CLASES Y OBJETOS

Guía de laboratorio – Capítulo VIII

Contenidos:

- ✓ Creación de una clase.
- ✓ Objeto (instancia de clase).
- ✓ Propiedades y métodos.

Fuentes: B. Douglas & P. Mark (2010). "C# para estudiantes". Pearson Educación. DF México.

CONCEPTO DE CLASE

Como hemos visto anteriormente, una clase describe cualquier cantidad de objetos que pueden fabricarse a partir de ella mediante la palabra clave **new**, por ejemplo:

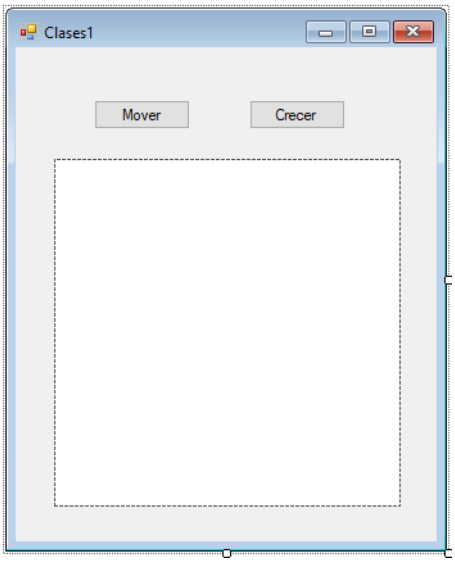
```
Form2 F2 = new Form2();

Pen lapiz = new Pen(Color.Black);
```

En guías anteriores vimos cómo utilizar las bibliotecas de clases, ya sea seleccionando controles del cuadro de herramientas o codificándolas de manera explícita, como en los ejemplos anteriores. En esta guía veremos de qué manera escribir nuestras propias clases.

Las clases consisten en:

- datos **private** (variables) que contienen información sobre el objeto;
- opcionalmente uno o más métodos constructores, que se utilizan al momento de crear un objeto; su propósito es llevar a cabo cualquier inicialización, por ejemplo, asignando valores iniciales a las variables del objeto;
- métodos **public** que el usuario del objeto puede invocar para llevar a cabo funciones útiles;
- propiedades que nos permiten acceder o modificar las características de un objeto;
- métodos **private**, que se utilizan sólo dentro del objeto y son inaccesibles desde fuera de él.

Creando una clase



Cuando el programador piensa en la creación de un nuevo programa, tal vez contemple la necesidad de incluir un objeto que no esté disponible en la biblioteca de clases de C#.

Para ejemplificar esto, comenzaremos un programa cuyo objetivo es desplegar y manipular un globo, y representaremos el globo como un objeto (un círculo dentro de un cuadro de imagen). Además, pondremos dos botones para cambiar la posición y el tamaño del globo:

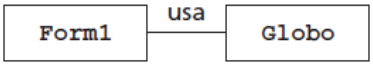
Los objetos a agregar y sus propiedades serán mínimamente los siguientes:

button1	
Name	btnMover
Text	"Mover"
Button2	
Name	btnCrecer
Text	"Crecer"
pictureBox1	
Name	picDibujo
BackColor	White

Construiremos este programa a partir de dos clases:

- La clase **Globo** proveerá los métodos denominados **Mover** y **CambiarTamaño**, cuyos comportamientos son obvios.
- La clase **Form1** provee la GUI del programa. Esta clase utilizará la clase **Globo** según sea necesario.

Ambas clases se muestran en el siguiente **diagrama de clases**:



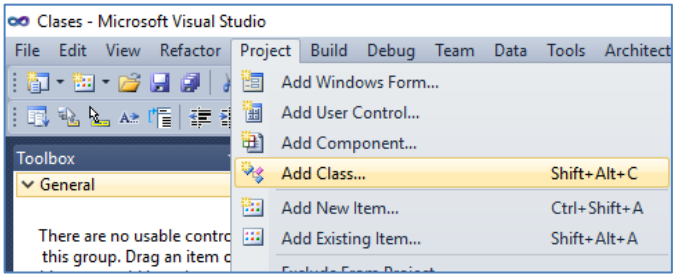
Estos diagramas representan las clases mediante rectángulos. Cualquier conexión que exista entre ellas se ilustra a través de líneas de unión. En este caso la relación se explica con una anotación arriba de la línea.

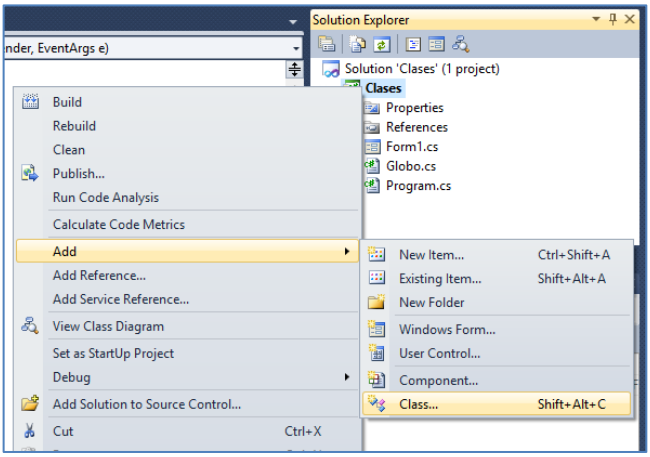
Una vez creada la GUI la clase **Form1**, creamos la clase **Globo**.

¿En dónde debemos colocar las clases? Una opción consiste en escribir todas las clases y colocarlas en un solo archivo, pero es una mejor práctica ponerlas en archivos diferentes.

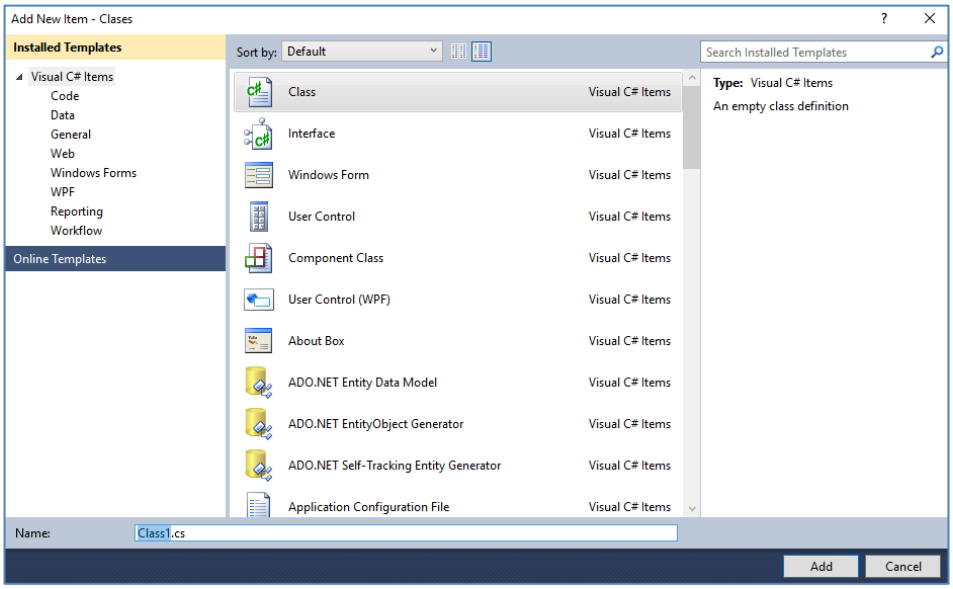
El IDE provee una herramienta para ayudarnos a hacerlo. Los pasos son:

Seleccionar **Agregar clase** del menú **Proyecto**, o clic derecho sobre el nombre del proyecto en el explorador de soluciones, opción **Agregar** y luego **Clase**:



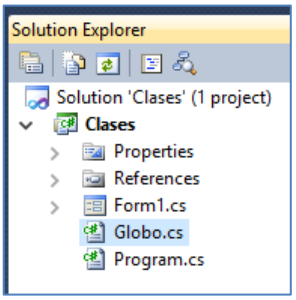


Revisar que esté seleccionado el ítem **Clase** y en el campo **Nombre**, modificar el predeterminado (en este caso la llamaremos “Globo”). Hacer clic en **Agregar**:



Este procedimiento crea un archivo distintivo que contendrá el código de la clase. Este archivo es parte del proyecto del programa, de manera que se compila y vincula automáticamente al ejecutarlo.

El resultado en el explorador de soluciones será el siguiente:



La estructura general de las clases es:

public class Globo

{

// variables de instancia

// propiedades

// métodos

}

El código para la clase **Globo**:

```
public class Globo
{
    //variables de clase o de instancia
    private int x = 50;
    private int y = 50;
    private int diametro = 20;

    //métodos
    public void MoverDerecha(int pasoX)
    {
        x = x + pasoX;
    }

    public void CambiarTamaño(int cambio)
    {
        diametro = diametro + cambio;
    }

    public void Mostrar(Graphics areaDibujo)
    {
        Pen lápiz = new Pen(Color.Black);
        areaDibujo.DrawEllipse(lápiz, x, y, diametro, diametro);
    }
}
```

```
using System.Drawing; //agregar esta biblioteca de clases en la Clase Globo o escribir el método Mostrar de la siguiente forma:

        public void Mostrar(System.Drawing.Graphics áreaDibujo)
        {
            System.Drawing.Pen lápiz = new System.Drawing.Pen(System.Drawing.Color.Black);
            áreaDibujo.DrawEllipse(lápiz, x, y, diametro, diametro);
        }
```

El encabezado de la descripción de la clase empieza con las palabras clave **public** y **class**, y proporciona el nombre de la misma. La descripción completa termina con una llave. La convención de C# (y de casi todos los lenguajes orientados a objetos) es que el nombre de las clases empieza con mayúscula. El cuerpo de la clase consiste en las declaraciones de las variables, los métodos y las propiedades. Observemos cómo mejora la legibilidad de la clase gracias al uso de sangría y líneas en blanco.

En este caso, las variables de clase son requeridas en los métodos propios de la clase Globo sin necesidad, por el momento, de crear **propiedades** para permitir que otro objeto o clase pueda acceder a su valor.

Variables private

El globo de nuestro ejercicio anterior tiene asociados ciertos datos: su tamaño (diametro) y posición (como coordenadas *x* y *y*). El objeto globo debe recordar esos valores, lo cual se logra guardándolos en variables que se describen de la siguiente forma:

```
private int x = 50;
private int y = 50;
private int diametro = 20;
```

Las variables **diametro**, **x** y **y** se declaran en la parte superior de la clase. Cualquier instrucción dentro de la clase puede acceder a ellas. Se denominan *variables a nivel de clase* o *variables de instancia*.

Las variables a nivel de clase casi siempre se declaran como **private**. Aunque *podríamos* describir estas variables como **public**, por lo general se considera una mala práctica. Es mejor dejarlas como **private** y usar propiedades o métodos para acceder a sus valores, como veremos más adelante.

Métodos public

Ciertas características de los objetos deben estar públicamente disponibles para otras secciones del programa.

Esto incluye los métodos diseñados para que otros métodos los utilicen. Como hemos visto antes, el globo tiene asociadas ciertas acciones, por ejemplo, la que permite modificar su tamaño.

Estas acciones se escriben como métodos. Para cambiar el tamaño, por ejemplo, utilizamos el siguiente código:

```
public void CambiarTamaño(int cambio)
{
    diametro = diametro + cambio;
}
```

Para indicar que está públicamente disponible, debemos anteponer al encabezado del método la palabra clave **public**.

Se utiliza un método públic también para para mover el globo, y para completar la clase creamos un método que permitirá que el globo se muestre cada vez que se lo soliciten.

Hemos marcado claramente la diferencia entre los elementos públicamente disponibles y los que son privados. Éste es un importante ingrediente de la filosofía de la programación orientada a objetos.

Los datos (variables) y las acciones (métodos) están agrupados, pero de tal forma que **parte de la información queda oculta al mundo exterior**. Por lo general son los datos los que se ocultan. Esto se denomina *encapsulamiento* u *ocultamiento de información*.

Desde la perspectiva del programador encargado de escribirlos, las clases u objetos tienen la estructura general que consistente en variables, propiedades y métodos.

Sin embargo, los usuarios ven el objeto, que les provee un servicio, de manera muy distinta: sólo los elementos públicos (por lo general métodos y propiedades) son visibles; todo lo demás está oculto en una caja impenetrable.

Una vez analizado el código de nuestra clase Globo, volvemos a nuestra clase **Form1** donde comenzaremos a crear las interacciones con la clase creada.

Declaramos las variables de instancia en la parte superior de la clase **Form1** (afuera de los métodos), incluyendo una variable llamada **globo**:

```
public partial class Form1 : Form
{
    Globo globo;
    Graphics áreaDibujo;

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

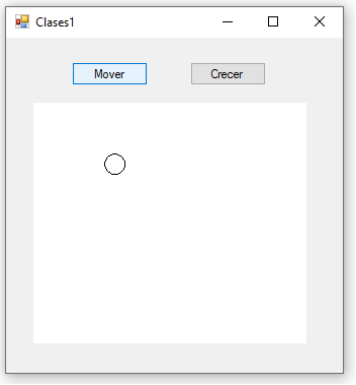
Dentro del método constructor de **Form1** realizamos la inicialización necesaria, incluyendo la creación de una nueva instancia de la clase **Globo**. Éste es el punto crucial en el que creamos un objeto a partir de nuestra propia clase. La idea de instanciar tanto el objeto globo, como el área de dibujo dentro del constructor es que al iniciar la clase Form1 ya se crea la estructura completa de la misma y estos dos objetos forman parte de ella.

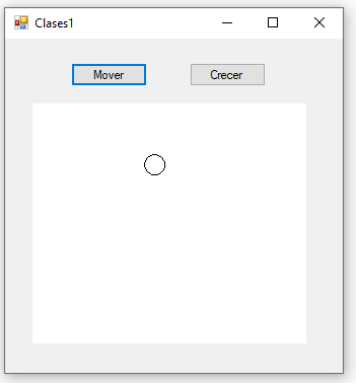
```
public Form1()
{
    InitializeComponent();
    globo = new Globo();//creamos la instancia/objeto globo
    areaDibujo = picDibujo.CreateGraphics();//nuestro pictureBox será el área de dibujo
}
```

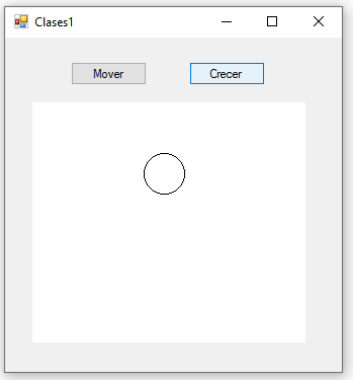
Ahora viene el código para responder a los eventos de clic de cada botón:

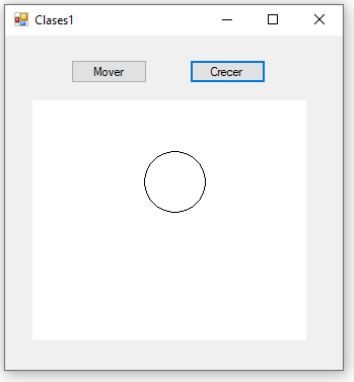
```
private void btnMover_Click(object sender, EventArgs e)
{
    globo.MoverDerecha(20); //llamamos al método MoverDerecha de la clase Globo
    areaDibujo.Clear(Color.White); //limpiamos el área de dibujo
    globo.Mostrar(areaDibujo); //llamamos al método Mostrar de la clase Globo
}

private void btnCrecer_Click(object sender, EventArgs e)
{
    globo.CambiarTamaño(20);
    areaDibujo.Clear(Color.White);
    globo.Mostrar(areaDibujo);
}
```

El resultado de este código será el siguiente:









Propiedades

Anteriormente comentamos que hacer **public** las variables de instancia constituye una mala práctica de programación, y como bien vimos en la guía de envío de datos entre formularios, las propiedades pueden utilizarse como un mecanismo para otorgar a los usuarios un acceso conveniente pero controlado a los datos de un objeto. De hecho, hemos utilizado ya las propiedades de los componentes del cuadro de herramientas y sabemos que, por ejemplo, podemos escribir instrucciones para acceder a los valores de las propiedades de un cuadro de texto:

```
nombre = textBox1.Text;
textBox1.Visible = false;
```

Éstos son ejemplos de cómo acceder a los datos que forman parte de un objeto, pero hay que recordar que existen dos tipos distintos de acceso:

- leer un valor: a esto se le conoce como acceso ***get*** (por ejemplo, extraer el texto de un cuadro de texto utilizando la propiedad **Text**);
- escribir el valor: a esto se le conoce como acceso ***set*** (por ejemplo, cambiar la propiedad **Visible** de un cuadro de texto).

Por ejemplo, supongamos que queremos permitir que el usuario de un objeto globo haga referencia (**get**) a la coordenada *x* del globo. Este valor se guarda en la variable llamada **x**, establecida en la parte superior de la clase.

Imaginemos que el usuario del objeto globo (**globo**) quiere ser capaz de extraer y utilizar el valor, como en el siguiente ejemplo:

```
textBox1.Text = Convert.ToString(globo.CoordX);
```

Supongamos también que, como programadores, deseamos que el usuario pueda cambiar (**set**) el valor de la coordenada *x* con una instrucción como la siguiente:

```
globo.CoordX = 56;
```

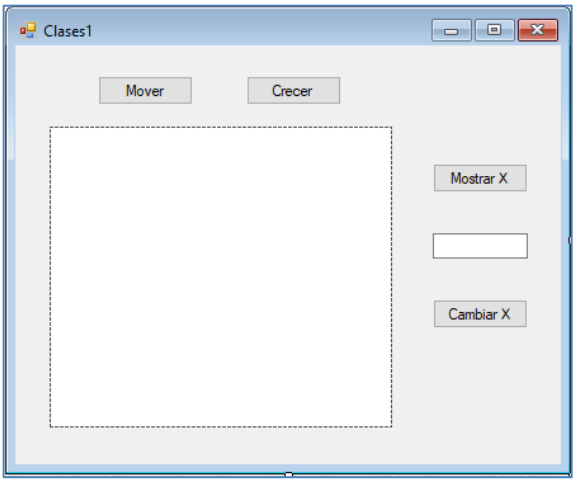
La manera de proporcionar estas herramientas es mediante una propiedad. Por lo que la clase **Globo**, deberá incluir el código de la propiedad:

```
public class Globo
{
    private int x = 50;
    private int y = 50;
    private int diametro = 20;

    public int CoordX //Propiedad que representa la variable x
    {
        get //lectura
        {
            return x;
        }
        set //escritura
        {
            x = value;
        }
    }
    //acá van los métodos de la clase globo: Mostrar, MoverDerecha y CambiarTamaño
}
```

Como mencionamos en la guía de envío de datos entre formularios, el encabezado de las propiedades es similar al que se utiliza en los métodos, excepto que no hay paréntesis para especificar parámetros.

La descripción consiste en dos componentes complementarios: uno con **get** y el otro con **set** como encabezados. Cada componente termina con su respectiva llave de cierre. La parte **get** es como un método: devuelve el valor deseado; la parte **set** es como un método: asigna el valor utilizando la palabra clave especial **value**, como se muestra en el ejemplo anterior.

Ahora podemos utilizar estas propiedades en el programa que emplea esta clase. Lo mejoraremos de manera que haya un botón para desplegar el valor de la coordenada *x* en un cuadro de texto.



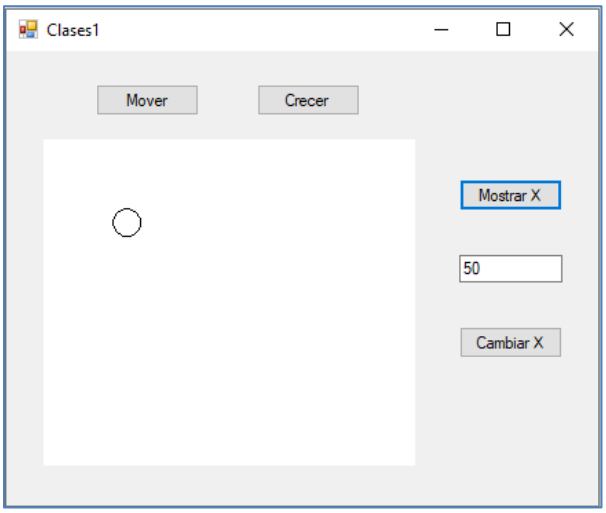
El siguiente es el código que responde a un clic del botón para mostrar la coordenada *x*; en él se utiliza la propiedad **get**:

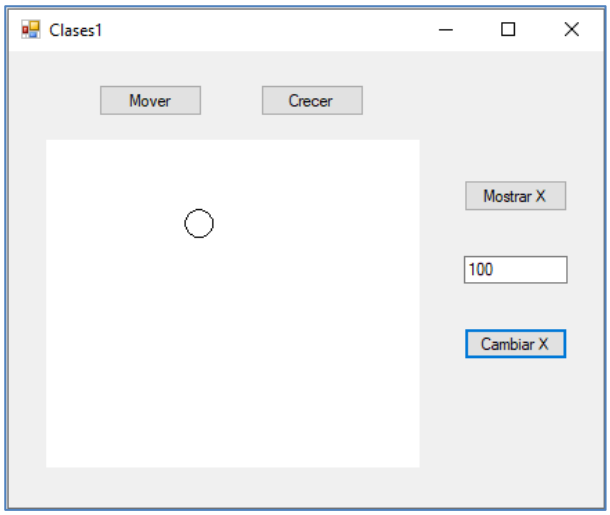
```
private void btnMostrar_Click(object sender, EventArgs e)
{
    areaDibujo.Clear(Color.White); //limpia el área de dibujo
    textBox1.Text = Convert.ToString(globo.CoordX); //mostramos el contenido de CoordX
    globo.Mostrar(areaDibujo); //llamamos al método Mostrar de la clase Globo
}
```

Este programa cuenta también con un botón para modificar la coordenada *x* (que se introduce en el cuadro de texto) mediante la propiedad **set** de la clase **GloboConPropiedades**. El código que responde a un clic en este botón es:

```
private void btnCambiar_Click(object sender, EventArgs e)
{
    areaDibujo.Clear(Color.White);
    globo.coordX = Convert.ToInt32(textBox1.Text); //asignamos el valor del text a CoordX
    globo.Mostrar(areaDibujo);
}
```

El resultado del programa será el siguiente: podremos ver la coordenada X actual del "globo" o modificarla:





Si sólo necesitamos poner a disposición del usuario un mecanismo para ver una propiedad (sin darle la posibilidad de modificar su valor), escribimos la declaración de la propiedad de la siguiente manera: **sólo lectura**

```
public int CoordX
{
    get
    {
        return x;
    }
}
```

Por otra parte, si deseáramos permitirle cambiar un valor, pero evitando que pueda verlo, escribiríamos lo siguiente: **sólo escritura**

```
public int CoordX
{
    set
    {
        x = value;
    }
}
```

Sabemos que sería más sencillo declarar el valor **x** como **public**. De esa manera el usuario del objeto sólo tendría que referirse al valor como **globo.x**. Esto es posible obviamente, pero constituye una muy mala práctica. En cambio, como ya hemos visto, hay varias razones por las que es preferible utilizar propiedades:

- La clase puede ocultar a los usuarios la representación interna de los datos, sin privarlos de la interfaz externa. Por ejemplo, el autor de la clase globo podría optar por mantener las coordenadas del centro de un globo, pero proveer a los usuarios las coordenadas de la esquina superior izquierda del cuadrado que lo contiene.
- El autor de la clase puede optar por restringir el acceso a los datos. Por ejemplo, la clase podría restringir el valor de la coordenada *x* para que tuviera acceso de sólo lectura (**get**), deshabilitando el acceso de escritura (**set**).
- La clase puede validar o comprobar los valores utilizados. Por ejemplo, podría ignorar un intento de proporcionar un valor negativo para una coordenada.

¿Método o propiedad?

Tanto los métodos como las propiedades proporcionan mecanismos para acceder a un objeto. Entonces ¿cómo elegimos cuál usar? Utilizamos los métodos cuando queremos que un objeto realice cierta acción (por lo general los nombres de los métodos son verbos).

En cambio, empleamos propiedades cuando deseamos hacer referencia a cierta información asociada a un objeto (generalmente los nombres de las propiedades son sustantivos).

Esto queda bien ilustrado en las bibliotecas de clases. Por ejemplo, los cuadros de texto tienen los métodos **Clear** (borrar) y **AppendText** (anexar texto) para realizar acciones, y cuenta con las propiedades **Text** (texto) y **Visible** para referirse al estado del objeto.

Algunos ejemplos de métodos asociados al globo de nuestro ejercicio anterior podrían ser: **Mostrar**, **MoverArriba**, **MoverAbajo**, **MoverIzquierda**, **MoverDerecha**, **ReducirTamaño** y **AumentarTamaño**, mientras que algunos ejemplos de propiedades serían: **CoordX**, **CoordY**, **Diametro**, **Color** y **Visible**.

En ocasiones se hace necesario elegir entre utilizar una propiedad o crear un método para cumplir un propósito determinado; en este sentido, la decisión es cuestión de estilo. Por ejemplo, para cambiar el color de un componente podríamos crear un método llamado **CambiarColor**, aunque también podríamos usar una propiedad denominada **Color** para el mismo fin.

Constructores

Al crear un objeto (digamos, globo), es necesario que se de valor a su posición y tamaño. A esto se le conoce como inicializar las variables. Hay dos formas de llevar a cabo esta labor. Una de ellas consiste en incluir la inicialización como parte de la declaración de las variables a nivel de clase, como ya hemos hecho:

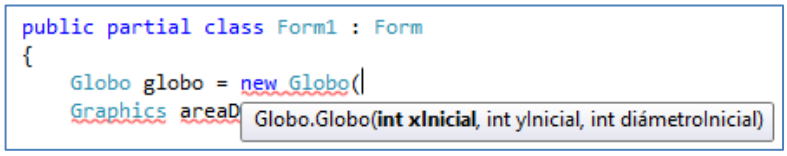
```
private int x = 50;
private int y = 50;
private int diametro = 20;
```

Otra manera de inicializar un objeto es escribir un método especial, conocido como *método constructor* o simplemente *constructor* (ya que está involucrado en la construcción del objeto). Este método siempre debe tener el mismo nombre que la clase; no tiene valor de retorno, y puede o no contar con parámetros.

Anteriormente hemos conocido y modificado el constructor por defecto que se crea en cada clase Form, pero en esta oportunidad crearemos un constructor para la clase **Globo**:

```
public Globo(int xInicial, int yInicial,int diametroInicial)
{
    x = xInicial;
    y = yInicial;
    diametro = diametroInicial;
}

private int x, y, diametro;
```

Teniendo el mismo nombre que la clase y sabiendo que se ejecuta antes que todo, podemos entonces aprovecharlo de la siguiente forma:




```
Globo globo = new Globo(30,30,50);
```

Otras acciones que podría realizar un método constructor incluyen la creación de otros objetos que el objeto en cuestión utilice, o la apertura de un archivo que emplee.

Si una clase carece de un constructor explícito, C# asume que tiene un solo constructor con cero parámetros, al cual se le conoce como constructor predeterminado.

Múltiples constructores

Una clase puede tener cero, uno o varios métodos constructores. Si tiene uno o más constructores, por lo general éstos llevan parámetros y deben invocarse con los parámetros apropiados.

Por ejemplo, en la clase **Globo** podríamos escribir los siguientes dos constructores:

```
public Globo(int xInicial, int yInicial,int diámetroInicial)
{
    x = xInicial;
    y = yInicial;
    diametro = diámetroInicial;
}

public Globo(int xInicial, int yInicial)
{
    x = xInicial;
    y = yInicial;
}
```

Lo cual nos permitirá crear, desde nuestro Form1, estas dos formas de objetos globo:

```
Globo globo1 = new Globo(30,30,50);
Globo globo2 = new Globo(70, 70);
```

Destrucción de objetos

Ya vimos cómo crear objetos mediante el uso de la poderosa palabra **new**. ¿Pero cómo deshacernos de ellos? Una respuesta obvia y certera sería que dejan de funcionar cuando el programa termina de ejecutarse. También es cierto que lo hacen cuando el programador deja de utilizarlos. Por ejemplo, si hacemos lo siguiente para crear un nuevo objeto:

```
Globo globo;
globo = new Globo(20, 100, 100);
//y después:
globo = new Globo(40, 200, 200);
```

Lo que ocurre es que el primer objeto creado con **new** tiene una existencia muy corta, ya que desaparece cuando el programa ya no tiene conocimiento de él y su valor es reemplazado por el objeto más reciente.

Cuando se destruye un objeto o instancia, la memoria que se utilizaba para almacenar los valores de sus variables y cualquier otro recurso es reclamada para que el sistema en tiempo de ejecución la emplee en otros procesos. A esto se le conoce como *recolección de basura*. En C# la recolección de basura es automática (a diferencia de lo que ocurre en otros lenguajes, como C++, en los que el programador tiene que llevar cuenta de los objetos que ya no son necesarios).

Por último, podemos destruir un objeto al asignarle el valor **null**; por ejemplo:

```
globo = null;
```

La palabra clave **null** de C# describe un objeto no existente (no instanciado).

Métodos y propiedades static

Algunos métodos no necesitan un objeto para trabajar. Por ejemplo, métodos matemáticos como los utilizados para obtener la raíz cuadrada (**Sqrt**), el seno de un ángulo (**Sin**) o el método para redondear (**Round**) se proporcionan dentro de una biblioteca de clases llamada **Math**. Para utilizarlos en un programa escribimos instrucciones como:

```
double x, y;
x = Math.Sqrt(y);
```

En esta instrucción hay dos variables **double** llamadas **x** y **y**, pero no hay objetos. Tenga en cuenta que **Math** es el nombre de una clase, y no de un objeto. El método de raíz cuadrada **Sqrt** actúa sobre su parámetro **y**. La pregunta es: si **Sqrt** no es método de un objeto, entonces ¿qué es? La respuesta es que los métodos como éste forman parte de una clase, pero se describen como **static**. Al utilizar uno de estos métodos hay que anteponer a su nombre el de la clase a la que pertenece.

La clase **Math** tiene la siguiente estructura, en la que los métodos se etiquetan como **static**:

```
public class Math
{
    public static double Sqrt(double x)
    {
        // cuerpo de Sqrt
    }
    public static double Sin(double x)
    {
        // cuerpo de Sin
    }
}
```

ToInt32 es otro ejemplo de un método **static**, en este caso, dentro de la clase **Convert**. Un ejemplo de una propiedad **static** se encuentra en la clase **Color**: los diversos colores (como **Color.White**, **Color.Black**, etc.) están disponibles para que otras clases los utilicen.

¿Cuál es el propósito de los métodos **static**? En la programación orientada a objetos todo se escribe como parte de una clase; no existe nada fuera de ellas. Pensemos en la clase **Globo**; ésta contiene variables **private**, como **x** y **y**, que registran el estado de un objeto. En cambio, los métodos independientes como **Sqrt** no involucran un estado y no forman obviamente parte de una clase; no obstante, deben obedecer la regla central de la programación orientada a objetos: formar parte de una clase. Esta es la razón de ser de los métodos **static**.

8.1. **Ejercicio práctico (no entregable):** Dados los siguientes bloques de código, ubicarlos en sus respectivos métodos y clases para realizar el programa: (Las clases son **Form 1** y **Dibujo**)

Métodos	`public void DibujarTablero(Graphics areaDibujo)` `{}`
	`public Form1()` `{}`
	`private void btnLineas_Click(object sender, EventArgs e)` `{}`
	`public Dibujo(int xInicial, int yInicial, int diametroInicial)` `{}`
	`private void btnCirculos(object sender, EventArgs e)` `{}`
	`public void DibujarCirculo(Graphics áreaDibujo, int numero)` `{}`
Códigos	`int cantCirculosDibujados = 0;` `Dibujo dibujo = new Dibujo(10, 10, 80);`

	`Pen lápiz = new Pen(Color.Black, 2);`
	`if (numero != 0)`
	`{ x = x + 100; }`
	`if (numero == 3)`
	`{`
	`x = 10;`
	`y = 110;`
	`}`
	`if (numero == 6)`
	`{`
	`x = 10;`
	`y = 210;`
	`}`
	`áreaDibujo.DrawEllipse(lápiz, x, y, diametro, diametro);`

	`dibujo.DibujarTablero(areaDibujo);`

	`Pen lápiz = new Pen(Color.Black, 2);`
	
	`areaDibujo.DrawLine(lápiz, 100, 0, 100, 300);`
	`areaDibujo.DrawLine(lápiz, 200, 0, 200, 300);`
	`areaDibujo.DrawLine(lápiz, 0, 100, 300, 100);`
	`areaDibujo.DrawLine(lápiz, 0, 200, 300, 200);`

	`Graphics areaDibujo;`
	`dibujo.DibujarCirculo(areaDibujo, cantCirculosDibujados);` `cantCirculosDibujados = cantCirculosDibujados + 1;`

```
InitializeComponent();
areaDibujo = picDibujo.CreateGraphics();
```

```
x = xInicial;
y = yInicial;
diametro = diametroInicial;
```

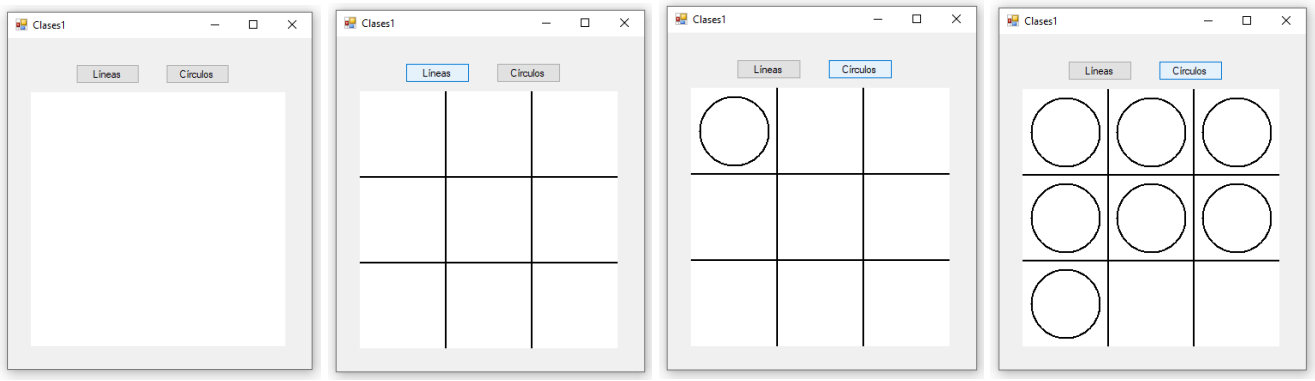
```
private int x, y, diametro;
```

Tener en cuenta que una vez ubicadas las líneas de código dentro (o fuera) de cada método, aún quedará al menos un error para solucionar, pero todo se encuentra en esta guía.

Las siguientes pantallas muestran cómo está conformada la interfaz y cómo debería funcionar:

El botón Líneas dibuja las líneas del pictureBox con un solo clic.

El botón Círculos, dibuja un círculo por cada vez que se presione clic.



8.2. **Ejercicio práctico (no entregable):** Crear un programa en el que se registren nombre, apellido y teléfono de personas y se concatenen los datos para agregarlos como elementos en un ListBox o DataGridView (pueden reutilizar y modificar el programa que vienen trabajando de carga de personas).

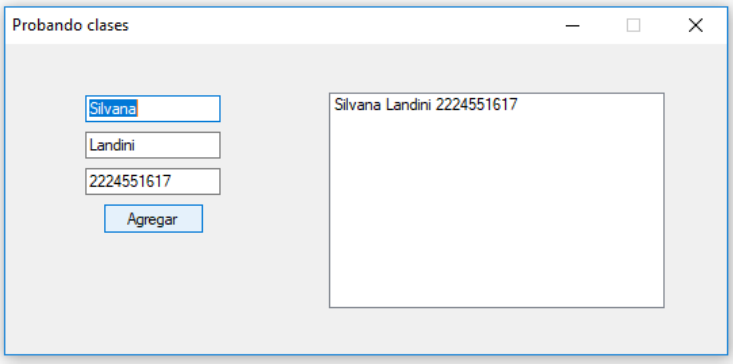
Utilizar los descriptores de acceso get y set.

Es imprescindible que el programa incluya una clase llamada **Persona.cs**, la cual debe tener 3 argumentos y sus propiedades: nombre, apellido y teléfono.

Dichas propiedades son las que van a almacenar los datos de cada persona cargada por el usuario. Las mismas deberán ser de lectura y escritura en los tres casos ya que se les va a asignar un valor al momento de cargar una nueva persona y luego se va a obtener su valor para poder agregar el nuevo registro en el objeto lista o grilla.

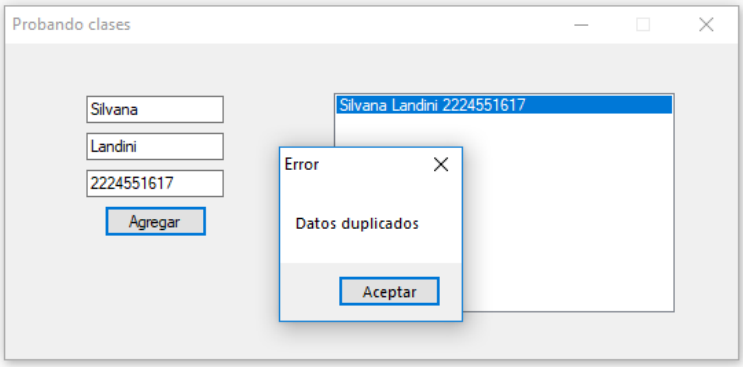
Debe tener al menos un método y en lo posible estar dentro de la clase mencionada.

Diseño a criterio de cada uno (la imagen es sólo demostrativa)



Al momento de ingresar los datos, se debe controlar que no haya repeticiones exactas, de lo contrario mostrar mensaje de error:

(Diseño del mensaje también a criterio de cada uno)



Esta validación podría ser un método en la clase personas, por ejemplo.

Debe permitir eliminar elementos de la lista haciendo doble clic sobre el mismo y preguntando mediante un mensaje si está seguro de borrarlo.