**APUNTES PROGRAMACIÓN**

**PRIMER TRIMESTRE**

* **Trabajo con Aplicación de Formularios de Windows:**
* **TextBox.Text:** Capta los datos introducidos por el usuario en un TextBox o Label según la propiedad Text:

string texto = txt1.Text;

* **Castear tipos de datos:** Convertir un tipo de dato en otro:
* **Pasar de string a int/float/double:** Método Parse():

int numero = int.Parse(txt1.Text);

* **Pasar int/float/double a string:** Método ToString():

string texto = numero.ToString()[[1]](#footnote-1);

* **MessageBox.Show():** Muestra alerta en pantalla con el texto entre paréntesis:

int numero 1;

MessageBox.Show(“El número es ” + numero);

* **Excepciones:** Errores en tiempo de ejecución del programa. Se capturan usando un bloque try-catch:
* **Error de formato:**

try

{

int numero = txt.Text;

}

catch (FormatException fEx)

{

MessageBox.Show(“Se ha producido el error: ” + fEx.Message);

}

* **Error al dividir por cero:**

catch (DivideByZeroException oEx)

{

MessageBox.Show(“Se ha producido el error: ” + oEx.Message);

}

* **Interaction.InputBox():** Crea cuadros de diálogo para que el usuario introduzca varios valores de forma consecutiva. Proceso para crear esta herramienta:
* **Explorador de soluciones:** Agregar referencia -> Microsoft.VisualBasic.
* **Añadir librería al código:** using Microsoft.VisualBasic.
* **Añadir la funcionalidad al código:**

string texto = Interaction.InputBox(“Texto”, “Título”, “Valor por defecto”);

* **Castear el valor introducido por el usuario a otro tipo de dato:**

int numero = int.Parse(Interaction.InputBox(“Introducir número”));

* **Otros:**
* **Añadir salto de línea:** “\n”;
* **Enfocar un textBox:** txtBox.Focus();
* **Se puede indicar el formato de string usando el método ToString():**
  + **ToString(“0.##”):** Formato #.##.Indica 2 decimales.
  + **ToString(“0#”):** Formato ## (ejemplo, 01, 22). Útil para la hora.
* **Cambiar un número positivo a negativo:** Multiplicar por -1.
* **Programación modular:**

**Parámetros**   **≠** **Argumentos**

↓ ↓

Los valores que espera Los valores que efectivamente

recibir la función recibe la función

* **Diferencias entre parámetros por valor y por referencia:**
* **Por valor:** Las variables no cambian su valor al pasar por una función (solo datos de entrada).
* **Por referencia:** Las variables cambian o reciben un valor al pasar por una función (datos de entrada y salida o de salida). Tipos de valor por referencia:
  + **Tipo ref:** Variable ya creada que se pasa como argumento a una función y que cambia su valor al terminar la ejecución (dato de entrada y salida).
  + **Tipo out:** Variable que se crea como parámetro de una función void y que toma un valor al ejecutarse la función (dato de salida). Es útil para **devolver varios valores** al pasarse por una función (por ejemplo, cociente y resto).
* Ejemplo de función con tipo out:

private void calcularMedia (double n1, double n2, double n3, **out** double media)

{

media = (n1 + n2 + n3) / 3;

}

private void Click (…)

{

double n1 = double.Parse(txtNota1.Text);

double n2 = double.Parse(txtNota2.Text);

double n3 = double.Parse(txtNota3.Text);

double media;

calcularMedia(n1, n2, n3, **out** media);

MessageBox.Show(“La media es “ + media);

}

* **Programación recursiva vs Programación iterativa:**
* **Caso base:** Hace que termine la recursividad.
* **Caso recursivo:** Versión reducida del problema que se reproduje hasta que se llega al caso base.
* **Consejos generales:**
* Usar **camelCase** para declarar las **variables** (primera letra en minúsculas) y **funciones** (primera letra en mayúscula).
* Las **constantes** se declaran en **mayúsculas**.
* No usar en las variables **palabras reservadas**.
* Usar el **mismo idioma** para todas las variables y funciones, solamente con caracteres del inglés.
* Añadir **comentarios** al código con // (1 línea) o /\* … \*/ (varias líneas).
* Al emplear programación modular, **situar las funciones auxiliares encima del código principal**.

**SEGUNDO TRIMESTRE**

* **Arrays unidimensionales (vectores):**
* **Características:** Datos homogéneos y tamaño fijo.
* **Declaración:** tipoDato[ ] nombreVector;
* **Dar tamaño al vector (obligatorio):** int[ ] numeros = new int [10];
* **Inicializar vector:** int[ ] números = {1, 2, 3};
* **Acceder a una posición del vector:** nombreVector[índice];
* **Rellenar vector con InputBox:**

for (int i = 0; i < vector.Length; i++)

{

vector[i] = int.Parse(Interaction.InputBox(“Introduzca elemento: “ + i));

}

* **Función con vector por parámetro:** void LeerVector(int[ ] vector)
* **OJO:** Al pasar un vector por parámetro siempre **se pasa por referencia** (ref), no por valor, a diferencia de las variables.
* **OJO: En los arrays, la posición inicial siempre es 0.** No confundir el índice (esto es, la posición, que va de 0 a número de elementos – 1) con el número de elementos (cantidad de datos en el array). Por ejemplo, si hay **4 datos**, **los índices van de 0 a 3**.
* **Arrays bidimensionales (matrices):**
* **Declaración:** tipoDato[,] nombreMatriz = new tipoDato[filas, columnas];
* **Acceder a una posición de la matriz:** matriz[2, 3] = 5;
* **Recorrer una matriz, leer sus valores y mostrar en texto:**

const int FILAS = 4;

const int COLUMNAS = 4;

int[,] matriz = new int[FILAS, COLUMNAS];

string texto;

for (int i = 0; i < FILAS; i++)

{

for (int j = 0; j < COLUMNAS; j++)

{

texto += matriz[i, j] + “, ”;

}

texto += “\n”;

}

* **Métodos para manipular arrays:**
* **array.Length():** Devuelve el número de elementos del array.
* **matriz.GetLength():** Devuelve el número de filas o columnas de una matriz:
* matriz.GetLength(0): Filas.
* matriz.GetLength(1): Columnas.
* **string.Compare(cad1, cad2):** Compara dos cadenas de caracteres. Devuelve:
* 0: si las cadenas son iguales.
* Menor que 0: si la cad1 es alfabéticamente menor.
* Mayor que 0: si la cad1 es alfabéticamente mayor.
* **Listas:** Permite trabajar grupos de datos, pero ofrece más versatilidad que un array, pues permite modificar el tamaño del grupo de forma dinámica, con la clase **List**. Para usarla:
* **Añadir la librería de la clase:** using System.Collections.Generic;
* **Sintaxis:** List<tipo> lista = new List<tipo>();
* **Añadir valores:** Mediante dos métodos:
* **Añadir al final:** lista**.**Add(valor)
* **Intercalar un valor:** lista.Insert(posición, valor)
* **Tamaño de la lista:** lista.Count
* **Acceder a elementos individuales de la lista:** lista[posición]
* **Bucle foreach:** Funciona como un bucle for, pero no precisa de los índices, pues una variable va tomando el valor de cada elemento de la lista:
* **Sintaxis:** foreach(tipo variable in List)
* **Ejemplo:**

foreach(int numero in lista)

{

suma += numero;

}

* **Borrar elementos:** Mediante 3 métodos:
* **Eliminar un valor:** lista.Remove(valor)
* **Eliminar el valor en una posición:** lista.RemoveAt(posición)
* **Eliminar todos los valores:** lista.Clear()
* **Búsqueda en la lista:** Mediante 3 métodos:
* **Devolver la primera posición de un valor:** lista.IndexOf(valor)
* **Devolver la última posición de un valor:** lista.LastIndexOf(valor)
* **Devolver si un valor está en la lista (true/false):** lista.Contains(valor)
* **Ordenar la lista:** lista.Sort()

**OJO:** Para usar índices dentro de un bucle foreach, hay que emplear otra **variable que controle las iteraciones del bucle**. Por ejemplo:

int posicion = 0;

foreach (int numero in lista)

{

if (posicion < (lista.Count – 1))

{

texto += numero + “, “;

posicion++;

}

else

{

texto += numero + “.”;

}

}

**OJO:** Al emplear foreach no se puede cambiar el tamaño de la lista en ejecución porque da error. En su lugar, hay que **emplear otro bucle**. Por ejemplo:

int contador = 0;

while (contador < lista.Count)

{

if (lista[contador] % 2 == 0)

{

lista.RemoveAt(contador);

}

else

{

contador++;

}

}

**OJO:** lista.Remove(valor) solamente elimina el valor en la **primera posición en la que se encuentre**, no de todas las posiciones.

**OJO:** Al trabajar con varias listas en la que una de ellas se vaya a copiar o mover elementos, conviene **inicializarlas** (lista.Clear()) para que los nuevos elementos que se añadan no incluyan los que ya tenía.

* **DialogResult:** Esta función permite obtener input del usuario por medio de botones en las cajas de diálogo de Windows Forms, y almacenar su resultado. Elementos:
* **Crear la variable de tipo DialogResult:** DialogResult respuesta;
* **Darle valor a la variable:**

respuesta = MessageBox.Show(“¿Desea continuar?”, “Continuar”, MessageBoxButtons.YesNo);

* **Continuar la ejecución según el valor de la variable:**

if (respuesta == DialogResult.Yes)

{

MessageBox.Show(“Continuamos.”);

}

else

{

MessageBox.Show(“El programa se detiene.”);

}

* **Botones de respuesta:** Se introducen mediante un MessageBox.Show(), con la instrucción MessageBoxButtons.tipo\_de\_botón. Tipos de botones comunes: YesNo, OK, YesNoCancel, ContinueCancel, etc.

**OJO:** Siempre que se vaya a usar un MessageBoxButtons dentro de un MessageBox hay que darle todos los parámetros de Show(string text, string caption, MessageBoxButtons buttons), no se puede saltar el caption (el título del Box).

* **Random:** Clase que permite generar números pseudoaleatorios con el método Next().
* **Sintaxis:**

Random random = new Random();

int numero = random.Next();

* **Sobrecarga del método Next():** Permite generar números random según lo que se le indique en los parámetros:
* **Next():** Número entre -2,147 millones y 2,147 millones.
* **Next(int):** Número menor que el valor especificado.
* **Next(int, int):** Número entre el menor (incluido) y el mayor (excluido) de los especificados
* **NextDouble():** Decimal entre 0.0 y menor que 1.0.
* **Regular Expressions:** Permiten comprobar el contenido de una cadena de caracteres. Uso:
* **Añadir biblioteca:** using System.Text.RegularExpressions;
* **Uso del método IsMatch():** Regex.IsMatch(string, “<expresión regular>”) -> Devuelve un valor booleano.
* **Por ejemplo:**
* **Solo letras:** Regex.IsMatch(palabra, "^[a-zA-Z]+$")
* **Letras y números:** Regex.IsMatch(palabra, "^[a-zA-Z0-9]+$")
* **Letras y espacios:** Regex.IsMatch(palabra, "^[a-zA-Z]\*$")
* **Programación Orientada a Objetos:**
* **Clase:** Plantilla o estructura preliminar de un objeto, que define los **atributos** (miembros) para almacenar datos y las **operaciones** (métodos) para manipular esos datos. Además, contiene **restricciones** para permitir o denegar el acceso a los atributos y operaciones. Elementos de las clases:
* **Abstracción:** Permite definir los atributos y operaciones de forma básica y fundamental.
* **Encapsulación:** Empaquetar los atributos y funciones en una caja negra inaccesible, permitiendo el acceso solo a aquellos que necesiten serlo.
* **Objeto:** Es una instancia de la clase, creada a partir de la plantilla de esa clase. En cuanto a software, es una unidad que contiene los miembros y métodos relacionados en dicha clase. Elementos de los objetos:
* **Identidad:** Permiten distinguirlos de otros objetos de la misma clase.
* **Comportamiento:** Funcionan y tienen utilidad similar a los demás objetos de su clase, y se desconoce su funcionamiento interno.
* **Trabajo con clases:**
* **Crear una nueva clase:** Después del proceso de abstracción para determinar los miembros y métodos necesarios para crear la clase, se deben seguir los pasos siguientes:
* Abrir o crear proyecto en Visual Studio.
* Menú Proyecto > Agregar clase
* Dar nombre a la clase en el menú Agregar nuevo elemento.
* Se creará un nuevo fichero con la siguiente estructura:

class Persona

{ }

* **Agregar miembros:** Para ello, hay que indicar el nivel de acceso a estos datos (public / private), porque no deben ser accesibles desde cualquier parte. Estos miembros pueden ser variables (campos) o constantes, y normalmente son privados, de modo que solo se puede acceder a ellos a través de métodos específicos. Ejemplo de nuevos miembros:

class Persona

{

private string nombre;

private int edad;

}

* **Agregar métodos:** Son subprogramas que permiten trabajar con los datos de la clase, por lo que **no necesitan ser pasados por parámetro** a estos subprogramas. También es posible especificar el nivel de acceso (public / private). Por ejemplo:

class Persona

{

private string nombre;

private int edad;

public void cumpleanos()

{

edad += 1;

}

}

* **Agregar propiedades:** Las propiedades son métodos especiales que permiten exponer y/o modificar los valores privados de una clase, de manera que las propiedades sí son públicas. Dos tipos:
* **Método get:** Recupera el valor de un miembro de la clase, pero no permite variar su valor.
* **Método set:** Permite asignar un valor a un miembro.

Por ejemplo:

public string Nombre

{

get { return nombre; }

set { nombre = value; }

}

Posteriormente, estas propiedades son accesibles desde los objetos creados en esta clase. Por ejemplo:

// Creación del nuevo objeto nuevaPersona

Persona nuevaPersona = new Persona();

// Llamada al método set del nuevo objeto

nuevaPersona.Nombre = Interaction.InputBox(“Introduzca nombre”);

// Llamada al método get del nuevo objeto

string nombre = nuevaPersona.Nombre;

**→ OJO:** Las propiedades no llevan ( ) al final en su declaración.

* **Crear objetos:** Para ejecutar los métodos y propiedades de una clase, es preciso crear una instancia u objeto. Sintaxis de creación del objeto:

Clase nombreObjeto = new Clase();

* **Construir objetos:** Permiten crear objetos con valores iniciales mediante métodos llamados **constructores**, que tienen las siguientes características:
* Tienen el **mismo nombre** que la clase.
* Se ejecutan **una única vez**, al crear el objeto.
* Se ejecuta **antes** que cualquier otro código de la clase.
* Pueden existir varios constructores, en función de los parámetros que se le pasen. Por ejemplo:

public Persona()

{

nombre = “”;

edad = 0;

}

public Persona(string pNombre, int pEdad)

{

nombre = pNombre;

edad = pEdad;

}

* **Trabajar con una lista de objetos:**
* **Crear la lista:** Empleando la misma sintaxis:

List<Clase> lista = new List<Clase>;

* **Añadir objetos a la lista:**

Clase objeto = new Clase();

lista.Add(objeto);

* **Iterar lista de objetos:**

for (i = 0; i < nombreLista.Count; i++)

{

Clase objeto = lista[i];

}

foreach (Clase objeto in lista)

{

texto = objeto.MostrarDatos();

}

→ **OJO:** Hay diferencias entre **crear un objeto** (Clase objeto = new Clase()) y **hacer referencia a él** (Clase objeto = lista[i]). En el primer caso, se crea el objeto con el constructor, y en el segundo, **se apunta al objeto, pero no se crea**, no se guarda en una variable, sino que los cambios que se haga a la referencia se aplican al objeto apuntado.

→ **OJO:** Con InputBox no puedo leer char, por lo que hay que leer string y luego pasarlo a char. Igualmente, no puedo leer valores booleanos, que se tienen que leer con DialogResult y MessagBoxButtons.

→ **OJO: Las listas se crean como clases y se tratan como objetos**, con sus miembros, propiedades, constructor y métodos propios.

→ **OJO:** Se puede importar una clase de un proyecto a otro, por lo que es posible reutilizarlas: Proyecto > Agregar elemento existente > Seleccionar la clase correspondiente.

* **Trabajar con varios formularios:**
* **Añadir nuevo formulario:** Proyecto > Añadir formulario > Darle nombre (por ejemplo, “subformulario”).
* **Instanciar el nuevo formulario en el formulario principal:**

Formulario subformulario = **new** Formulario();

* **Mostrar el nuevo formulario:**

subformulario.**ShowDialog()**;

* **Persistencia de la información:** Para poder trabajar con información persistente entre formularios hay que:
* **Declarar las clases como públicas:** Indicar con la palabra reservada “public” que la clase es pública al momento de definirla:

**public** class Lista {}

* **Instanciar el objeto en el formulario principal:** Si no, al salir del subformulario se borraría la información. Estos se crean con new:

Lista lista = **new** Lista();

* **En los subformularios, se crean objetos de esa misma clase (con acceso público), pero sin new (pues son punteros):**

**public** Lista lista;

* **Luego, en el botón que llama al formulario, hay que instanciarlo, pasarle el objeto al subformulario y mostrarlo:**

private void b\_Click(…)

{

Formulario subformulario = new Formulario();

**subformulario.lista = lista;**

subformulario.ShowDialog();

}

* **Atributo “static”:** Se emplea para definir una clases que no tiene constructores, sino que solamente tiene miembros y métodos que también son *static*. De este modo, no se pueden crear instancias de estas clases (con el new), sino que se llaman por su nombre y el método a emplear, por ejemplo, Auxiliar.Método(). Es muy útil para crear funciones de apoyo o *helpers*.

**TERCER TRIMESTRE**

* **Herencia:** Capacidad de crear clases que heredan ciertos aspectos o funcionalidades de otras clases primarias.
* **Sintaxis:** La **clase derivada** (subclase) hereda los miembros, propiedades y métodos de la **clase base** (superclase), que pasan a formar parte de ella:

public class ClaseDerivada **:** ClaseBase

{

}

* **Ejemplo de trabajo con herencia:** 
  + Creación de una **clase base denominada Figura**, que posee los datos comunes a las distintas figuras geométricas, de la que todas pueden tomar datos:

public class Figura

{

private int posicionX;

private int posicionY;

private string colorFigura;

public string ColorFigura

{

get { return colorFigura; }

set { colorFigura = value; }

}

public Figura (int x, int y, string color)

{

posicionX = x;

posicionY = y;

colorFigura = color;

}

**}**

* + Creación de una **clase derivada denominada Círculo**, que hereda la funcionalidad de la clase Figura, pero añade sus propios miembros, propiedades o funciones:

public class Circulo **:** Figura

{

private int radioCirculo;

private int RadioCirculo

{

get { return radio;}

set { radio = value; }

}

public Circulo(int x, int y, string color, int radio) : base(x, y, color)

{

radioCirculo = radio;

}

}

* + **Accesibilidad:** 
    - El constructor de la clase derivada llama al constructor de la clase base (a través de la **palabra reservada** **base**). En este caso, Círculo llama a Figura a través de su constructor para inicializar también sus miembros.
    - Todo lo que es público en la clase base, también lo es en la clase derivada. Es decir, se puede acceder a la propiedad ColorFigura a través de una instancia del objeto Círculo, por ejemplo:

Circulo circulo = new Circulo(10, 10, “Azul”, 5);

circulo.ColorFigura = “Rojo;

* + - Sin embargo, no se puede acceder a los miembros posicionX y posicionY porque son miembros privados de la clase base.
    - El modificador **protected**, en cambio, sí permite a las clases heredadas acceder a los miembros de la clase base, pero no son accesibles desde fuera.
* **Redefinición de métodos:** La clase derivada puede llamar métodos de la base. Pero, además, puede redefinirlos, sustituyendo su valor de salida por el suyo propio, usando la **palabra reservada new**. Sin embargo, esta no es la forma adecuada de trabajar en Programación Orientada a Objetos (si se crea una instancia de Figura que apunte a un objeto Círculo, al llamar al método, lo llamará como si fuera Figura en lugar de como Círculo).
* **Sobreescritura de métodos:** Es la forma adecuada de trabajar con funciones genéricas en la clase base y funciones más específicas en la clase derivada. Para ello, se crean métodos virtuales (con la **palabra reservada virtual**) en la clase base y métodos de sobreescritura (con la **palabra reservada override**) en la clase derivada.

public class Figura

{

public virtual string QuienSoy()

{ return “Soy una figura.”; }

}

public class Circulo **:** Figura

{

public override string QuienSoy()

{ return “Soy un círculo.”; }

}

**→ OJO:** De hecho, es posible llamar al método virtual de la clase base desde el método de sobreescritura, y cada uno cumplirá su función:

public class Circulo **:** Figura

{

public override string QuienSoy()

{

string texto;

texto = base.QuienSoy();

texto += “ En concreto, un círculo.”;

return texto;

}

}

**→ OJO:** Es posible sobrescribir el método ToString(), porque es un método heredado de System.Object. Se puede sobreescribir en Figura para que salga en un formato en concreto y, posteriormente, en Círculo para ajustarlo aún más.

* **Clases y métodos abstractos:**
* **Clase abstracta:** Clase de la que **no van a existir objetos**, pero que sí implementa elementos útiles para las clases que deriven de ella. Es el caso de Figura, de la que no se van a instanciar objetos. Se definen con la **palabra reservada abstract**, y puede tener métodos concretos y abstractos:

public abstract class Figura

{}

**→ OJO:** Es muy importante saber que no es posible crear objetos de una clase abstracta, pero **sí se pueden crear punteros**. Es decir,

~~Figura figura = new Figura()~~ → Es incorrecto y da error.

Circulo circulo = new Circulo();

Figura figura = circulo; **→** Es correcto, el puntero apunta al objeto.

**→ OJO:** Por eso, aunque no se creen objetos, l**as clases abstractas necesitan constructor** para crear punteros.

**→ OJO:** A la hora de crear nuevos objetos, se puede poner tanto la clase base como la clase derivada como tipo del objeto a crear. Lo importante es que el new se aplique sobre la clase derivada y no sobre la clase base:

Por ejemplo:

Figura circulo = new Circulo();

Es lo mismo que:

Circulo circulo = new Circulo();

* **Método abstracto:** No tienen cuerpo, porque **van a ser totalmente reescritos** por las clases derivadas. Por ejemplo, un método que implemente el área de una figura:

public abstract class Figura

{

public abstract double Area();

}

public class Circulo **:** Figura

{

public override double Area()

{

return Math.PI \* radio \* radio;

}

}

**→ OJO:** Estos métodos deben obligatoriamente sobrescribirse en las clases derivadas, o darán error.

* **Polimorfismo:** Refiere a que un objeto de la clase base puede adoptar varias formas en las clases derivadas, lo que permite apuntar con un mismo puntero de la clase base a diferentes objetos de clases derivadas. Ejemplo:

Circulo circulo = new Circulo();

Cuadrado cuadrado = new Cuadrado();

Figura figura = circulo; // El puntero figura apunta al objeto circulo.

figura = cuadrado; // Ahora el puntero figura apunta al objeto cuadrado.

**→ OJO:** Esto implica que en cada momento asume las propiedades y métodos de la clase derivada a la que apunta, lo que implica que, en su caso, ejecutará los métodos sobrescritos (override), no los métodos de la clase base (virtual).

* **Creación de una lista de figuras usando el polimorfismo:** La clase para la lista debe permitir incluir elementos de la clase base para que puedan ser accesibles y trabajar con ellos:

public class ListaFiguras

{

List <Figura> lista;

public ListaFiguras()

{

lista = new List<Figura>;

}

void obtenerDatos(out int posX, out int posY, out string color)

{

posX = int.Parse(InputBox(“Introducir posición X”);

posY = int.Parse(InputBox(“Introducir posición Y”);

color = InputBox(“Introducir color”);

}

void Anyadir(Figura figura)

{

lista.Add(figura);

}

}

* **Recorrer la lista de objetos:** Este es el verdadero potencial del polimorfismo, poder recorrer una lista de objetos diferentes (heredados) usando un puntero de la clase base. Esto es así porque el puntero toma los valores del objeto al que apunta y ejecuta sus métodos, no los de la clase base:

private void Button()

{

int cont = 1;

foreach (Figura figura in listaFiguras)

{

string texto;

texto = “Figura num. ” + cont + “\n”;

texto += “Área: ” + figura.Area();

MessageBox.Show(texto);

cont++;

}

}

* **Obtener el tipo de un objeto:** El método **GetType()**, heredado de System.Object, permite identificar el tipo de un objeto mediante una comparación (**typeof()**) y así poder clasificarlos a pesar de estar en una lista con objetos de otras clases. Ejemplo:

public int ContarCirculos()

{

int cont = 0;

foreach (Figura figura in listaFiguras)

{

if (figura.GetType() == typeof(Circulo))

{

cont++;

}

}

return cont;

}

* **Acceder desde un puntero de clase base a un método específico de la clase derivada a la que apunta:** En caso de que el puntero apunte hacia un método de la clase derivada que no esté en la clase base, se debe hacer un **casting** hacia el objeto de la clase derivada. Ejemplo:

La clase base no tiene un método para calcular el perímetro, pero la clase derivada Cuadrado sí:

public double Perimetro()

{

return 4 \* lado;

{

Si en la clase Lista se accede a ese método en concreto a través de un puntero de la clase Figura, es preciso castear el objeto:

public SumaPerimetros()

{

double suma = 0;

foreach (Figura figura in listaFiguras)

{

suma += (Cuadrado)figura.Perimetro();

}

return suma;

}

* **Acceso a Datos en .Net Framework: ADO.NET**
* **Componentes de ADO.NET:**
  + **Proveedor de datos .NET:** Acceso a los datos. Permite vincular una aplicación con una base de datos para conectarse, ejecutar comandos y recuperar resultados. Según el tipo de base de datos, se emplea un proveedor diferente (para SQL Server se emplea System.Data.SqlCliente, etc.).
  + **Conjunto de datos:** Manipulación de los datos. Se trabaja a través del objeto DataSet, que permite enlazar los datos con controles de formulario. El objeto DataAdapter ofrece el método Fill() al que se le pasan por parámetro el DataSet a rellenar con datos y un string con el nombre de la tabla dentro del DataSet, producto de la consulta.
* **Conectar una base de datos SQL Server:** 
  + Crear **nuevo proyecto** en Visual Studio y **copiar la base de datos en la carpeta** del proyecto.
  + Añadir la **biblioteca System.Data.SqlClient**.
  + **Acceder al evento Load()** del formulario (doble click en el formulario).
  + Crear el **objeto de conexión** en el evento y darle la **ruta de la base de datos**:

Ver → Otras Ventanas → Orígenes de datos → Agregar nuevo origen de datos → Base de datos → Conjunto de datos → Agregar conexión → Origen de los datos: Archivo de SQL Server / Examinar: Elegir la base de datos añadida a la carpeta.

* + **Copiar la cadena de conexión** con Ctrl+C y cancelar el asistente de conexión.
  + **Pegar la cadena en el evento Load()**. Ojo: Hay que cambiar las \ por \\.
  + **Abrir y cerrar la conexión en el evento Load()**.
  + **Código final**:

using System.Data.SqlClient;

private void Form1\_Load(…)

{

string cadenaConexion = “Data Source=(LocalDB)…”;

SqlConnection con = new SqlConnection(cadenaConexion);

con.Open();

con.Close();

}

* **Recoger registros de la base de datos:** 
  + **DataSet:** Los datos recogidos de la base de datos se almacenan en el DataSet, Sin embargo, el objeto DataSet y el objeto SqlConnection no pueden interactuar entre sí, sino que precisan de un adaptador entre ellos. Creando un DataSet:
    - **Declaración fuera del evento Load():**

DataSet dsProfesores;

* + - **Creación del objeto dentro del evento Load():**

dsProfesores = new DataSet();

* + **DataAdapter:** Permite adaptar los datos de la base de datos a un DataSet. Creando un DataAdapter:
    - **Declaración fuera del evento Load():**

SqlDataAdapter da;

* + - **Creación del objeto dentro del evento Load() seleccionando todos los datos de una tabla con una sentencia SQL:**

string cadenaSQL = “SELECT \* FROM Profesores”;

da = new SqlDataAdapter(cadenaSQL, con);

* + **Rellenar el DataSet con el DataAdapter:**

da.Fill(dsProfesores, “Profesores”);

* **Mostrar datos del DataSet en un TextBox:** 
  + **Crear un objeto para recoger registros del DataSet:** El objeto DataRow permite almacenar filas de un DataSet:

DataRow fila;

* + **Seleccionar un registro por posición:**

fila = dsProfesores.Tables[“Profesores”].Rows[posición];

* + **Asignar cada valor de la fila a un TextBox mediante su posición a su nombre:**

txtDNI.Text = fila[0].ToString();

txtDNI.Text = fila[“DNI”].ToString();

* **Navegar por la base de datos:** Permite mostrar el primer registro, el último, el siguiente o el anterior.
  + **Primer registro:**

private int posicion = 0;

mostrarRegistro(posicion);

* + **Siguiente registro:**

posicion++;

mostrarRegistro(posicion);

* + **Anterior registro:** Ojo, se puede salir del límite.

posicion--;

mostrarRegistro(posicion);

* + **Último registro:**

private int maxRegistro = dsProfesores.Tables[“Profesores”].Rows.Count;

private int posicion = maxRegistro – 1;

mostrarRegistro(posicion);

* **Añadir nuevo registro a la base de datos:** 
  + **Botón Añadir:** Limpia todos los TextBox para poder introducir datos.

txtDNI.Clear();

* + **Botón Guardar:** 
    - **Guardar en el DataSet:**
      * **Crear una nueva fila en el DataSet:**

DataRow fila = dsProfesores.Tables[“Profesores”].NewRow();

* + - * **Almacenar en la fila los datos que constan en los TextBox:** Se hace bien con un índice o bien con el nombre de la columna:

fila[0] = txtDNI.Text;

fila[“DNI”] = txtDNI.Text;

* + - * **Añadir la fila al DataSet:**

dsProfesores.Tables[“Profesores”].Rows.Add(fila);

* + - **Guardar en la base de datos:** 
      * **Reabrir la base de datos que fue cerrada en el evento Load():**

SqlCommandBuilder cd = new SqlCommandBuilder(da);

* + - * **Actualizar la base de datos usando el DataAdapter:**

da.Update(dsProfesores, “Profesores”);

* **Actualizar registros:** Similar a añadir registros, pero en lugar de con una nueva fila, se modifican los datos de la fila actual.

DataRow fila = dsProfesores.Tables[“Profesores”].Row[posicion];

fila[0] = txtDNI.Text;

SqlCommandBuilder cd = new SqlCommandBuilder(da);

da.Update(dsProfesores, “Profesores”);

* **Eliminar registros:** Borrar la fila actual y actualizar la tabla:

dsProfesores.Tables[“Profesores”].Row[posicion].Delete();

SqlCommandBuilder cd = new SqlCommandBuilder(da);

da.Update(dsProfesores, “Profesores”);

* **Acceso a datos usando POO:** En POO las buenas prácticas requieren encapsular todo el código relacionado con la base de datos y tratarla como un objeto manipulable.
  + **Crear una clase para la información necesaria:** Se trata de una clase con los datos básicos del programa. Debe tener sus propios miembros, propiedades, constructor y métodos para trabajar los datos obtenidos.
  + **Crear una clase que se encargue de toda la base de datos:** Esta clase gestionará la **conexión** con la base de datos, la creación del **DataSet y el DataAdapter**, así como las operaciones **CRUD** (create, read, update y delete). La clase (de apoyo) se llamará **SqlDBHelper**.
  + **Crear un objeto formulario que recibirá solamente aquello que necesita de la clase de la base de datos:** Se simplifica el trabajo con el formulario.
* **Abrir y ver una base de datos desde Visual Studio:** Se puede abrir una base de datos desde Visual Studio sin necesidad de una explorador de bases de datos. Para ello:
  + **Crear carpeta para la nueva base de datos en proyecto vacío:** Click derecho en el Explorador de soluciones → Añadir Carpeta → Darle nombre (“AppData”).
  + **Añadir la base de datos al proyecto:** Click derecho en la carpeta → Añadir elemento existente → Seleccionar la base de datos (.mdf).
  + **Abrir el explorador de servidores:** Ver → Explorador de servidores.
  + **Comprobar que ya se muestran las tablas, sus datos y su diseño:** Incluso se pueden incluir nuevos campos, funciona como SQL Server.
* **Crear una nueva base de datos desde Visual Studio:** 
  + **Desde la carpeta “AppData” crear nuevo elemento:** Seleccionar “base de datos basada en servicio” (.mdf).
  + **Comprobar que se muestra en el explorador de servidores, aunque no tiene tablas.**
  + **Agregar nueva tabla:** Click derecho en la carpeta “Tablas” → Agregar nueva tabla.
  + **Cambiar nombre de la tabla:** En la sentencia de SQL que se muestra, cambiar el nombre de la tabla.
  + **Reiniciar la clave principal:** Par que no dé problemas, eliminar y volver a poner la clave principal.
  + **Clave principal hacer identidad autoincremental:** En las propiedades de la derecha, cambiar la propiedad **identity** a true.
  + **Introducir datos:** Click derecho sobre la tabla → Mostrar los datos de la tabla. En la pantalla central se pueden modificar los datos e introducir nuevos.

1. Si MessageBox.Show() empieza con texto, las variables que se añadan después ya irán como string, no hace falta ponerles ToString(). [↑](#footnote-ref-1)