Herramientas de Programación 1

Tema Nº8:

Programación Orientada a Objetos

Indicador de logro Nº8:Desarrolla aplicaciones modulares implementando aplicaciones con la Programación Orientada a Objetos en C#.

**TEMA 01 Teoría de los**

Imagen que contiene Icono

Descripción generada automáticamente

**TEMA Nº8:**

PROGRAMACION ORIENTADA A OBJETOS: CLASES Y OBJETOS

Las Clases

El componente principal de la programación orientada a objetos es la clase. Nosotros podemos pensar en la clase como si fuera un plano o modelo por medio del cual podemos crear objetos. Por medio del modelo podemos construir, por ejemplo, un automóvil y con este objeto ya podemos llevar a cabo nuestras actividades.

**Ejemplos:**

class Rectangulo

{

public double largo;

public double ancho;

private double area;

public void CalcularArea()

{

area = largo \* ancho;

}

public double ObtenerArea()

{

return area;

}

public double ObtenerPerimetro()

{

double perimetro = (largo + ancho) \* 2;

return perimetro;

}

Los datos son la información con la que trabajará la clase. Declarar un dato es muy similar a declarar una variable, necesitamos indicar el acceso ya que tenemos básicamente tres tipos de acceso: public, private y protected. Cuando nosotros tenemos un dato con acceso public cualquier elemento del exterior puede acceder al dato, leerlo y modificarlo. Cuando tenemos el acceso private solamente los métodos definidos dentro de la clase podrán leerlo o modificarlo. Un punto muy importante con relación a los datos que no debemos olvidar es que los datos definidos en la clase son conocidos por todos los métodos de la misma clase. Es decir, actúan como si fueran globales para la clase. Cualquier método puede acceder a ellos directamente sin necesidad de que los pasemos como parámetro.

Los métodos son las funciones que llevan a cabo el proceso o la lógica de la clase, y crear un método dentro de la clase es muy parecido a la forma que hemos utilizado anteriormente. Los métodos también tendrán un tipo de acceso, al igual que los datos. Trabajarán sobre los datos de la clase. Solamente los métodos que necesiten ser invocados desde el exterior deben tener acceso public. Si el método sólo se invoca desde el

mismo interior de la clase su acceso debe ser private. Esto lo hacemos con fines de seguridad y para mantener el encapsulamiento correctamente.

Los Objetos

Para instanciar un objeto de la clase cubo debemos utilizar el código que se muestra a continuación:

**Ejemplos:**

Rectangulo miRectangulo = new Rectangulo();

Quien realiza realmente la instanciación es new y esa instancia queda referenciada como miRectangulo. A partir de miRectangulo podremos empezar a trabajar con él.

Asignarles valores a datos public:

**Ejemplos:**

miRectangulo.largo = 20.5;

Cómo invocar métodos public de un objeto:

**Ejemplos:**

miRectangulo.CalcularArea();

double area = miRectangulo.ObtenerArea();

Encapsulamiento: Protección De Datos Y Creación De Propiedades

Con la clase Rectangulo hemos visto cómo crear una clase sencilla que ya tiene funcionalidad.

Sin embargo, presenta un problema: dos de sus datos son públicos, lo que nos puede llevar a corrupción de información. Para proteger los datos tenemos que hacerlos privados y proveer una función de interfaz a aquellos a los que se necesita acceder por el exterior. A esta función de interfaz se lo denominamos como PROPIEDAD.

**Ejemplos:**

class Rectangulo

{

private double \_largo;

private double \_ancho;

private double \_area;

//propiedad Largo

public double Largo

{

get

{

return \_largo;

}

set

{

if (value <= 0)

\_largo = 1;

else

\_largo = value;

}

}

//propiedad Ancho

public double Ancho

{

get

{

return \_ancho;

}

set

{

if (value <= 0)

\_ancho = 1;

else

\_ancho = value;

}

}

//propiedad Ancho

public double Area

{

get

{

return \_largo \* \_ancho;

}

set

{

if (value <= 0)

\_area = 1;

else

\_area = value;

}

}

}

Acceder y asignar valores a las propiedades:

**Ejemplos:**

Rectangulo miRectangulo = new Rectangulo();

miRectangulo.Largo = 15;

miRectangulo.Ancho = 0;

double area = miRectangulo.Area; //area = 15

**EL CONSTRUCTOR EN LAS CLASES**

El constructor es un método especial que podemos utilizar con las clases. Éste generalmente es usado para inicializar los valores de los datos con los que trabajará el objeto.

El constructor es un método especial y tiene ciertas características que lo distinguen de los demás métodos.

Su primera característica es que tiene el mismo nombre de la clase y su segunda característica más importante es que no tiene tipo, es decir, que no solamente no regresa nada, sino que no tiene tipo alguno.

El constructor es invocado en forma automática cuando el objeto es instanciado, ya que esto nos da la oportunidad de llevar a cabo cosas en el instante que se instancia el objeto, como por ejemplo, hacer inicializaciones. El constructor puede tener en su interior cualquier código válido de C# y también puede tener parámetros o no. Si utilizamos los parámetros tendremos que pasar los valores necesarios en el momento en el que instanciamos el objeto.

Veamos un constructores con y sin parámetros:

**Ejemplos:**

class Rectangulo

{

private double \_largo;

private double \_ancho;

private double \_area;

public Rectangulo()

{

\_largo = 0;

\_ancho = 0;

\_area = \_largo \* \_ancho;

}

public Rectangulo(double pLargo, double pAncho)

{

\_largo = pLargo;

\_ancho = pAncho;

\_area = \_largo \* \_ancho;

}

Invocación de los constructores:

Rectangulo miRectanguloA = new Rectangulo();

Rectangulo miRectanguloB = new Rectangulo(12, 20);

**Ejemplos:**

**RECURSOS**

1. Hardware

* Una computadora con 4 GB e RAM y procesador de 2.0 GHz.

1. Software

* Visual Studio Professional 2015 / 2017

**PROCEDIMIENTO**

**Laboratorio 8:**

8.1 Clases y Objetos.

**Objetivos**

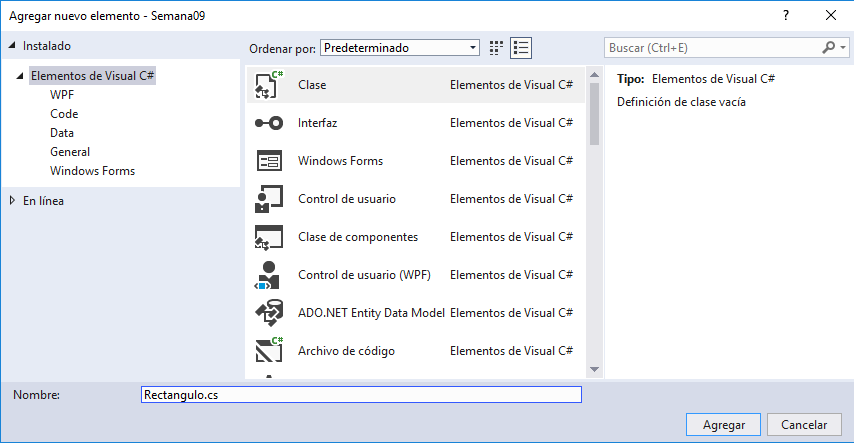
* Entender y aplicar los conceptos de la POO en el desarrollo de aplicaciones.

**Introducción**

Durante esta actividad, vamos aprender a declarar y definir Clases, además de utilizarlos mediante la creación de Objetos.

Paso 1. Ejemplo 1.

1. Creamos el proyecto Semana09.
2. Agregamos la clase Rectangulo mediante la opción: Agregar > Clase…



1. Modificamos la definición de esta clase de acuerdo a:

public class Rectangulo

{

private double \_largo;

private double \_ancho;

private double \_area;

private double \_perimetro;

public Rectangulo()

{

\_largo = 0;

\_ancho = 0;

\_area = 0;

\_perimetro = 0;

}

public Rectangulo(double pLargo, double pAncho)

{

\_largo = pLargo;

\_ancho = pAncho;

\_area = 0;

\_perimetro = 0;

}

public double Largo

{

get

{

return \_largo;

}

set

{

if (value <= 0)

\_largo = 1;

else

\_largo = value;

}

}

public double Ancho

{

get

{

return \_ancho;

}

set

{

if (value <= 0)

\_ancho = 1;

else

\_ancho = value;

}

}

public double Area

{

get

{

return \_area;

}

}

public double Perimtero

{

get

{

return \_perimetro;

}

}

public void CalcularArea()

{

\_area = \_largo \* \_ancho;

}

public void CalcularPerimetro()

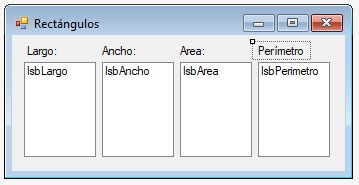
{

\_perimetro = 2 \* (\_largo + \_ancho);

}

}

1. Al formulario le damos el siguiente diseño:



|  |  |  |
| --- | --- | --- |
| Control | Propiedad | Valor |
| Form1 | (Name) | frmRectangulos |
|  | Text | Rectángulos |
| ListBox1 | (Name) | lsbLargo |
| ListBox2 | (Name) | lsbAncho |
| ListBox3 | (Name) | lsbArea |
| ListBox4 | (Name) | lsbPerimetro |

1. Agregamos un método en la clase del formulario que nos permita mostrar los datos de un Rectángulo:

private void mostrarRectangulo(Rectangulo pRectangulo)

{

lsbLargo.Items.Add(pRectangulo.Largo.ToString());

lsbAncho.Items.Add(pRectangulo.Ancho.ToString());

lsbArea.Items.Add(pRectangulo.Area.ToString());

lsbPerimetro.Items.Add(pRectangulo.Perimtero.ToString());

}

1. Agregamos código al evento Load del formulario:

private void frmRectangulos\_Load(object sender, EventArgs e)

{

Rectangulo rectangulo1 = new Rectangulo();

Rectangulo rectangulo2 = new Rectangulo();

rectangulo2.Largo = 15;

rectangulo2.Ancho = 0;

Rectangulo rectangulo3 = new Rectangulo();

rectangulo3.Largo = 15;

rectangulo3.Ancho = 0;

rectangulo3.CalcularArea();

rectangulo3.CalcularPerimetro();

Rectangulo rectangulo4 = new Rectangulo(12, 20);

Rectangulo rectangulo5 = new Rectangulo(12, 20);

rectangulo5.CalcularArea();

rectangulo5.CalcularPerimetro();

mostrarRectangulo(rectangulo1);

mostrarRectangulo(rectangulo2);

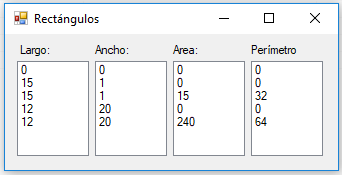
mostrarRectangulo(rectangulo3);

mostrarRectangulo(rectangulo4);

mostrarRectangulo(rectangulo5);

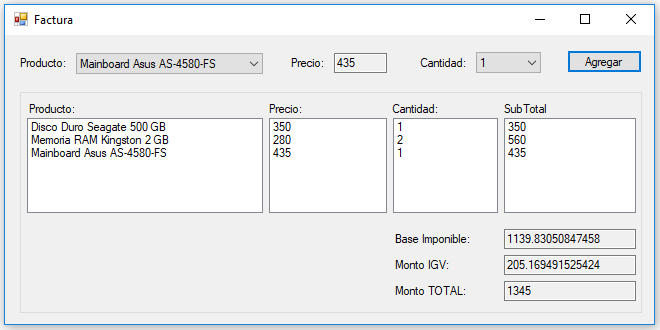
}

1. Ejecutamos la aplicación:



**Paso 2. Ejemplo 2.**

Vamos a desarrollar la funcionalidad de una factura, para ello vamos a crear la clase Producto y la clase Item.



Se tendrá una lista de productos que se crearán al momento de ejecutar la aplicación.

Cada línea del detalle de la factura será un objeto de la clase Item, por lo esta clase contiene una clase Producto y dos propiedades: Cantidad y SubTotal.

1. Agregamos la clase Producto mediante la opción: Agregar > Clase…

public class Producto

{

private int \_id;

private string \_nombre;

private double \_precio;

public int Codigo

{

get

{

return \_id;

}

set

{

\_id = value;

}

}

public string Nombre

{

get

{

return \_nombre;

}

set

{

\_nombre = value;

}

}

public double Precio

{

get

{

return \_precio;

}

set

{

\_precio = value;

}

}

public Producto()

{

\_id = 0;

\_nombre = "";

\_precio = 0;

}

public Producto(int pId, string pNombre, double pPrecio)

{

\_id = pId;

\_nombre = pNombre;

\_precio = pPrecio;

}

}

1. Agregamos la clase Item mediante la opción: Agregar > Clase…

public class Item

{

private Producto \_producto;

private double \_cantidad;

private double \_subTotal;

public Producto Producto

{

get

{

return \_producto;

}

set

{

\_producto = value;

//también funciona con:

//\_producto.Codigo = value.Codigo;

//\_producto.Nombre = value.Nombre;

//\_producto.Precio = value.Precio;

}

}

public double Cantidad

{

get

{

return \_cantidad;

}

set

{

\_cantidad = value;

}

}

public double SubTotal

{

get

{

return \_subTotal;

}

set

{

\_subTotal = value;

}

}

public Item(Producto pProducto, double pCantidad, double pSubTotal)

{

\_producto = pProducto;

\_cantidad = pCantidad;

\_subTotal = pSubTotal;

}

}

1. Vamos a crear el formulario de la factura con estas propiedades:

|  |  |  |
| --- | --- | --- |
| Control | Propiedad | Valor |
| Form1 | (Name) | frmFactura |
|  | Text | Factura |
| ComboBox1 | (Name) | cmbProducto |
|  | DropDownStyle | DropDownList |
| ComboBox2 | (Name) | cmbCantidad |
|  | DropDownStyle | DropDownList |
| TextBox1 | (Name) | txtPrecio |
|  | ReadOnly | True |
| Button1 | (Name) | btnAgregar |
|  | Text | Agregar |
| ListBox1 | (Name) | lsbProducto |
| ListBox2 | (Name) | lsbPrecio |
| ListBox3 | (Name) | lsbCantidad |
| ListBox4 | (Name) | lsbSubtotal |
| TextBox2 | (Name) | txtBaseImponible |
|  | ReadOnly | True |
| TextBox3 | (Name) | txtMontoIgv |
|  | ReadOnly | True |
| TextBox4 | (Name) | txtMontoTotal |
|  | ReadOnly | True |

1. Agregamos en el formulario las listas con la que vamos a trabajar y un atributo que guardara el porcentaje del Igv que aplicaremos a la factura:

public partial class frmFactura : Form

{

private ArrayList listaProductos = new ArrayList();

private ArrayList listaItems = new ArrayList();

private static double porcentajeIgv = 18;

public frmFactura()

{

InitializeComponent();

}

1. Para cargar los ComboBox agregamos el siguiente código en el evento Load del formulario:

private void frmFactura\_Load(object sender, EventArgs e)

{

Producto producto1 = new Producto(1, "Disco Duro Seagate 500 GB", 350);

Producto producto2 = new Producto(2, "Memoria RAM Kingston 2 GB", 280);

Producto producto3 = new Producto(3, "Memoria RAM Kingston 4 GB", 500);

Producto producto4 = new Producto(4, "Procesador Core i7 9naGen", 775);

Producto producto5 = new Producto(5, "Procesador Core i9 9naGen", 1196);

Producto producto6 = new Producto(6, "Mainboard Asus AS-4580-FS", 435);

Producto producto7 = new Producto(7, "Monitor LG 2150 LED", 290);

Producto producto8 = new Producto(8, "Gaming Monitor WQHD Samsung", 715);

listaProductos.Add(producto1);

listaProductos.Add(producto2);

listaProductos.Add(producto3);

listaProductos.Add(producto4);

listaProductos.Add(producto5);

listaProductos.Add(producto6);

listaProductos.Add(producto8);

foreach (Producto producto in listaProductos)

{

cmbProducto.Items.Add(producto.Nombre);

}

cmbProducto.SelectedIndex = 0;

for(int i = 0; i < 5; i++)

{

cmbCantidad.Items.Add((i + 1).ToString());

}

cmbCantidad.SelectedIndex = 0;

}

1. Vamos implementar una función que nos permita devolver el producto (objeto) enviándole como parámetro el nombre del producto:

private Producto obtenerProducto(string pNombreProducto)

{

Producto productoBuscado = new Producto();

foreach (Producto producto in listaProductos)

{

if (pNombreProducto.Contains(producto.Nombre))

{

productoBuscado = producto;

}

}

return productoBuscado;

}

1. Para mostrar el precio del producto seleccionado agregamos el siguiente código en el evento SelectedIndexChanged del ComboBox cmbProducto:

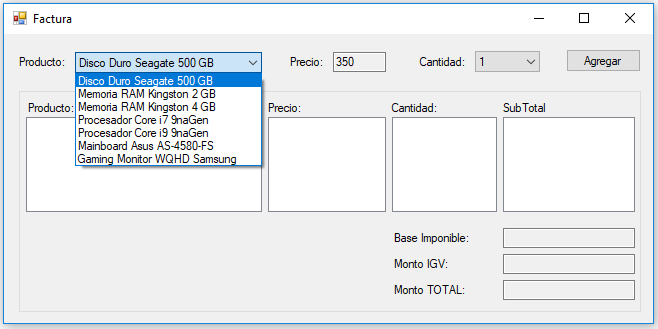
private void cmbProducto\_SelectedIndexChanged(object sender, EventArgs e)

{

txtPrecio.Text = obtenerProducto(cmbProducto.Text).Precio.ToString();

}

1. Ejecutamos la aplicación:



1. Vamos agregar dos métodos, uno que muestre en los ListBox los datos de los ítems que tenemos en la lista de ítems y la otra que calcule y nos muestre los totales de la factura:

private void mostrarItems()

{

lsbProducto.Items.Clear();

lsbPrecio.Items.Clear();

lsbCantidad.Items.Clear();

lsbSubtotal.Items.Clear();

foreach (Item item in listaItems)

{

lsbProducto.Items.Add(item.Producto.Nombre);

lsbPrecio.Items.Add(item.Producto.Precio.ToString());

lsbCantidad.Items.Add(item.Cantidad.ToString());

lsbSubtotal.Items.Add(item.SubTotal.ToString());

}

}

private void mostrarTotales()

{

double baseImponible = 0;

double montoIgv = 0;

double montoTotal = 0;

foreach (Item item in listaItems)

{

montoTotal = montoTotal + item.SubTotal;

}

baseImponible = montoTotal / (1 + porcentajeIgv / 100);

montoIgv = montoTotal - baseImponible;

txtBaseImponible.Text = baseImponible.ToString();

txtMontoIgv.Text = montoIgv.ToString();

txtMontoTotal.Text = montoTotal.ToString();

}

1. Finalmente vamos agregar código al evento Click del botón btnAgregar:

private void btnAgregar\_Click(object sender, EventArgs e)

{

Producto producto = obtenerProducto(cmbProducto.Text);

double cantidad = Convert.ToDouble(cmbCantidad.Text);

double subtotal = producto.Precio \* cantidad;

Item item = new Item(producto, cantidad, subtotal);

listaItems.Add(item);

mostrarItems();

mostrarTotales();

}

1. Ejecutamos la aplicación y validamos su funcionalidad.

Herencia Y Polimorfismo

Herencia

El mecanismo de herencia es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera (a la que se le suele llamar clase hija) será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda (a la que se le suele llamar clase padre o clase base). Las clases que derivan de otras se definen usando la siguiente sintaxis:

class <nombreHija>:<nombrePadre>

{

<miembrosHija>

}

Ejemplo:

public class Trabajador : Persona

{

public double Sueldo { get; set; }

}

Si tenemos la clase padre:

public class Persona

{

public string Nombres { get; set; }

public string ApellidoPaterno { get; set; }

public string ApellidoMaterno { get; set; }

public string Dni { get; set; }

public DateTime FechaNacimiento { get; set; }

public int Edad

{

get

{

TimeSpan diferencia = DateTime.Now - FechaNacimiento;

int anios = Convert.ToInt32(Math.Truncate(diferencia.Days / 365.2425));

return anios;

}

}

public Persona(string pNombres, string pApellidoPaterno, string pApellidoMaterno,

string pDni, DateTime pFechaNacimiento)

{

Nombres = pNombres;

ApellidoPaterno = pApellidoPaterno;

ApellidoMaterno = pApellidoMaterno;

Dni = pDni;

FechaNacimiento = pFechaNacimiento;

}

}

La definición de la clase hija sería:

public class Trabajador : Persona

{

public double Sueldo { get; set; }

public Trabajador(double pSueldo,

string pNombres, string pApellidoPaterno, string pApellidoMaterno,

string pDni, DateTime pFechaNacimiento)

: base(pNombres, pApellidoPaterno, pApellidoMaterno, pDni, pFechaNacimiento)

{

Sueldo = pSueldo;

}

}

Si queremos instanciar un objeto de la clase Trabajador lo haríamos así:

Trabajador trabajador = new Trabajador(3500, "Juan", "Perez", "Pino", "06168499", new DateTime(1990, 12, 28));

Nótese que ha sido necesario prefijar la definición de los miembros de Persona de la palabra reservada public. Esto se debe a que por defecto los miembros de una tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo public conseguimos que sean accesibles desde cualquier código.

Llamadas por defecto al constructor base:

Si en la definición del constructor de alguna clase que derive de otra no incluimos inicializador base, el compilador considerará que éste es :base().Es especialmente significativo reseñar el caso de que no demos la definición de ningún constructor en la clase hija, ya que en estos casos la definición del constructor que por defecto introducirá el compilador será en realidad de la forma:

<nombreClase>(): base()

{}

Es decir, este constructor siempre llama al constructor sin parámetros del padre del tipo que estemos definiendo, y si ése no dispone de alguno se producirá un error al compilar.

Polimorfismo: Métodos Virtuales

Ya hemos visto que es posible definir tipos cuyos métodos se hereden de definiciones de otros tipos. Lo que ahora vamos a ver es que además es posible cambiar dicha definición en la clase hija, para lo que habría que haber precedido con la palabra reservada virtual la definición de dicho método en la clase padre. A este tipo de métodos se les llama métodos virtuales, y la sintaxis que se usa para definirlos es la siguiente:

virtual <tipoDevuelto> <nombreMétodo>(<parámetros>)

{

<código>

}

Si en alguna clase hija quisiésemos dar una nueva definición del <código> del método, simplemente lo volveríamos a definir en la misma pero sustituyendo en su definición la palabra reservada virtual por override. Es decir, usaríamos esta sintaxis:

override <tipoDevuelto> <nombreMétodo>(<parámetros>)

{

<nuevoCódigo>

}

Nótese que esta posibilidad de cambiar el código de un método en su clase hija sólo se da si en la clase padre el método fue definido como virtual. En caso contrario, el compilador considerará un error intentar redefinirlo.

Laboratorio 2:

8.2 Herencia y Métodos virtuales.

Objetivos

* Entender y aplicar los conceptos de Herencia de la POO en el desarrollo de aplicaciones.

Introducción

Durante esta actividad, vamos aprender a declarar Clases heredadas y definir métodos virtuales.

Paso 1. Herencia.

1. Creamos el proyecto Semana10.
2. Agregamos la clase padre Persona mediante la opción: Agregar > Clase…

public class Persona

{

public string Nombres { get; set; }

public string ApellidoPaterno { get; set; }

public string ApellidoMaterno { get; set; }

public string Dni { get; set; }

public DateTime FechaNacimiento { get; set; }

public int Edad

{

get

{

TimeSpan diferencia = DateTime.Now - FechaNacimiento;

int anios = Convert.ToInt32(Math.Truncate(diferencia.Days / 365.2425));

return anios;

}

}

public Persona()

{

Nombres = "";

ApellidoPaterno = "";

ApellidoMaterno = "";

Dni = "";

FechaNacimiento = DateTime.Now;

}

public Persona(string pNombres, string pApellidoPaterno, string pApellidoMaterno,

string pDni, DateTime pFechaNacimiento)

{

Nombres = pNombres;

ApellidoPaterno = pApellidoPaterno;

ApellidoMaterno = pApellidoMaterno;

Dni = pDni;

FechaNacimiento = pFechaNacimiento;

}

public string ObtenerNombreCompleto()

{

return Nombres + " " +ApellidoPaterno + " " + ApellidoMaterno;

}

}

1. Agregamos la clase hija Trabajador mediante la opción: Agregar > Clase…

public class Trabajador : Persona

{

public double Sueldo { get; set; }

public Trabajador():base()

{

Sueldo = 0;

}

public Trabajador(double pSueldo,

string pNombres, string pApellidoPaterno, string pApellidoMaterno,

string pDni, DateTime pFechaNacimiento)

: base(pNombres, pApellidoPaterno, pApellidoMaterno, pDni, pFechaNacimiento)

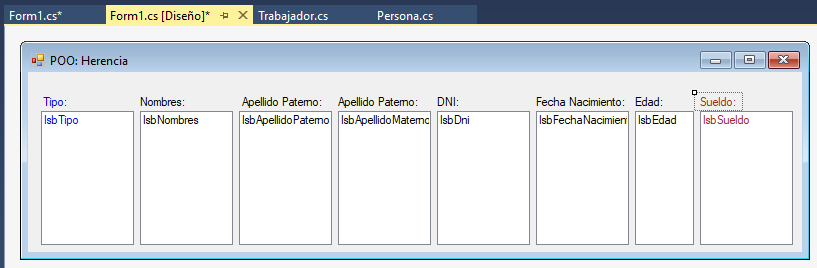
{

Sueldo = pSueldo;

}

}

1. Diseñamos el formulario de acuerdo a:



|  |  |  |
| --- | --- | --- |
| Control | Propiedad | Valor |
| Form1 | (Name) | frmHerencia |
|  | Text | POO: Herencia |
| ListBox1 | (Name) | lsbTipo |
|  | ForeColor | Blue [Web] |
| ListBox2 | (Name) | lsbNombres |
| ListBox3 | (Name) | lsbApellidoPaterno |
| ListBox4 | (Name) | lsbApellidoMaterno |
| ListBox5 | (Name) | lsbDni |
| ListBox6 | (Name) | lsbFechaNacimiento |
| ListBox7 | (Name) | lsbEdad |
| ListBox8 | (Name) | lsbSueldo |
|  | ForeColor | Brown [Web] |

1. Agregamos dos métodos en la clase del formulario que nos permita mostrar los datos de los objetos que se han creado:

public void MostrarPersona(Persona pPersona)

{

lsbTipo.Items.Add("Persona");

lsbNombres.Items.Add(pPersona.Nombres);

lsbApellidoPaterno.Items.Add(pPersona.ApellidoPaterno);

lsbApellidoMaterno.Items.Add(pPersona.ApellidoMaterno);

lsbDni.Items.Add(pPersona.Dni);

lsbFechaNacimiento.Items.Add(pPersona.FechaNacimiento.ToShortDateString());

lsbEdad.Items.Add(pPersona.Edad.ToString());

lsbSueldo.Items.Add("");

}

public void MostrarTrabajador(Trabajador pTrabajador)

{

lsbTipo.Items.Add("Trabajador");

lsbNombres.Items.Add(pTrabajador.Nombres);

lsbApellidoPaterno.Items.Add(pTrabajador.ApellidoPaterno);

lsbApellidoMaterno.Items.Add(pTrabajador.ApellidoMaterno);

lsbDni.Items.Add(pTrabajador.Dni);

lsbFechaNacimiento.Items.Add(pTrabajador.FechaNacimiento.ToShortDateString());

lsbEdad.Items.Add(pTrabajador.Edad.ToString());

lsbSueldo.Items.Add(pTrabajador.Sueldo.ToString());

}

1. Agregamos código al evento Load del formulario:

private void frmHerencia\_Load(object sender, EventArgs e)

{

Persona persona1 = new Persona();

Persona persona2 = new Persona();

persona2.Nombres = "Julia";

persona2.ApellidoPaterno = "Roldán";

persona2.ApellidoMaterno = "Robles";

persona2.Dni = "22598010";

persona2.FechaNacimiento = new DateTime(2000, 01, 31);

Persona persona3 = new Persona("Juan", "Perez", "Pino", "06168499",

new DateTime(1990, 12, 28));

Trabajador trabajador1 = new Trabajador();

Trabajador trabajador2 = new Trabajador();

trabajador2.Nombres = "Roberto";

trabajador2.ApellidoPaterno = "Camarena";

trabajador2.ApellidoMaterno = "Rivera";

trabajador2.Dni = "01012590";

trabajador2.FechaNacimiento = new DateTime(1970, 06, 30);

trabajador2.Sueldo = 3600;

Trabajador trabajador3 = new Trabajador(1850, "Lucy", "Peralta", "Zambrano", "16178493",

new DateTime(1976, 07, 02));

MostrarPersona(persona1);

MostrarPersona(persona2);

MostrarPersona(persona3);

MostrarTrabajador(trabajador1);

MostrarTrabajador(trabajador2);

MostrarTrabajador(trabajador3);

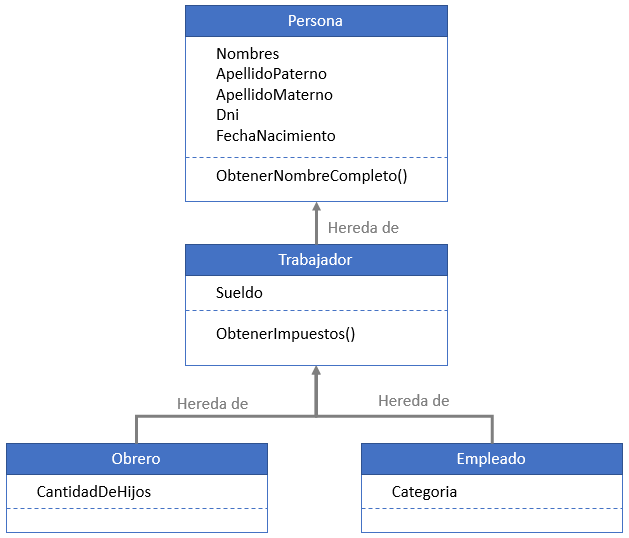
}

1. Ejecutamos la aplicación y validamos su funcionalidad.



Paso 2. Polimorfismo: Métodos Virtuales.

Para este ejemplo la clase Trabajador (del ejemplo anterior) será el padre de dos nuevas clases: Obrero y Empleado.



1. Agregamos una clase virtual a la clase Trabajador

public class Trabajador : Persona

{

public double Sueldo { get; set; }

public Trabajador():base()

{

Sueldo = 0;

}

public Trabajador(double pSueldo,

string pNombres, string pApellidoPaterno, string pApellidoMaterno,

string pDni, DateTime pFechaNacimiento)

: base(pNombres, pApellidoPaterno, pApellidoMaterno, pDni, pFechaNacimiento)

{

Sueldo = pSueldo;

}

virtual public double ObtenerImpuestos()

{

return 0;

}

}

1. Agregamos la clase hija Obrero mediante la opción: Agregar > Clase…

public class Obrero : Trabajador

{

public double CantidadDeHijos { get; set; }

public Obrero() : base()

{

CantidadDeHijos = 0;

}

override public double ObtenerImpuestos()

{

double impuestos = 0;

if (this.Sueldo > 1000)

impuestos = this.Sueldo \* 15 / 100; // 15% del Sueldo

else

impuestos = this.Sueldo\* 10 / 100; // 10% del Sueldo

return impuestos;

}

}

1. Agregamos la clase hija Empleado mediante la opción: Agregar > Clase…

public class Empleado : Trabajador

{

public string Categoria { get; set; }

public Empleado() : base()

{

Categoria = "";

}

override public double ObtenerImpuestos()

{

double impuestos = 0;

if (this.Sueldo > 3000)

impuestos = this.Sueldo \* 25 / 100; // 25% del Sueldo

else

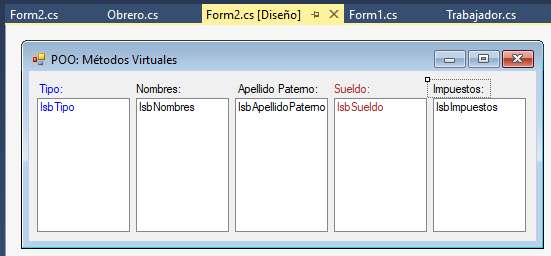
impuestos = this.Sueldo \* 20 / 100; // 20% del Sueldo

return impuestos;

}

}

1. Diseñamos el formulario de acuerdo a:



|  |  |  |
| --- | --- | --- |
| Control | Propiedad | Valor |
| Form2 | (Name) | frmMetodosVirtuales |
|  | Text | POO: Métodos Virtuales |
| ListBox1 | (Name) | lsbTipo |
|  | ForeColor | Blue [Web] |
| ListBox2 | (Name) | lsbNombres |
| ListBox3 | (Name) | lsbApellidoPaterno |
| ListBox4 | (Name) | lsbSueldo |
|  | ForeColor | Brown [Web] |
| ListBox5 | (Name) | lsbImpuestos |

1. Agregamos dos métodos en la clase del formulario que nos permita mostrar los datos de los objetos que se han creado:

public void MostrarObrero(Obrero pObrero)

{

lsbTipo.Items.Add("Obrero");

lsbNombres.Items.Add(pObrero.Nombres);

lsbApellidoPaterno.Items.Add(pObrero.ApellidoPaterno);

lsbSueldo.Items.Add(pObrero.Sueldo.ToString());

lsbImpuestos.Items.Add(pObrero.ObtenerImpuestos().ToString());

}

public void MostrarEmpleado(Empleado pEmpleado)

{

lsbTipo.Items.Add("Empleado");

lsbNombres.Items.Add(pEmpleado.Nombres);

lsbApellidoPaterno.Items.Add(pEmpleado.ApellidoPaterno);

lsbSueldo.Items.Add(pEmpleado.Sueldo.ToString());

lsbImpuestos.Items.Add(pEmpleado.ObtenerImpuestos().ToString());

}

1. Agregamos código al evento Load del formulario:

private void frmMetodosVirtuales\_Load(object sender, EventArgs e)

{

Obrero obrero1 = new Obrero();

obrero1.Nombres = "Roberto";

obrero1.ApellidoPaterno = "Camarena";

obrero1.Sueldo = 999;

obrero1.CantidadDeHijos = 3;

Obrero obrero2 = new Obrero();

obrero2.Nombres = "Miriam";

obrero2.ApellidoPaterno = "Vigo";

obrero2.Sueldo = 1001;

obrero2.CantidadDeHijos = 2;

Empleado empleado1 = new Empleado();

empleado1.Nombres = "Isabel";

empleado1.ApellidoPaterno = "Rodriguez";

empleado1.Sueldo = 3001;

empleado1.Categoria = "A2";

Empleado empleado2 = new Empleado();

empleado2.Nombres = "Rodrigo";

empleado2.ApellidoPaterno = "Luna";

empleado2.Sueldo = 2999;

empleado2.Categoria = "B1";

MostrarObrero(obrero1);

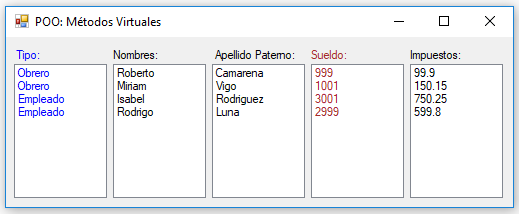
MostrarObrero(obrero2);

MostrarEmpleado(empleado1);

MostrarEmpleado(empleado2);

}

1. Ejecutamos la aplicación y validamos su funcionalidad.

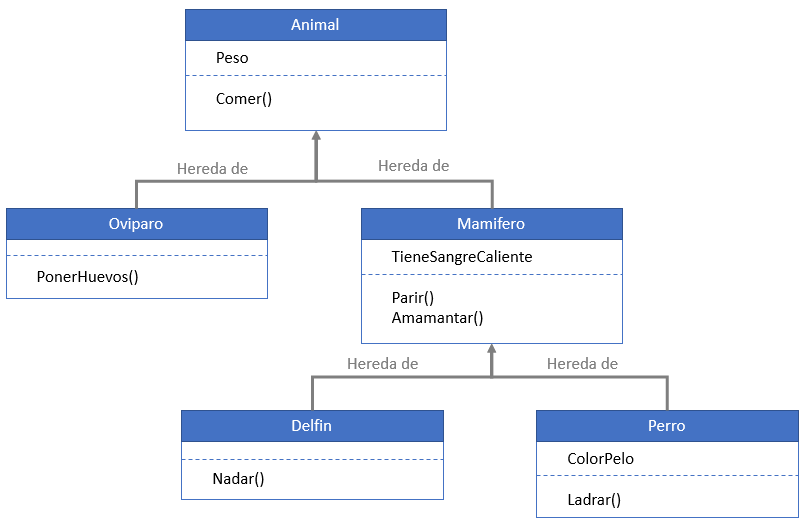


**Actividad:**

Revisa y analiza los siguientes enlaces en la plataforma virtual, luego realiza las actividades propuestas:

* **Clases y Objetos en C#.**
* https://www.youtube.com/watch?v=xz7gn9UXdrA&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=27
* <https://www.youtube.com/watch?v=kjEWZVglsnk&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=28>
* <https://www.youtube.com/watch?v=_eyFoySmHPk&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=29>
* https://www.youtube.com/watch?v=FobGQKXVpH0&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=30
* <https://www.youtube.com/watch?v=7HGwVYsdAxI&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=31>
* https://www.youtube.com/watch?v=HifATXjJ1KE&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=32
* **Herencia.**
* <https://www.youtube.com/watch?v=jvfUsKQgj88&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=41>
* <https://www.youtube.com/watch?v=Drs8FAtF1Ao&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=42>
* <https://www.youtube.com/watch?v=b8TkibB7CGg&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=43>
* https://www.youtube.com/watch?v=rXHjlBQgq7c&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=44
* **Polimorfismo.**
* https://www.youtube.com/watch?v=XzKL94OMDV4&list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B&index=45

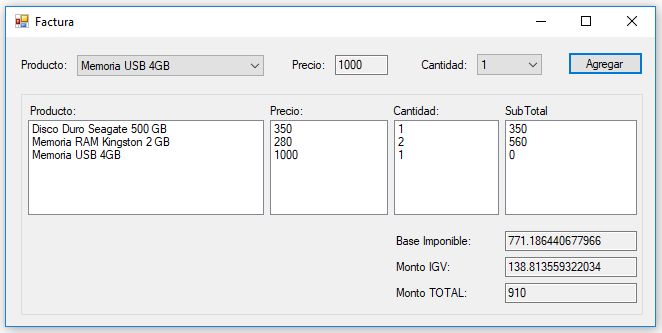
1. En el proyecto crear las clases, propiedades y métodos necesarios para implementar el siguiente diagrama de clases:



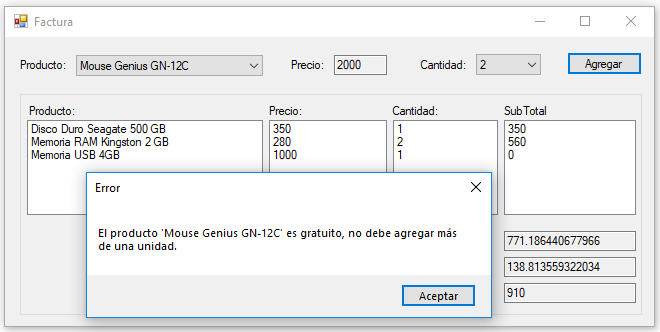
Restricciones:

* 1. Cada clase de tener dos constructores: uno sin parámetros y el otro que tenga los parámetros necesarios para inicializar los valores de todas sus propiedades.
  2. Los constructores de una clase hija debe hacer referencia al constructor de la clase de su padre.
  3. El método Comer() de la clase Animal debe ser virtual, por lo tanto debe redefinir su comportamiento en las clases: Oviparo, Delfin y Perro.

1. Hacer las siguientes modificaciones en el ejemplo 2 (Factura):
   1. Agregar a la clase Producto la propiedad “Gratuito” de tipo bool, esta propiedad indicara si el producto será entregado gratuitamente, esto es, no se cobrara cuando se factura, al margen del valor que tenga en la propiedad Precio.
   2. Agregar a la lista de productos dos productos que tengan la propiedad Gratuito con valor verdadero.



* 1. Cuando se agrega un producto gratuito debe validarse que no se puede entregar más de uno, el mensaje de advertencia debe mostrar el nombre del producto.



* 1. Agregar la funcionalidad de eliminar un ítem de la lista de ítems, cuando se elimine un ítem: debe eliminarse de la lista de ítems y mostrar los ítems que queden en los ListBox, también deben de recalcularse los totales.