```
private void button1_Click(object sender, EventArgs e)
{
    MiCliente.Nombre = this.textBox1.Text;
    MiCliente.Apellido = this.textBox2.Text;
}
```

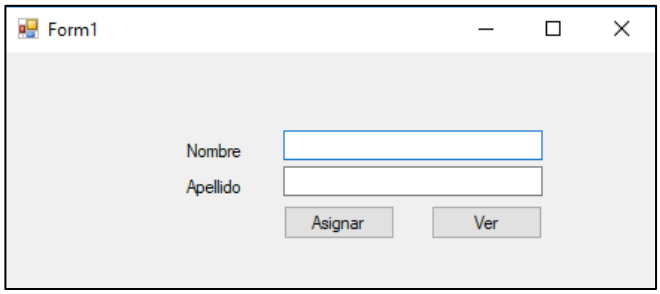
Ej0005

Finalmente leemos las **propiedades** para corroborar que los valores fueron almacenados como parte del estado del objeto.

```
1 referencia
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("El Nombre es: " + this.textBox1.Text + "\r\nEl Apellido es: " + this.textBox2.Text);
}
```

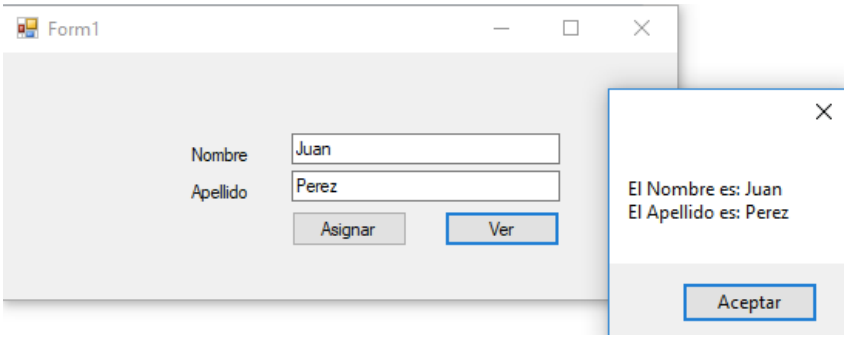
Ej0005

La interfaz del programa permite la carga y visualización de las propiedades.



Ej0005

Para corroborarlo ingrsamos un nombre y apellido y procedemos a oprimir el botón Asignar. Luego Oprimimos el botón Ver.



Ej0005

En la sintaxis anterior la **propiedad** de define **implicitamente,** ya que como pudo observarse en ningún momento se definieron campos para almacenar el nombre y el apellido. El compilador genera automaticamente la ubicación de almacenamiento para los campos que respaldan a la propiedad. El compilador también implementa el cuerpo de los descriptores de acceso **get** y **set**.

Propiedades con campos definidos por el programador.

Se puede definir una **propiedad** de manera que el valor quede respaldado en una variable privada definida por el programador.

```
public class Cliente

{
    string Vnombre = "";
    string Vapellido = "";
    0 referencias
    public string Nombre
    {
        get { return this.Vnombre; }
        set { this.Vnombre = value; }
    }
    0 referencias
    public string Apellido
    {
        get { return this.Vapellido; }
        set { this.Vapellido = value; }
    }
}
```

Ej0006

En los descriptores de acceso se puede colocar código. En el ejemplo siguiente se valida que la hora ingresada se encuentre en 0 y 24.

```
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        Reloj R = new Reloj();
        R.Horas = int.Parse(this.textBox1.Text);
        this.textBox2.Text = (R.Horas * 3600).ToString();
    }
}
2 referencias
public class Reloj
{
    int Vhoras = 0;
    2 referencias
    public int Horas
    {
        get { return this.Vhoras;}
        set {
                if (value < 0 || value > 24)
                { MessageBox.Show("La hora debe ser mayor a 0 y menor a 24 !!!"); }
                else
                { this.Vhoras = value; }
            }
    }
}
```

Ej0007

Los **descriptores de acceso** de **propiedad** suelen constar de instrucciones de una sola línea que simplemente asignan o devuelven el resultado de una expresión. Las definiciones de cuerpos de expresión constan del símbolo => seguido de la expresión que se va a asignar a la propiedad o a recuperar de ella.

```
public class Cliente
{
    string Vnombre = "";
    string Vapellido = "";
    1 referencia
    public string Nombre { get => this.Vnombre; set => this.Vnombre = value; }
    1 referencia
    public string Apellido { get=> this.Vapellido; set=> this.Vapellido=value; }
}
```

Ej0008

Propiedades de solo lectura y solo escritura.

Como sus nombres lo indican, estan **propiedades** sirven solo para ser leidas o solo para ser escritas. Estas últimas son menos utilizadas que la primeras pero en el siguiente ejemplo se puede observar como se implementan.

```
public class Persona
{
    DateTime VfechaDeNacimiento;
    1 referencia
    public DateTime FechaDeNacimiento
    {
        set {this.VfechaDeNacimiento = value; }
    }
    1 referencia
    public byte Edad
    {
        get
        {
            byte axo = (byte)this.VfechaDeNacimiento.Year;
            if (this.VfechaDeNacimiento.DayOfYear <= DateTime.Now.DayOfYear) { axo -= 1; }
            return(byte)(DateTime.Now.Year - axo);
        }
    }
}
```

Ej0009

En el ejemplo anterior se puede observar la **propiedad FechaDeNacimiento** de **solo escritura** y la **propiedad Edad** de **solo lectura.**

Indizadores.

Los Indizadores se utilizan almacenar o recuperar elemntos guardados en una instancia. El valor indizado se puede establecer o recuperar sin especificar explícitamente un miembro. Son similares a propiedades, excepto en que sus descriptores de acceso usan parámetros.

En el ejemplo siguiente, en la clase AlmacenaString se puede observar un **indizador** implementado de manera que permite almacenar 10 string dentro del objeto. Para almacenar los valores o recuperarlos solo se necesita especificar el nombre de la instancia y el índice deseado.

```
class AlmacenaString
{
    // Se declara un arrar para almacenar 10 string.
    private string[] ArrString = new string[10];
    // Se define el indizador.
    3 referencias
    public string this[int i]
    {
        get { return ArrString[i]; }
        set { ArrString[i] = value; }
    }
}
```

Ej0010

Propiedades con restricciones de accesibilidad en los descriptores.

Las partes get y set de una propiedad se denominan descriptores de acceso. De forma predeterminada, estos descriptores de acceso tienen la misma visibilidad o nivel de acceso de la propiedad al que pertenecen. Se puede restringir el acceso a uno de estos descriptores de acceso. Normalmente, esto implica restringir la accesibilidad del descriptor de acceso set, mientras que se mantiene el descriptor de acceso get accesible públicamente.

En el ejemplo siguiente se puede observar como el descriptor de acceso set incorpora **protected**. Esto produce que dentro de la clase Cliente se puede hacer uso de él. También se podría utilizar en una clase que herede de Cliente. Pero por ejemplo si otra clase instancia un Cliente y desea utilizar la propiedad Nombre, estará disponible el get pero no así el set.

```
public class Cliente
{
    string Vnombre = "";
    1 referencia
    public string Nombre
    {
        get { return this.Vnombre; }
        protected set { this.Vnombre = value; }
    }
    0 referencias
    public string Apellido { get; set; }
}
```

Ej0011

```
0 referencias
private void button1_Click(object sender, EventArgs e)
{
    MiCliente.Nombre = this.textBox1.Text;
    Mi
}
(campo) Cliente Form1.MiCliente
La propiedad o el indizador 'Cliente.Nombre' no se pueden usar en este contexto porque el descriptor de acceso set es inaccesible
0 referencias
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("El Nombre es: " + this.textBox1.Text + "\r\nEl Apellido es: " + this.textBox2.Text);
}
```

Ej0011

Métodos.

Los **métodos** se construyen a partir de un bloque de código que contiene una serie de instrucciones. Ese bloque de código recibe un nombre, lo que se denomina nombre del método. Los metodos pueden retornar un valor o no. De la misma manera los métodos pueden poseer parámetros o no. Los métodos poseen definida una visibilidad como por ejemplo **public** o **private** entre otras. El método Main es el punto de entrada para cada una aplicación de C#.

Firma del método.

Los métodos se declaran en una clase o una estructura. Como se mencionó anteriormente se debe especificar el el nivel de acceso. Existen otros modificadores opcionales como **abstract** (clases, métodos, propiedades, indexadores y eventos que no poseen implementación) o **sealed** (impide que otras clases hereden de ella), el valor de retorno, el nombre del método y cualquier parámetro de método. Todas estas partes forman la **firma** del método.

Se debe tener en cuenta que un tipo de valor devuelto por un método no forma parte de la **firma** del método.

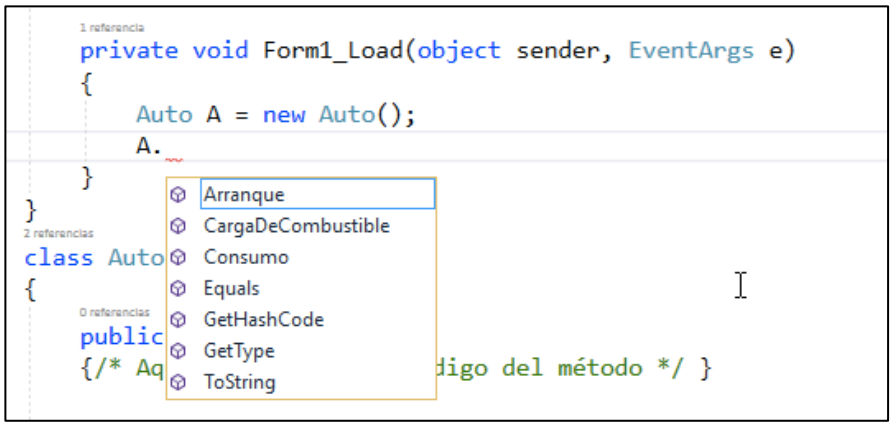
```
class Auto
{
    0 referencias
    public void Arranque()
    {/* Aquí se coloca el cóodigo del método */ }

    0 referencias
    public void CargaDeCombustible(int litros)
    { /* Aquí se coloca el cóodigo del método */ }

    0 referencias
    public  int Consumo(int km, int velocidad)
    { /* Aquí se coloca el cóodigo del método */ return 1; }
}
```

Ej0012

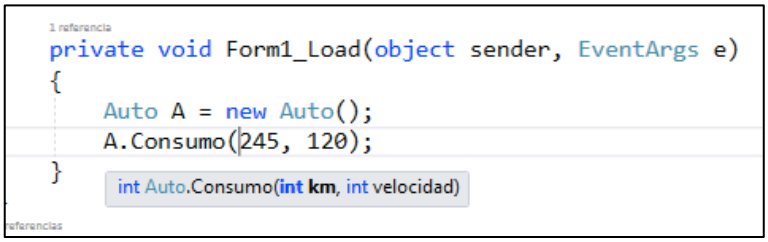
En el ejemplo 12 se pueden observar tres métodos. El primero no retorna valor y no posee parámetros. El segundo no retorna valor y posee un parámetro de tipo entero y el tercero retorna un valor entero y posee dos parámetros, ambos del tipo entero.

Acceso a métodos.

Para acceder a un método tradicional, primero se debe crear un objeto y luego se debe colocar el nombre de la variable que lo apuntay agreguar un punto, sobre el listado desplegado se debe seleccionar el nombre del método. Los argumentos se enumeran entre paréntesis y están separados por comas. Los métodos de la clase Auto se pueden llamar como en el siguiente ejemplo.



Ej0012



Ej0012

Parámetros por Valor y por Referencia

Existen dos tipos de variables, variables del tipo **valor** y variables del tipo referencia. Una variable tipo de **valor** contiene sus datos directamente, en oposición a la variable tipo de **referencia**, que contiene una referencia (la posición en memoria) a sus datos.

Pasaje por Valor de un Value Type

Si tenemos una función con parámetros y estos son del tipo **valor**, al pasarle valores se realiza un copia de estos tomando como ámbito lo que defina la función. De esta forma se preserva el valor que contiene la variable utilizada como argumento o sea lo que se pasó desde fuera de la función. Esta opera como algo independiente al valor recibido por el parámetro de la función. Ningún cambio realizado en el parámetro dentro del método afecta a los datos originales almacenados en la variable utilizada como argumento.

```
class PasajePorValor
{
    1 referencia
    public void RecibeNumero(int i)
    {
    MessageBox.Show("valor del parámetro i dentro de la función: " + i);
    i = 20;
    MessageBox.Show("valor del parámetro i luego de asignarle 20 dentro de la función: " + i);
    }
}
```

Ej0013

En el ejemplo Ej0013 se puede observar una clase denominada **PasajePorValor** que posee un método llamado **RecibeNumero**. Este método posee un parametro i de un tipo de tipo valor (**int**).

El llamado al método se hace a traves de la variable PV que apunta al objeto que se instanció. Al ejecutar el código, se puede observar que el valor del argumento i(10) definido fuera de la función y que se pasa al parámetro i de la función, no ve alterado su valor cuando dentro de la función se le asigna un valor de 20. Esto se debe a que el parámetro i copia el valor recibido por el argumento y cada uno

se almacena en distintas posiciones de memoria. Si bien el nombre de las variables que se utilizan como argumento y parámetro es el mismo, cada una opera en distintos ámbitos y por ende ocupa distintas direcciones de memoria.

```
private void Form1_Load(object sender, EventArgs e)
{
    PasajePorValor PV = new PasajePorValor();
    int i = 10;
    MessageBox.Show("valor de la variable i que será usada como " +
                    "argumento antes de llamar a la función: " + i);
    PV.RecibeNumero(i);
    MessageBox.Show("valor de la variable i después de llamar a la función: " + i);
}
```

Ej0013

Pasaje por Referencia de un Value Type.

El pasaje por referencia se distingue del anterior en que el argumento se pasa como un parámetro **ref**. El valor del argumento subyacente, **i**, se cambia cuando se modifica **i** en el método.

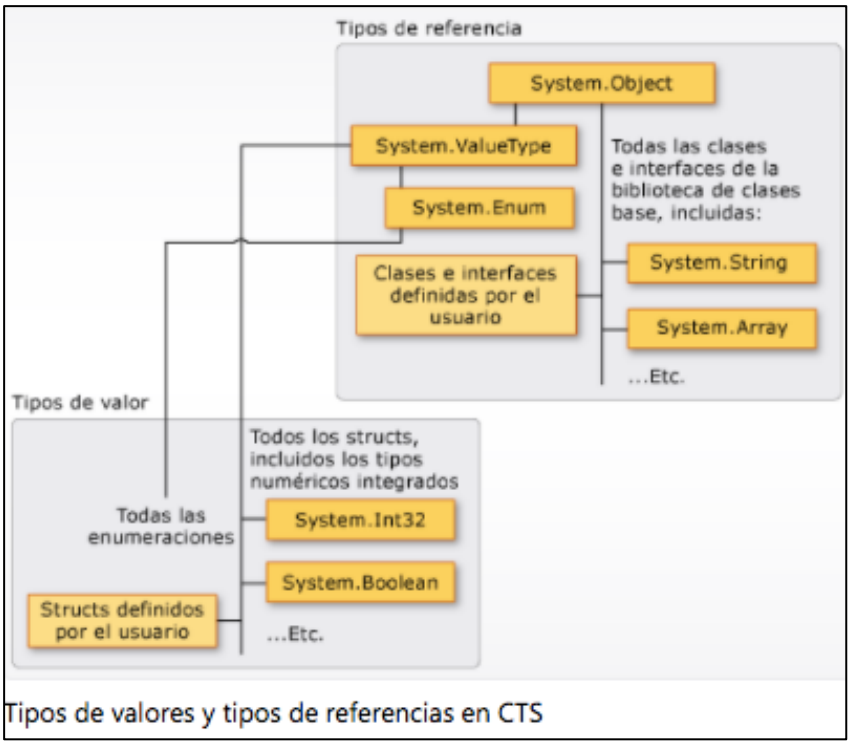
En este ejemplo Ej0014, no es el valor de **i** el que se pasa, sino una referencia a **i**. El parámetro **i** no es un entero, se trata de una referencia a un **int** en este caso, una referencia a **i**. Por tanto, cuando **i** se modifica dentro del método, lo que realmente se modifica es lo que hay en el lugar a donde hace referencia (o apunta) **i**.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }

    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        PasajePorValor PV = new PasajePorValor();
        int i = 10;
        MessageBox.Show("valor de la variable i que será usada como " +
                        "argumento antes de llamar a la función: " + i);
        PV.RecibeNumero(ref i);
        MessageBox.Show("valor de la variable i después de llamar a la función: " + i);
    }
}
2 referencias
class PasajePorValor
{
    1 referencia
    public void RecibeNumero(ref int i)
    {
        MessageBox.Show("valor del parámetro i dentro de la función: " + i);
        i = 20;
        MessageBox.Show("valor del parámetro i luego de asignarle 20 dentro de la función: " + i);
    }
}
```

Ej0014

Pasaje por Valor de un Reference Type

Para poder observar como funciona esta combinación repasaremos el esquema de clasificación de typos que posee .



Tipos de valores y tipos de referencias en CTS

Como se puede observar, los tipos numéricos integrados se corresponde con el grupo de **Value Type** y las clases, los Array y String al grupo **Reference Type**.

Es por ello que los elemplos Ej0013 y Ej0014 se han realizado considerando un tipo entero (**int i**) que es un **Value Type** y los dos que desarrollaremos en adelante utilizarán el tipo **Array** que se corresponde con un **Reference Type.**

Cuando pasamos por valor un **Reference Type** como se muestra en el ejemplo Ej0015, se está pasando al parámetro de la función una copia de la referencia (en este caso el Array). Se podrá cambiar los valores de los elementos del Array pasado y estos cambios se podrán visualizar fuera de la función. En cambio, el intento de volver a asignar el parámetro a otra ubicación de memoria solo funciona dentro del método y no afecta a la variable original, **MiArray**.

En el ejemplo anterior, el array **MiArray**, que es un tipo de referencia, se pasa al método sin el parámetro **ref**. Se pasa al método una copia de la referencia, que apunta a **MiArray**. El resultado muestra que es posible que el método cambie el contenido de un elemento del array, en nuestro ejemplo de 1 a 9. En cambio, si se asigna una nueva porción de memoria al usar el operador **new** dentro del método **CambioDeValores**, la variable **pArray** que represent al parámetro de la función, hace referencia a una nueva matriz. Por tanto, cualquier cambio que hubiese después no afectará a la matriz original **MiArray.** De hecho, se crean dos

matrices en este ejemplo, una dentro de **Form1_Load** y otra dentro del método **CambioDeVAlores**.

```
private void Form1_Load(object sender, EventArgs e)
{
    int[] MiArray = { 1, 3, 5 };
    MessageBox.Show("Valor del primer elemento del Array antes de llamar" +
                            " a la función CambioDeVAlores es: " + MiArray[0]);
    CambioDeValores(MiArray);
    MessageBox.Show("Valor del primer elemento del Array después de llamar" +
                    " a la función CambioDeVAlores es: " + MiArray[0]);
}
1 referencia
void CambioDeValores(int[] pArray)
{
    pArray[0] = 9;  // Este cambio afecta al elemento original.
    pArray = new int[4] { 2, 4, 6, 8};   // Este cambio es local.
    MessageBox.Show("Dentro del método el primer elemento del Array es: " + pArray[0]);
}
```

Ej0015

Pasaje por Referencia de un Reference Type.

En este caso podremos observar cómo al realizar la nueva asignación de memoria dentro de la función (`pArray = new int[4] {2, 4, 6, 8};`) los valores son afectados, inclusive fuera de la función, aspecto que no se veía reflejado en el ejemplo anterior.

```
private void Form1_Load(object sender, EventArgs e)
{
    int[] MiArray = { 1, 3, 5 };
    MessageBox.Show("Valor del primer elemento del Array antes de llamar" +
                            " a la función CambioDeVAlores es: " + MiArray[0]);
    CambioDeValores(ref MiArray);
    MessageBox.Show("Valor del primer elemento del Array después de llamar" +
                    " a la función CambioDeVAlores es: " + MiArray[0]);
}
1 referencia
void CambioDeValores(ref int[] pArray)
{
    pArray[0] = 9;  // Este cambio afecta al elemento original.
    pArray = new int[4] {2, 4, 6, 8};   // Este cambio afecta al elemento original.
    MessageBox.Show("Dentro del método el primer elemento del Array es: " + pArray[0]);
}
```

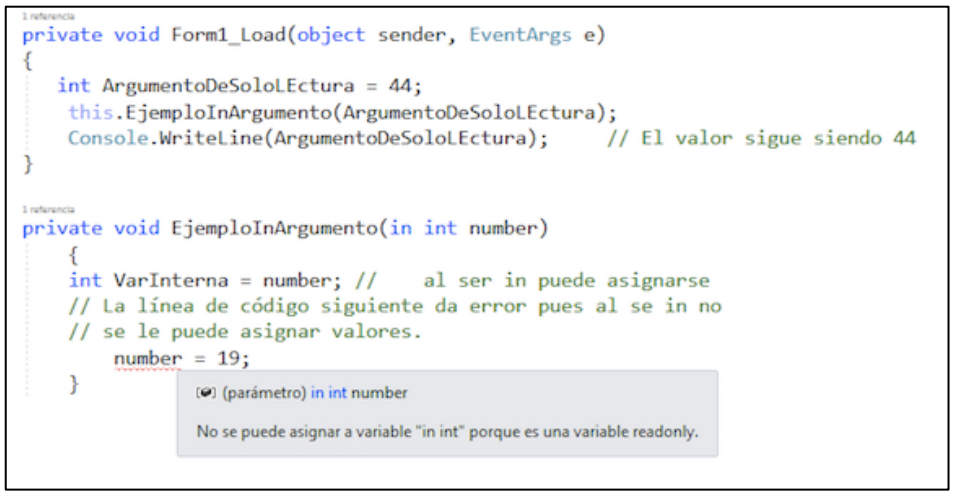
Ej0016

En el ejemplo Ej0016 cunado se solicita ver el primer elemento del array luego de invocar la función **CambioDeValores** se observará 2 y no 9 como en el ejemplo Ej0015.

Uso de los modificadores de parámetro *in* y *out.*

La palabra clave **in** hace que los argumentos se pasen por referencia. Es como las palabras clave **ref** o **out**, salvo que el método llamado no puede modificar los argumentos que posean el modificador **in**. Mientras que los argumentos **ref**, como ya vimos, sí se pueden modificar.

Las variables que se han pasado como argumentos **in** deben inicializarse antes de pasarse en una llamada de método. Sin embargo, es posible que el método llamado no asigne ningún valor o modifique el argumento.



Ej0017

Como se puede observar el parámetro *number* afectado con el modificador **in,** al ser asignado dentro del método *EjempoInArgumento* a la variable *VarInterna*, no causa ningún problema. No obstante la línea siguiente al intentar asignarle en número 19 al parámetro *number* genera un error, debido a que los parámetros con **in** no pueden ser alterados dentro del método.

La palabra clave **out** hace que los argumentos se pasen por referencia. Esto es como la palabra clave **ref**, salvo que **ref** requiere que se inicialice la variable antes de pasarla. Para usar un parámetro out, tanto la definición de método como el método de llamada deben utilizar explícitamente la palabra clave out. Esto permitirá que la variable utilizada como parámetro, pueda recibir un valor dentro del método y ese valor será tomado por la variable utilizada como argumento al llamar a la función.

```
private void Form1_Load(object sender, EventArgs e)
{
    int valor;
    this.EjemploOut(out valor);
    MessageBox.Show("El valor observado es el cargado dentro de " +
        "la función por medio del parámetro out: " + valor.ToString());
}
1 referencia
void EjemploOut(out int p)
{
    p = 55;
}
```

Ej0018

Clases anidadas.

Una clase anidada es una clase definida dentro de otra clase. Por defecto la clase anidad tendre un modificador de acceso privado a pesar que se le puede modificar.

El uso de las clases anidadas entre otras cosas sirve para organizar clases que en general se crean con el objetivo de darle servicios a quien la contiene o la naturaleza de su existencia está muy ligada a ella.

Si la clase contenida se define como privada solo se podrá utilizar dentro de la clase contenedora.

En el ejemplo **Ej0019** se puede observar como la clase **Contenedor** anida la clase **Contenido** cuyo modificdor de acceso es **private**. Esto causa que dentro de la clase **Contenedor,** en el método denominado **Uso,** se pueda instanciar un objeto del tipo **Contenido** sin inconvenientes. Pero si observamos el método **Form1_Load** de la clase **Form1,** encontraremos que al intentar declarar la variable **VarC** del tipo **Contenedor.Contenido**, no solo arroja un error la definición de la variable, sino que también el intento de instanciar un objeto de ese tipo. Esto es debido a la visibilidad (private) que posee la clase **Contenido**.

Así como en este caso se ha aplicado el modoficador de acceso **private**, también se pueden utilizar los otros modificadores de acceso: public, protected, internal, protected internal o private protected.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }

    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Contenedor.Contenido VarC = new Contenedor.Contenido();
    }
}
2 referencias
public class Contenedor
{
    0 referencias
    void Uso() { Contenido C = new Contenido(); }
    4 referencias
    private class Contenido
    {

    }
}
```

'Contenedor.Contenido' no es accesible debido a su nivel de protección

Ej0019

Si la clase contenida se define como pública también se la podrá utilizar fuera de la clase contenedora. A continuación se observa esto en el ejemplo **Ej0020** desarrollado con esa modificación en base al **Ej0019**. Se puede observar como los errores observados en el ejemplo anterior han desaparecido.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Contenedor.Contenido VarC = new Contenedor.Contenido();
    }
}
2 referencias
public class Contenedor
{
    0 referencias
    void Uso() { Contenido C = new Contenido(); }
    4 referencias
    public class Contenido {}
}
```

Ej0020

El tipo anidado o interno puede tener acceso al tipo contenedor o externo. Para tener acceso al tipo contenedor, se lo puede pasar como un argumento al constructor del tipo anidado. El tema constructores se analizará más adelante en el material. El ejemplo **Ej0021** muestra como hacer esto.

Un tipo anidado tiene acceso a todos los miembros que estén accesibles para el tipo contenedor. Puede tener acceso a los miembros privados y protegidos del tipo contenedor, incluidos los miembros protegidos heredados.

```
    private void Form1_Load(object sender, EventArgs e)
    {
        Contenedor.Contenido VarC = new Contenedor.Contenido();
    }
}
4 referencias
public class Contenedor
{
    1 referencia
    void Uso()
    { Contenido C = new Contenido(this); }
    6 referencias
    public class Contenido
    {
        Contenedor C;
        1 referencia
        public Contenido() { }
        1 referencia
        public Contenido(Contenedor pContenedor)
        { C = pContenedor; C.Uso(); }
    }
}
```

Ej0021

2. Sobrecarga

Sobrecargar un miembro significa crear dos o más miembros en la misma clase con el mismo nombre que solo difieren en el número o tipo de parámetros.

Se puede **sobrecargar** los métodos, constructores y propiedades indizadas. La **sobrecarga** es una de las técnicas más importantes para mejorar la facilidad de uso y la productividad. Construir una **sobrecarga** en el número de parámetros permite proporcionar versiones más sencillas y reutilizables en métodos y constructores. **Sobrecargar** alterando el tipo de los parámetros permite usar el mismo nombre de miembro para realizar operaciones idénticas en un conjunto seleccionado de diferentes tipos de miembros.

Sobrecarga en constructores

Sobrecargar un miembro significa crear dos o más miembros en la misma clase con el mismo nombre que solo difieren en el número o tipo de parámetros.

En el ejemplo Ej0028 se pueden observar dos clases que **sobrecargar** sus constructores. En el primer caso por la cantidad de parámetros que posee cada constructor y en el segundo por el tipo de datos de los parámetros.

```
public class Radio
{
    0 referencias
    public Radio() { }
    0 referencias
    public Radio(string pNombrePrograma) { }
    0 referencias
    public Radio(string pNombrePrograma, bool pGrabar) { }
}
3 referencias
public class Potencia
{
    0 referencias
    public Potencia(int pBase,int pPotencia) { }
    0 referencias
    public Potencia(decimal pBase, int pPotencia) { }
    0 referencias
    public Potencia(double pBase, int pPotencia) { }
}
```

Ej0028

Sobrecarga en propiedades.

Las propiedades indizadas se pueden sobrecargar utilizando la misma estrategia esplicada anteriormente. Por los tipos de sus parámetros o por la cantidad de parámetros. En el ejemplo **Ej0029** se puede visualizar lo expuesto.

```
    private void Form1_Load(object sender, EventArgs e)
    {
        Persona P = new Persona();
        P[""] = "Juan Martinez";
        MessageBox.Show(P["Director"]);
        MessageBox.Show(P["Director", "Ingeniería"]);
    }
}
2 referencias
public class Persona
{
    string Vnombre = "";
    2 referencias
    public string this[string pCargo]
    {
        get { return pCargo + " " + this.Vnombre; }
        set { this.Vnombre = value; }
    }
    1 referencia
    public string this[string pCargo, string pDepartamento]
    {
        get { return "Departamento: " + pDepartamento + " - " + pCargo + " " + this.Vnombre; }
        set { this.Vnombre = value; }
    }

}
```

Ej0029

Sobrecarga de métodos.

Los métodos sobrecargados permiten bajo el mismo nombre flexibilizar su uso. El ejemplo **Ej0030** muestra un ejemplo de lo enunciado.

```
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Calculo C = new Calculo();
        MessageBox.Show(C.Sumar(3, 5).ToString());
        MessageBox.Show(C.Sumar(3.5, 5.4).ToString());
        MessageBox.Show(C.Sumar(new int[] { 2, 4, 6, 8 }).ToString());
    }
}
2 referencias
public class Calculo
{
    1 referencia
    public int Sumar(int n1, int n2) { return n1 + n2; }
    1 referencia
    public double Sumar(double n1, double n2) { return n1 + n2; }
    1 referencia
    public int Sumar(int[] n)
    {int r = 0; foreach (int x in n) { r += x; } return r; }
}
```

Ej0030

- 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 es una propiedad de una clase?
2. ¿Qué tipos de propiedades existen?
3. ¿Qué ámbitos pueden tener los campos, métodos y propiedades de las clases?
4. ¿Qué características posee cada ámbito existente si se lo aplico a un campo, un método y una propiedad de una clase?
5. ¿A qué se denomina tiempo de vida de un objeto?
6. ¿Qué es un campo de una clase?
7. ¿Qué es un método de una clase?
8. ¿A qué denominamos sobrecarga?
9. ¿Qué tipos de parámetros puede tener un método?
10. ¿Para qué y como se usan los modificadores de parámetros in y out en los métodos?

Guía de ejercicios

Los siguientes ejercicios son conceptuales y no se deben desarrollar en código. Para su elaboración puede utilizar cualquier graficador (por ejemplo draw.io). Para representar una clase, sus propiedades, métodos y las relaciones entre ellas utilice la simbología UML que su profesor le indicará en clase.

1. Desarrollar una jerarquía de clases relacionadas por herencia y agregación que represente la estructura necesaria para identificar claramente los elementos encontrados en un sistema de calificaciones de una universidad. En cada clase detallar métodos y propiedades.

2. Defina una situación donde se desee llegar desde un estado inicia a uno final, por ejemplo, una cuenta bancaria que posee un saldo inicial y se la somete a una operación bancaria. Desarrollar la clase cuenta con las propiedades y métodos que considere pertinentes y demuestre como cambia el estado del objeto en la medida que se le realizan depósitos, extracciones y transferencias.

3. Genere una estructura de clases donde se pueda observar la herencia simple y se justifique utilizar el polimorfismo.

4. Genere una estructura de clases donde se pueda observar la herencia múltiple y se justifique utilizar el polimorfismo.

5. Desarrollar conjuntos de objetos reales donde para su agrupación y clasificación sea necesario aplicar la categorización conceptual, la clásica y el agrupamiento prototípico. Justifique.

6. Desarrollar un programa que posea una estructura de clases donde se puedan observar dos métodos y el constructor sobrecargados.

7. Desarrollar un programa que posea una estructura de clases donde se pueda observar una propiedad de solo lectura, una propiedad de solo escritura, una propiedad de escritura-lectura, una propiedad de predeterminada y una propiedad con argumentos.