

TEMA 7. Programación Orientada a Objetos.

1. Introducción.

- ¿Qué es una clase?
- ¿Qué es un objeto?

2. Trabajar con Clases

- Crear
- Agregar miembros
- Agregar métodos
- Agregar propiedades
- Crear una instancia
- Constructores y destructores

3. Ejemplos de Clases.

1. INTRODUCCIÓN

Los objetos son fundamentales para la programación en Visual C# .NET. Los formularios, controles y bases de datos son objetos. En este tema, aprenderemos cómo crear nuestros propios objetos desde **clases definidas por nosotros** y cómo podemos utilizar objetos para simplificar la programación e incrementar la reutilización de código.

- ¿Qué es una clase?

Una **clase** es una plantilla o una estructura preliminar de un objeto. Esta estructura preliminar define **atributos** para almacenar datos y define **operaciones** para manipular esos datos. Una clase también define un conjunto de restricciones para permitir o denegar el acceso a sus atributos y operaciones.

Para crear una clase bien diseñada, utilizaremos la *abstracción*. Al implementar la abstracción, definiremos un concepto utilizando un mínimo conjunto de funcionalidades pensadas cuidadosamente que proporcione el comportamiento fundamental de la clase de un modo fácil de utilizar. Por desgracia, no es fácil crear buenas abstracciones de software. Normalmente, requiere un profundo conocimiento del problema que ha de resolver la clase y su contexto, una gran claridad de ideas y mucha experiencia.

El formulario Visual C# .NET con el que hemos estado trabajando es un buen ejemplo de abstracción. Las propiedades esenciales de un formulario, como el título y color de fondo, se han abstraído en la clase **Form**. Algunas operaciones esenciales que se han abstraído son abrir, cerrar y minimizar. Pensemos que la clase Form da una funcionalidad importante que nos permite realizar muchas cosas con un formulario. Esto nos permite utilizar objetos de tipo formulario, sin necesidad de conocer su funcionamiento interno.

La abstracción se garantiza mediante la *encapsulación*. La encapsulación es el empaquetamiento de atributos y funcionalidades para crear un objeto que esencialmente es una “caja negra” (cuya estructura interna permanece privada). Empaquetamos los detalles

2. TRABAJAR CON CLASES.

La plataforma Microsoft .NET proporciona la biblioteca de clases del .NET Framework, pero también nosotros podemos **crear nuestras propias clases**. Este apartado describe cómo crear una nueva clase, agregar miembros de datos, métodos y propiedades a la clase, y establecer modificadores de acceso públicos y privados. Esta lección también describe cómo crear una instancia de una clase y cómo inicializar objetos.

- **Crear una nueva clase.**

Una vez finalizado el proceso de abstracción para determinar las entidades relevantes de un problema determinado, estaremos preparados para crear clases que reflejen las entidades en la abstracción.

1. Abrir o crear un proyecto en Visual Studio .NET (si no hay ninguno abierto).
2. En el menú **Proyecto**, hacer clic en **Agregar clase**.
3. En el cuadro **Agregar nuevo elemento**, escribir el nombre de la nueva clase y hacer clic en **Abrir**. (Le podemos dar nombre a nuestra clase).

Se creará un nuevo fichero en el que se contendrá la nueva clase. El Editor de código proporciona instrucciones de programación que marcan las instrucciones de inicio y final de la clase, como sigue:

```
class tEmpleado
{
}
```

- **Agregar miembros de datos**

Tras agregar una nueva clase a nuestro proyecto, podemos agregar miembros de datos a la clase. Cuando agregamos miembros de datos de una instancia a una clase, especificamos el nivel de acceso estableciendo los modificadores de acceso.

Los miembros de datos de una instancia incluyen variables y constantes miembro. Las variables miembro también se denominan *campos* y pueden ser de cualquier tipo de datos de C#.

Por ejemplo, vamos a agregar dos miembros, el nombre y la edad del empleado:

```
class tEmpleado
{
    // Declaración de miembros de la clase.
    private string mNombre;
    private int mEdad;
}
```

Podemos controlar la accesibilidad de las entidades dentro de esta clase: algunas estarán accesibles únicamente desde dentro y otras desde dentro y desde fuera. Los miembros de la entidad accesibles únicamente desde dentro son **privados**. Para ello utilizamos la palabra reservada **private**.

Los miembros de la entidad accesibles tanto desde dentro como desde fuera son **públicos**. Utilizaremos la palabra reservada **public**.

Como veremos más adelante, lo ideal es que los **miembros (datos) de la clase sean privados** y accedamos a ellos a través de los métodos o propiedades de la misma.

- **Agregar métodos a una clase**

Podemos agregar métodos a una clase.

Un método será un **subprograma** que nos permitirá trabajar con los datos de la clase.

Los miembros o datos de la clase **serán accesibles** desde estos subprogramas, es decir, no hay que pasarlos por parámetros a estas funciones.

Cuando agregamos métodos, especificamos el **nivel de acceso** estableciendo el modificador de acceso (**public** o **private**). Los métodos **public** serán accesibles desde fuera de la clase, mientras que los **private** solo serán accesibles desde otros subprogramas de la clase.

Los métodos tendrán una sintaxis similar a las funciones, que hemos visto en el tema de programación modular.

Por ejemplo, si queremos añadir un subprograma que aumente en un año la edad de la persona.

```
public void cumpleAnyos()  
{  
    mEdad = mEdad + 1;  
}
```

Como se puede observar en el ejemplo, los métodos de una clase **pueden acceder directamente** a los miembros de la misma.

- **Agregar Propiedades**

Podemos agregar propiedades a una nueva clase en nuestro proyecto.

Podemos asignar la propiedad, y podemos recuperar el valor de la propiedad desde la clase.

Las propiedades permiten que una clase exponga una manera pública de obtener y establecer valores, ocultando el código de implementación o comprobación.

Como veremos más adelante, utilizaremos las propiedades para asignar valores a los miembros o datos y para obtener dichos valores.

De esa manera los miembros serán privados, y las propiedades públicas.

Podemos realizar dos tipos de procedimientos en propiedades en Visual C# .NET: **Get** y **Set**.

- El procedimiento **Get** recupera el valor de la propiedad desde la clase. No debería modificar el valor.
- El procedimiento **Set** asigna valor a la propiedad.

Por ejemplo, si queremos que nuestra clase tenga la propiedad nombre que haga referencia al miembro mNombre lo haríamos así:

```
public string Nombre
{
    get { return mNombre; }
    set { mNombre = value; }
}
```

Cuando luego tengamos un objeto de tipo:

```
tEmpleado emp;
```

se ejecutará la parte **set** cuando asignemos un valor a la propiedad:

```
emp.Nombre = "David";
```

y se ejecutará la parte **get** cuando devolvamos el valor de la propiedad:

```
texto = emp.Nombre;
```

Como hemos dicho las propiedades nos permiten realizar código de comprobación. Por ejemplo si quisiéramos hacer una propiedad Edad:

```
public int Edad
{
    get { return mEdad; }
    set
    {
        // No permitimos edades negativas o superiores a 100.
        if (value >= 0 && value <= 100)
            mEdad = value;
    }
}
```

De esta manera evitaríamos que a mEdad se le introdujera un valor fuera del rango 0..100

- **Como crear una instancia de una clase**

Para ejecutar los métodos y utilizar las propiedades de una clase, debemos crear una instancia de la clase.

La instancia de una clase se denomina objeto.

Para crear una instancia de una clase, declaramos una instancia del tipo de la clase y utilizamos la palabra clave **new para crear el nuevo objeto**, como se muestra en la siguiente línea de código:

```
tipoClase nombre_objeto = new tipoClase();
```

Por ejemplo para crear una instancia (objeto) empleado lo podemos hacer de la siguiente forma:

```
tEmpleado emp = new tEmpleado();
```

Podremos acceder a las propiedades y métodos que nos interese de este objeto.

```
emp.Nombre = Interaction.InputBox("Introduzca el nombre");
emp.Edad = int.Parse(Interaction.InputBox("Introduzca la edad"));
emp.cumpleAnyos();
```

- **Constructores**

En Visual C# .NET, la inicialización de nuevos objetos se controla utilizando constructores. Para crear un constructor para una clase, se crea una función pública con el mismo nombre que la clase en cualquier lugar de la definición de la clase.

El constructor tiene las siguientes características:

- ✓ El código del bloque siempre se ejecutará antes de cualquier otro código de una clase.
- ✓ El constructor solo se ejecutará una vez, cuando se cree un objeto.

El siguiente ejemplo muestra cómo hacer el constructor:

```
// Constructor
public tEmpleado()
{
    mNombre = "";
    mEdad = 0;
}
```

La siguiente línea de código crea un objeto de la clase.

```
tEmpleado emp = new tEmpleado();
```

Esto resulta útil si deseamos inicializar nuestro objeto cuando lo creamos.

En una misma clase podemos tener varios constructores (deben tener distintos parámetros):

```
// Constructor
public tEmpleado(string nombre, int edad)
{
    mNombre = nombre;
    mEdad = edad;
}
```

Luego podemos crear un objeto utilizando cualquiera de los constructores:

```
tEmpleado emp = new tEmpleado("David", 47);
```

3. EJEMPLO DE CLASE.

Vamos a ver a continuación un ejemplo completo en el que se declara una clase empleado, con los miembros de datos, sus propiedades y los métodos tanto públicos como privados.

```
class tEmpleado
{
    // Declaración de miembros de la clase.
    private string mNombre;
    private int mEdad;
    // Aquí almacenaremos ventas realizadas por el empleado.
    // Utilizamos el tipo List (igual que ArrayList pero indicando el
    // el tipo de los elementos.
    private List<double> mVentas;

    // Propiedades de la clase
    public string Nombre
    {
        get { return mNombre; }
        set { mNombre = value; }
    }

    public int Edad
    {
        get { return mEdad; }
        set
        {
            // No permitimos edades negativas o superiores a 100.
            if (value >= 0 && value <= 100)
                mEdad = value;
        }
    }

    // Constructor
    public tEmpleado()
    {
        mNombre = "";
        mEdad = 0;
        // Creamos la lista de ventas.
        mVentas = new List<double>();
    }
}
```



```
// Métodos de la clase
// Cumpleaños del empleado
public void cumpleAnyos()
{
    mEdad = mEdad + 1;
}

// Añadir una venta al empleado.
public void anyadirVenta(double venta)
{
    if (venta > 0)
        mVentas.Add(venta);
}

// Función que devuelve un texto con la lista de ventas
// Es privada ya que solo se utilizará dentro de la clase
// (en mostrarDatos)
private string mostrarVentas()
{
    string texto;
    int i;

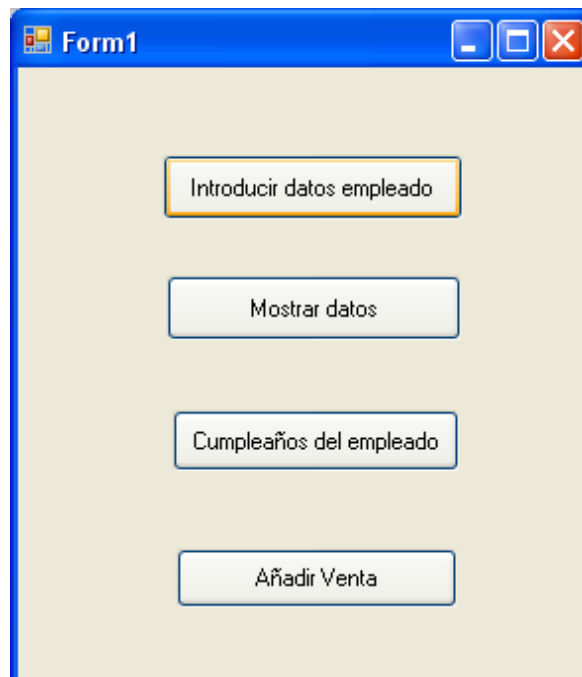
    if (mVentas.Count > 0)
    {
        texto = "Las ventas son : ";
        for (i = 0; i < mVentas.Count; i++)
            texto = texto + mVentas[i] + ", ";

        texto = texto + "\n";
    }
    else
        texto = "Empleado sin ventas.\n";

    return texto;
}
```

```
// Función que devuelve un texto con los datos del empleado.  
// Es pública ya que se llama desde fuera de la clase  
public string mostrarDatos()  
{  
    string texto;  
  
    texto = "Datos del empleado:\n";  
    texto = texto + "Nombre: " + mNombre + "\n";  
    texto = texto + "Edad: " + mEdad + "\n";  
    texto = texto + mostrarVentas();  
  
    return texto;  
}  
}
```

A continuación vemos como se manejaría esta clase en el formulario. En él creamos una instancia u objeto de la clase:



The image shows a screenshot of a Windows application window titled "Form1". The window has a standard Windows XP-style title bar with minimize, maximize, and close buttons. The main area of the form is light beige and contains four rectangular buttons arranged vertically. The buttons are labeled "Introducir datos empleado", "Mostrar datos", "Cumpleaños del empleado", and "Añadir Venta". The buttons have a light blue gradient and a thin border.

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

    tEmpleado emp = new tEmpleado();

    private void bIntroducir_Click(object sender, EventArgs e)
    {
        emp.Nombre = Interaction.InputBox("Introduzca el nombre  
del empleado.");
        emp.Edad = int.Parse(Interaction.InputBox("Introduzca  
la edad del empleado.));
    }

    private void bMostrar_Click(object sender, EventArgs e)
    {
        MessageBox.Show(emp.mostrarDatos());
    }

    private void bCumpleanyos_Click(object sender, EventArgs e)
    {
        emp.cumpleAnyos();
    }

    private void bAnadirVenta_Click(object sender, EventArgs e)
    {
        double venta;

        venta = double.Parse(Interaction.InputBox("Introduzca  
la venta.));
        emp.anyadirVenta(venta);
    }
}
```

En este código dentro del formulario empezamos a darnos cuenta de la potencialidad de la programación orientada a objetos y de la **facilidad** con la que podemos utilizar un objeto de una clase en la cual hemos hecho una buena abstracción.

Al encapsular el funcionamiento interno de la clase, el programador que utiliza un **objeto** de la clase tEmpleado únicamente se preocupa de **utilizar los métodos públicos sin necesidad de conocer internamente la clase**.

Por ejemplo, para añadir una venta al empleado sabe que debe llamar al método `emp.anyadirVenta(venta)`, **sin preocuparse de cómo se gestionan** internamente esas ventas, si con un vector, con una lista, o si se guardan en una base de datos...

O cuando sea el cumpleaños del empleado simplemente llamamos a `emp.cumpleAnyos()`. Si en el futuro quisiéramos como programadores de la clase aumentar la funcionalidad de ese método `cumpleAnyos`, lo haríamos en la clase y ese cambio afectaría a todos los objetos.

- **Trabajar con una lista de objetos**

Imaginemos que a continuación queremos cambiar la funcionalidad del problema, de manera que trabajemos con una lista de empleados.

Vamos a crear una lista a la que vayamos añadiendo objetos de tipo empleado:

```
List<tEmpleado> listaEmp = new List<tEmpleado>();

private void bIntroducir_Click(object sender, EventArgs e)
{
    int i;
    tEmpleado emp;

    // Creamos un nuevo empleado.
    emp = new tEmpleado();
    emp.Nombre = Interaction.InputBox("Introduzca el nombre
                                     del empleado.");
    emp.Edad = int.Parse(Interaction.InputBox("Introduzca
                                              la edad del empleado.));

    // Lo añadimos a la lista de empleados.
    listaEmp.Add(emp);
}

private void bMostrar_Click(object sender, EventArgs e)
{
    int i;
    tEmpleado emp;
    string texto;

    texto = "Los empleados de la empresa son :\n";
    for (i = 0; i < listaEmp.Count; i++)
    {
        // Asignamos a emp el empleado de la posición i.
        emp = listaEmp[i];
        texto = texto + emp.mostrarDatos();
    }

    MessageBox.Show(texto);
}
```

Es interesante notar un par de puntos en el código que acabamos de ver. El primero es que al leer los empleados nuevos hacemos: `emp = new tEmpleado();`.

Esto es así porque cada empleado que leemos es nuevo y **necesitamos crearlo** con el constructor **new**. Luego lo añadimos a la lista.

Sin embargo, cuando estamos mostrando los empleados no tenemos que hacer new ya que los empleados ya están creados en el vector. Al hacer `emp = listaEmp[i];` no estamos copiando el empleado en emp sino que **estamos haciendo referencia** al mismo.

Tener en cuenta que los objetos son referencias, es decir, al hacer `emp = listaEmp[i];` lo que estamos haciendo es que emp “apunte” al empleado que hay en la posición i.

Si modificáramos el contenido de emp se modificaría el contenido del empleado i ya que son **el mismo objeto**.

Veámoslo en el siguiente ejemplo.

Queremos que sea el cumpleaños de un empleado. Para ello pedimos su nombre, lo buscamos y a continuación llamamos al método cumpleaños de ese objeto.

```
// Función que devuelve la posición en la que se encuentra
// el empleado con el nombre.
// Si no lo encuentra devuelve -1
int buscarEmpleado(List<tEmpleado> listaEmp, string nombre)
{
    int pos, i;
    tEmpleado emp;

    pos = -1;
    i = 0;
    while (i < listaEmp.Count && pos == -1)
    {
        emp = listaEmp[i];
        if (emp.Nombre == nombre)
            pos = i;

        i++;
    }

    return pos;
}
```

```
private void bCumpleanyos_Click(object sender, EventArgs e)
{
    int pos;
    tEmpleado emp;
    string nombre;

    nombre = Interaction.InputBox("Introduzca el nombre  
del empleado del cumpleaños");
    pos = buscarEmpleado(listaEmp, nombre);
    if (pos != -1)
    {
        emp = listaEmp[pos];
        emp.cumpleAnyos();
    }
    else
        MessageBox.Show("No existe empleado con ese nombre.");
}
```

A la hora de mostrar todos los empleados de la lista también podríamos haber utilizado la estructura foreach:

```
private void bMostrar_Click(object sender, EventArgs e)
{
    string texto;

    texto = "Los empleados de la empresa son :\n";
    foreach(tEmpleado emp in listaEmp)
    {
        texto = texto + emp.mostrarDatos();
    }

    MessageBox.Show(texto);
}
```

Otra posible forma de haber encarado este problema, ya que estamos trabajando la Programación Orientada a Objetos, podría haber sido **crear una clase tListaEmpleados** en la que el **List** fuera un **miembro** de esa clase.

Esto lo estudiaremos durante la realización de los **ejercicios**. En concreto, a partir del ejercicio 4 (que tenéis resuelto).