Profesores: *Neiner Maximiliano, Villegas Octavio, Görner Mariano*

Nota:
Esta guía forma parte de un trabajo práctico (TP Nro. 1), que será entregado en dos partes (una antes del primer parcial y la otra antes del segundo parcial), las fechas de entrega serán publicadas por el profesor de Laboratorio II a su debido tiempo. Tomando siempre como fecha límite de entrega el último día disponible para regularizar la cursada de dicha materia (segunda fecha de finales).

Antes de empezar lea lo siguiente:

Para los ejercicios de esta guía, usted deberá asignarle al atributo 'Title'de la clase Console, el número de ejercicio, por ejemplo ***Console.Title = "Ejercicio Nro ##"***, donde ## será el número del ejercicio.
Del mismo modo se deberán nombrar las clases que contengan al método ***Main***, por ejemplo, ***Class Ejercicio_##***.
Para visualizar los valores decimales de los ejercicios, Ud. deberá dar el siguiente formato de salida al método Write/WriteLine: ***"#,###.00"***.

CONCEPTOS BASICOS

01. Ingresar 5 números por consola, guardándolos en una variable escalar. Luego calcular y mostrar: el valor máximo, el valor mínimo y el promedio.

02. Ingresar un número y mostrar: el cuadrado y el cubo del mismo. Se debe validar que el número sea mayor que cero, caso contrario, mostrar el mensaje: ***"ERROR. Reingresar número!!!"***.

    Nota: *Utilizar el método* ***'Pow'*** *de la clase* ***Math*** para realizar la operación.

03. Mostrar por pantalla todos los **números primos** que haya hasta el número que ingrese el usuario por consola.

    Nota: *Utilizar estructuras repetitivas, selectivas y la función módulo (%).*

04. Un **número perfecto** es un entero positivo, que es igual a la suma de todos los enteros positivos (excluido el mismo) que son divisores del número.
    El primer número perfecto es 6, ya que los divisores de 6 son 1, 2 y 3; y 1 + 2 + 3 = 6.
    Escribir una aplicación que encuentre los 4 primeros números perfectos.

    Nota: *Utilizar estructuras repetitivas, selectivas y la función módulo (%).*

05. Un **centro numérico** es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales.
El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1; 2; 3; 4; 5) y (7; 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595.
Se pide elaborar una aplicación que calcule los centros numéricos entre 1 y el número que el usuario ingrese por consola.

Nota: *Utilizar estructuras repetitivas, selectivas y la función módulo (%).*

06. Escribir un programa que determine si un año es bisiesto.
Un año es bisiesto si es múltiplo de 4. Los años múltiplos de 100 no son bisiestos, salvo si ellos también son múltiplos de 400.
Por ejemplo: el año 2000 es bisiesto pero 1900 no.

Nota: *Utilizar estructuras repetitivas, selectivas y la función módulo (%).*

07. Hacer un programa que pida por pantalla la fecha de nacimiento de una persona (día, mes y año) y calcule el número de días vividos por esa persona hasta la fecha actual (tomar la fecha del sistema con ***DateTime.Now***).

Nota*: Utilizar estructuras selectivas. Tener en cuenta los años bisiestos.*

08. Por teclado se ingresa el valor hora, el nombre, la antigüedad (en años) y la cantidad de horas trabajadas en el mes de 'n' empleados de una fábrica.
Se pide calcular el importe a cobrar teniendo en cuenta que el total (que resulta de multiplicar el valor hora por la cantidad de horas trabajadas), hay que sumarle la cantidad de años trabajados multiplicados por $ 150, y al total de todas esas operaciones restarle el 13% en concepto de descuentos.
Mostrar el recibo correspondiente con el nombre, la antigüedad, el valor hora, el total a cobrar en bruto, el total de descuentos y el valor neto a cobrar de todos los empleados ingresados.

Nota: *Utilizar estructuras repetitivas y selectivas.*

09. Escribir un programa que imprima por pantalla una pirámide como la siguiente:

```
*
***
*****
*******
*********
```

El usuario indicará cuál será la altura de la pirámide ingresando un número entero positivo. Para el ejemplo anterior la altura ingresada fue de 5.

Nota: *Utilizar estructuras repetitivas y selectivas.*

10. Partiendo de la base del ejercicio anterior, se pide realizar un programa que imprima por pantalla una pirámide como la siguiente:

```
    *
   ***
  *****
 *******
*********
```

Nota: *Utilizar estructuras repetitivas y selectivas.*

METODOS ESTATICOS (DE CLASE)

11. Ingresar 10 números enteros que pueden estar dentro de un rango de entre 100 y -100.
Para ello realizar una clase llamada **'Validacion'** que posea un método estático llamado **Validar**, que posea la siguiente firma:
bool Validar(int, int, int).
Terminado el ingreso mostrar el valor mínimo, el valor máximo y el promedio.

Nota: *Utilizar variables escalares, NO utilizar vectores.*

12. Realizar un programa que sume números enteros hasta que el usuario lo determine, por medio de un mensaje "¿Continua? (S/N)".
En el método estático ***ValidaS_N()*** de la clase **ValidarRespuesta**, se validará el ingreso de opciones.
El método **NO** recibe parámetros y devuelve un valor de tipo booleano, *TRUE* si se ingreso una 'S' y *FALSE* si se ingreso una 'N'.
El método deberá validar si otro caracter fue ingresado mostrando un mensaje de error y pidiendo el reingreso del mismo.

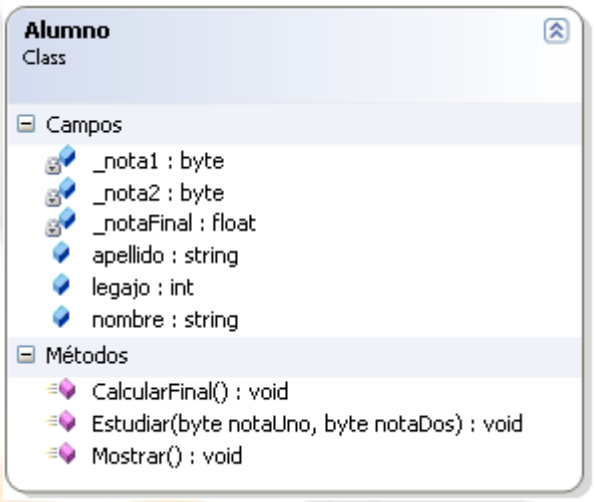
13. Desarrollar una clase llamada **'Conversor'**, que posea dos métodos de clase (**estáticos**):
    - ***string DecimalBinario(double)***. Convierte un número de decimal a binario.
    - ***double BinarioDecimal(string)***. Convierte un número binario a decimal.

14. Realizar una clase llamada **'CalculoDeArea'** que posea 3 métodos de clase (**estáticos**), ***double CalcularCuadrado(double)***, ***double CalcularTriangulo(double, double)*** y ***double CalcularCirculo(double)***, que realicen el cálculo del área que corresponda.
El ingreso de los datos como la visualización se deberán realizar desde el método Main().

15. Realizar un programa que permita realizar operaciones matemáticas simples (suma, resta, multiplicación y división).
Para ello se le debe pedir al usuario que ingrese dos números y la operación que desea realizar (pulsando el caracter +, -, * ó /).
El usuario decidirá cuando finalizar el programa.
Crear una clase llamada **Calculadora** que posea tres métodos estáticos (de clase):
    - **Calcular** (público): Recibirá tres parámetros, el primer número, el segundo número y la operación matemática. El método devolverá el resultado de la operación.
    - **Validar** (**Privado**): Recibirá como parámetro el segundo número. Este método se debe utilizar sólo cuando la operación elegida sea la DIVISION. Este método devolverá TRUE si el número es distinto de CERO.
    - **Mostrar** (público): Este método recibe como parámetro el resultado de la operación y lo muestra por pantalla. No posee valor de retorno.

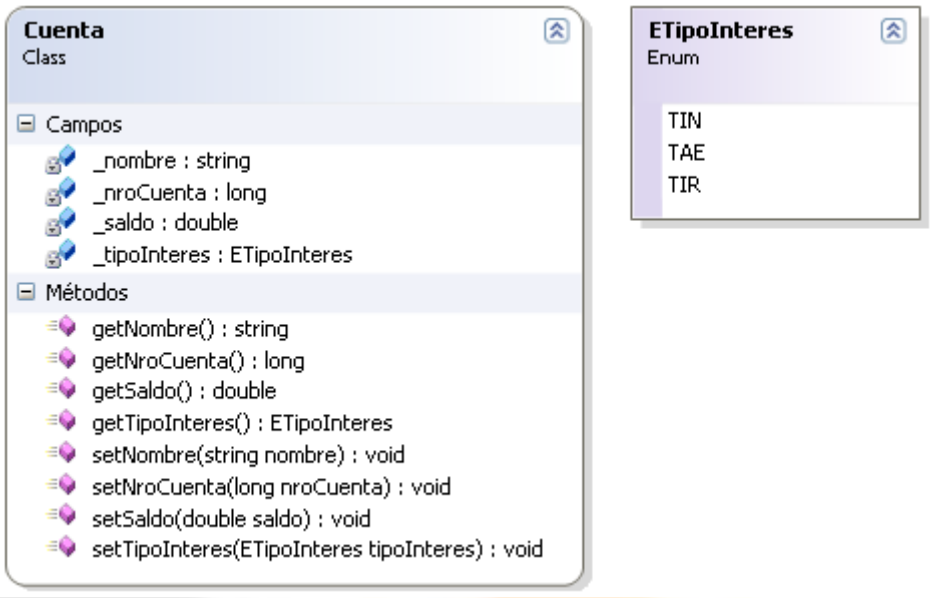
OBJETOS

16. Crear la clase 'Alumno' de acuerdo al siguiente diagrama:



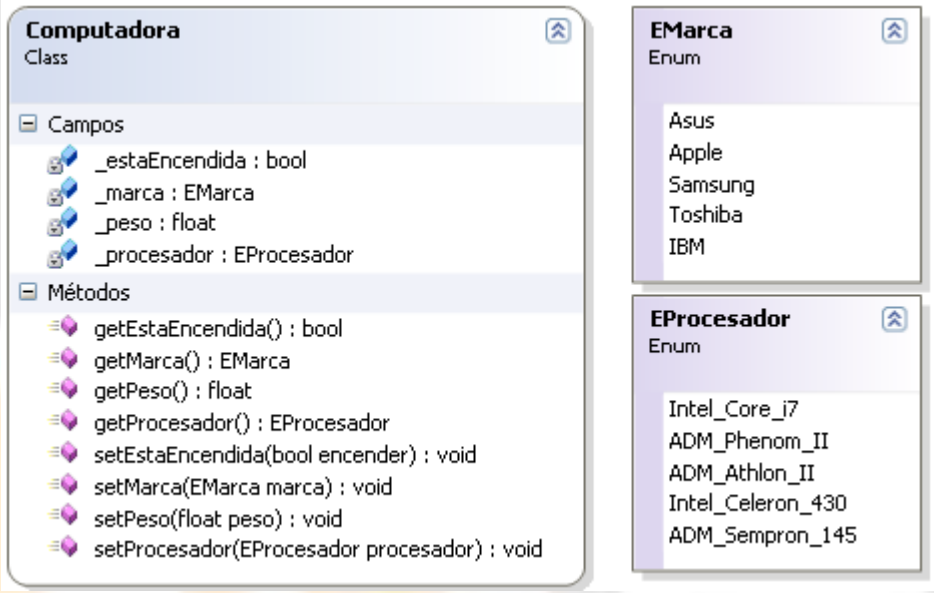
Se pide crear 3 instancias de la clase 'Alumno' (3 objetos), colocarle nombre, apellido y legajo a cada uno de ellos.
Sólo se podrá ingresar las notas (nota1 y nota2) a través del método **Estudiar**.
El método **CalcularFinal** deberá colocar la nota del final sólo si las notas 1 y 2 son mayores o iguales a 4, caso contrario la inicializará con -1. Para darle un valor a la nota final utilice el método de instancia **Next** de la clase **Random**.
Por último, el método Mostrar, expondrá en la consola todos los datos de los alumnos, a excepción de la nota final. Este valor se mostrará sólo si es distinto de -1.

17. Crear una clase llamada **Cuenta** y un enumerado llamado **ETipoInteres**, según se muestra en el siguiente diagrama:



Se pide crear los métodos públicos (***getters*** y ***setters***) necesarios para poder manipular un objeto de tipo **Cuenta**, sabiendo que dichos métodos deberán de realizar las validaciones correspondientes (en ningún caso se deberá pedir el reingreso de datos dentro del método).

18. Realizar una aplicación parecida a la del punto anterior pero con la clase **Computadora** y los enumerados **EMarca** y **EProcesador**:

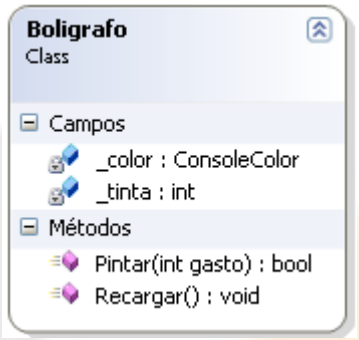


Agregar un **constructor** que reciba como parámetros todos sus atributos y los métodos:

1- **InformarEstado:** informa el estado actual de la computadora (marca, procesador, si está encendida o no, etc.).
2- **Encender.**
3- **Apagar**

Nota: *Las clases, los enumerados y el Program deben estar en namespaces distintos.*

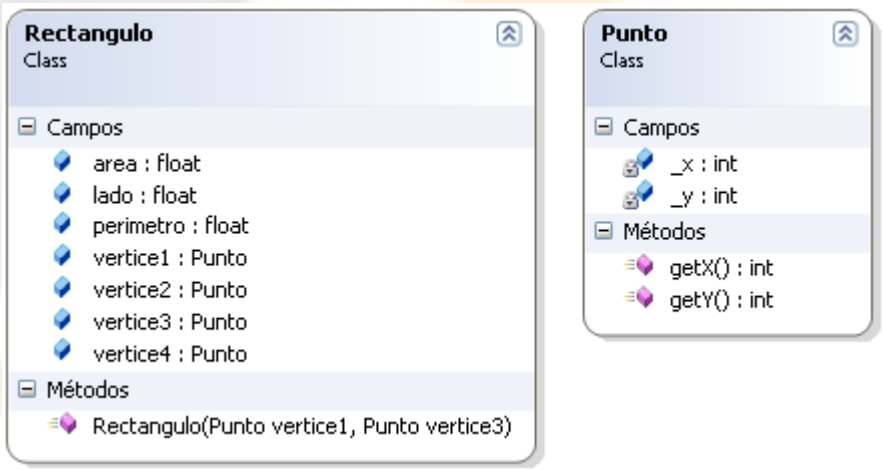
19. Crear la clase **Bolígrafo** a partir del siguiente diagrama:



Generar sólo los métodos ***getters*** para ambos atributos.
En el Main, crear un bolígrafo de tinta azul (***ConsoleColor.Blue***) y una cantidad inicial de tinta de 100 (cantidad máxima para todos los bolígrafos) y otro de tinta roja (***ConsoleColor.Red***) y 50 de tinta. Utilizar todos los métodos en el Main.

Nota: *Crear el constructor que crea conveniente. La clase Boligrafo y el Program deben estar en namespaces distintos.*

20. Escribir una aplicación con estos dos espacios de nombres (namespaces): **Geometría** y **PruebaGeometria.**
Dentro del espacio de nombres *Geometría* se deberán escribir dos clases: **Punto** y **Rectangulo.**



La clase **Punto** ha de tener dos atributos ***privados*** con acceso de sólo lectura (sólo con **getters**), que serán las coordenadas del punto.
La clase **Rectangulo** tiene los atributos de tipo **Punto** vertice1, vertice2, vertice3 y vertice4 (que corresponden a los cuatro vértices del rectángulo).
La base de todos los rectángulos de esta clase será siempre horizontal. Por lo tanto, debe tener un constructor para construir el rectángulo por medio de los vértices 1 y 3 (Utilizar el método **Abs** de la clase **Math**, dicho método retorna el valor absoluto de un número, para obtener la distancia entre puntos).
Los atributos lado, área y perímetro se deberán inicializar una vez construido el rectángulo.

En el espacio de nombres **PruebaGeometria** es donde se escribirá una clase con el método **Main** para probar si funcionan las clases escritas anteriormente. En este espacio de nombres se utilizará la directiva 'using' para poder utilizar todos los miembros del espacio de nombres **Geometría** directamente.
En la clase que contiene el Main, desarrollar un método de **clase** que muestre todos los datos del rectángulo que recibe como parámetro.

SOBRECARGA DE OPERADORES

21. Construir dos clases: la clase **Euro** y la clase **Dolar**. Se debe lograr que los objetos de estas clases se puedan sumar, restar, comparar, incrementar y disminuir con total normalidad como si fueran tipos numéricos, teniendo presente que 1 Euro equivale a 1.3642 dólares.
Además, tienen que ser compatibles entre sí y también con el tipo ***Double***. Sobrecargar los operadores **explicit** y/o **implicit**.

Nota: *Las clases y el Program deben estar en namespaces distintos.*

22. Crear tres clases: **Fahrenheit**, **Celsius** y **Kelvin**. Realizar un ejercicio similar al anterior, teniendo en cuenta que:
    F = C * (9/5) + 32
    C = (F-32) * 5/9
    F = K * 9/5 – 459.67
    K = (F + 459.67) * 5/9

Nota: *Las clases y el Program deben estar en namespaces distintos.*

23. Tomando la clase **Conversor** del ejercicio 13, se pide:

Agregar las clases:
- ***NumeroBinario***:
  - único atributo número (string)
  - único constructor (recibe un parámetro de tipo string)
- ***NumeroDecimal***
  - único atributo número (double)
  - único constructor (recibe un parámetro de tipo double)

Agregar sobrecarga de operadores:

Clase NumeroBinario

a. string + (NumeroBinario, NumeroDecimal) (*)
b. string - (NumeroBinario, NumeroDecimal) (*)
c. bool == (NumeroBinario, NumeroDecimal)
d. bool != (NumeroBinario, NumeroDecimal)

Clase NumeroDecimal

a. double + (NumeroDecimal, NumeroBinario) (*)

b. double - (NumeroDecimal, NumeroBinario) (*)
c. bool == (NumeroDecimal, NumeroBinario)
d. bool != (NumeroDecimal, NumeroBinario)

(*): Utilizar los métodos de la clase ***Conversor***.

Agregar las instrucciones necesarias en el 'Main()' para:

1. Instanciar la clase NumeroBinario (con el valor "1001")
2. Instanciar la clase NumeroDecimal (con el valor 9)

Operar con los números:

```
String nBinario = objBinario + objDecimal;
Double nDecimal = objDecimal + objBinario;
Mostrar los resultados por consola.
nBinario = objBinario - objDecimal;
nDecimal = objDecimal - objBinario;
Mostrar los resultados por consola.
Comparar ambos números y determinar si son iguales o no.
```

Agregar conversiones **implícitas** para poder ejecutar las siguientes instrucciones en el 'Main()':

```
NumeroBinario objBinario = "1001";
NumeroDecimal objDecimal = 9;
```

Nota: *Cambiar el modificador de visibilidad de los constructores, de las clases NumeroBinario y NumeroDecimal, de* ***public*** *a* ***private****.*

Agregar conversiones **explícitas** para poder ejecutar las siguientes instrucciones en el 'Main()':

```
Console.WriteLine("El valor en Binario es {0}",(string)objBinario);
Console.WriteLine("El valor en Decimal es {0}",(double)objDecimal);
```

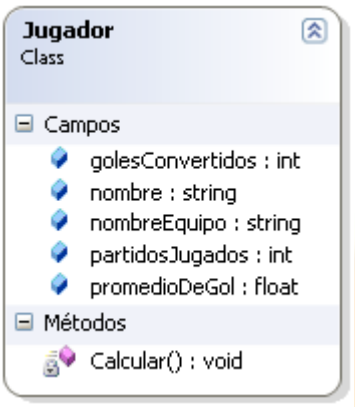
VECTORES Y COLECCIONES

24. Crear y cargar con valores enteros dos vectores A y B de dimensión 10. Generar un tercer vector C de 10 elementos donde la componente ***C[i]*** sea igual al mínimo valor de ***A[i]*** y ***B[i]***. Mostrar los tres vectores. Por último, mostrar de forma ascendente y descendente el vector C.

25. Leer 20 números enteros (positivos y negativos) distintos de cero. Mostrar el vector tal como fue ingresado y luego mostrar los positivos ordenados en forma decreciente y por último mostrar los negativos ordenados en forma creciente.

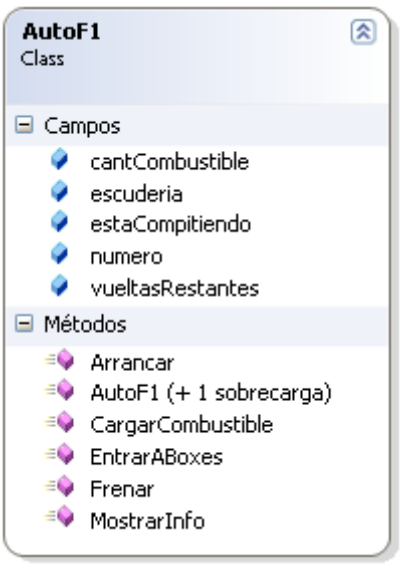
26. Realizar el ejercicio anterior pero esta vez con las siguientes colecciones: ***Pilas***, ***Colas*** y ***Listas***.

27. Crear una Clase llamada **Jugador**, que almacene la siguiente información sobre jugadores de fútbol:



Declarar una colección (**generics**), a la cual se le agregaran 10 elementos de tipo Jugador.
Escribir un programa que cargue los datos en la colección.
Se ingresarán: nombre del jugador, nombre del equipo, cantidad de partidos jugados y cantidad de goles convertidos.
Por cada jugador se calculará el promedio de goles por partido (utilizando el método **privado Calcular**) y se almacenará en el miembro correspondiente de cada elemento de la colección.
Luego de este proceso se debe mostrar por pantalla la información contenida en la colección (nombre del jugador, nombre del equipo, cantidad de partidos jugados, cantidad de goles convertidos y promedio de goles por partido), ordenados de mejor promedio a peor promedio de gol.
Esta información se deberá persistir en un **archivo de texto** llamado 'Jugadores.txt', ubicado en el directorio raíz.

28. Diseñar una clase llamada **AutoF1** con los siguientes atributos y métodos:



Realizar un programa que simule una carrera de F1 (de 15 vueltas).
Para ello instanciar 4 objetos e inicializarlos con valores que deberán ser validados.

La cantidad máxima de combustible será de 120 Kg y la cantidad mínima de 15 Kg.
Los autos consumen 10 Kg de combustible por vuelta. El programa deberá preguntarle al piloto si desea concurrir a Boxes cuando la cantidad de combustible sea menor a 25 Kg.
El piloto determinará cuando ingresar, sabiendo que cuando no tenga combustible (cantidad = 0), el auto se detendrá.
Tener en cuenta que por cada vez que se ingrese a Boxes se incrementará la cantidad de vueltas restantes en 1.
El programa tendrá que avisar a los competidores la cantidad de vueltas restantes a partir del último tercio de la competencia.
Ganará la competencia aquel auto con menor cantidad de vueltas restantes.

PROPIEDADES

29. Crear una clase llamada **Cuenta** que posea los siguientes atributos **privados**:
    - _nombre (String)
    - _nroCuenta (Int64)
    - _saldo (Double)
    - _tipoInteres (Single)

    Se pide crear las propiedades de lectura/escritura para cada atributo de la clase. La **propiedad** *Saldo* será de sólo lectura.
    Dicha clase tendrá que tener el método **void Depositar**(Int64) que acumulará en el atributo _saldo el valor del parámetro recibido.
    Además tendrá un método estático **void Mostrar**(Cuenta) que mostrará todos los datos del objeto recibido por parámetro.

30. Necesitamos una clase para almacenar los datos de una factura. Dichos datos son: nombre del cliente, teléfono, dirección, ciudad, provincia, código postal, CUIT o CUIL y porcentaje del IVA.
    Por otra parte tienes que tener presente que en una misma factura puede haber una o varias líneas de detalle con los siguientes datos cada una: cantidad, descripción, precio unitario e importe.
    Esta clase debe ofrecer, además, **propiedades** que devuelvan la base imponible (total sin IVA), la cuota de IVA y el total a pagar.
    Escribir también un método Main, que demuestre que funciona correctamente el programa de facturación.
    Nota: *Las clases y el Program deben estar en namespaces distintos.*

31. Teniendo la clase **Libro** con los siguientes atributos **privados**:
    - ISBN(*International Standard Book Number*) (String)
    - Titulo (String)
    - Autor (Clase)
    - Editorial (Clase)
    - Cantidad (Int64)

Los métodos, constructores y propiedades corren por cuenta del alumno.
Se pide diseñar una aplicación que luego de registrar las existencias de los libros se visualice un menú que le permita al bibliotecario elegir alguna de las siguientes opciones:

- ***OPCION 1***: PRESTAMOS DE LIBROS

El bibliotecario ingresará:
ISBN (De 1 a 100, caso contrario informar error y volver a ingresar) – Utilizar **propiedades**.
El programa le mostrará el autor y el título y sólo se prestará el libro si la cantidad de ejemplares es mayor a 1.
Si el usuario confirma el préstamo, se restará 1 a la cantidad de ejemplares.
Si la cantidad de ejemplares es 1, se mostrará la leyenda ***"Sólo queda el ejemplar de lectura en sala"*** y no se registrará el préstamo.

- ***OPCION 2***: DEVOLUCION DE LIBROS

El bibliotecario ingresará:
ISBN (De 1 a 100, caso contrario informar error y volver a ingresar) – Utilizar **propiedades**.
El programa le mostrará el autor y el título y si el usuario confirma, registrará la devolución, sumando 1 a la cantidad de ejemplares.

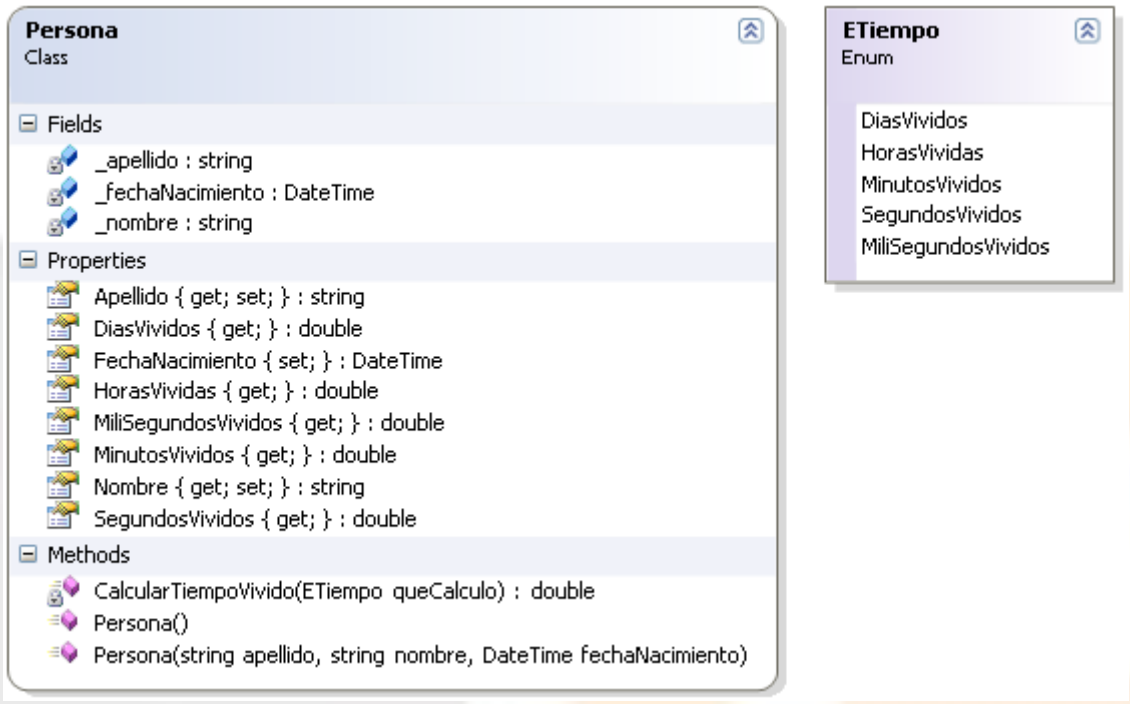
- ***OPCION 3***: FIN

Significa que no se sigue trabajando con el menú de opciones y que antes de finalizar debe informar lo siguiente:
Listado ordenado en forma decreciente por cantidad de veces que se prestó un libro:

TITULO	AUTOR	VECES PRESTADO
XXXXXXXXXXX	XXXXXXXXXXXXXXXXXXX	XXX

Esta información se deberá persistir en un **archivo de texto** llamado 'Libros.txt', ubicado en el directorio raíz.

32. Se desea crear una clase y un enumerado con las siguientes características:



Las propiedades de sólo lectura DiasVividos, HorasVividas, MiliSegundosVividos, MinutosVividos y SegundosVividos invocarán al método ***privado CalcularTiempoVivido***, el cual retornará un valor de tipo *Double* que representa el tiempo transcurrido entre el valor del atributo _fechaNacimiento y la fecha actual.

Nota: *Utilizar una variable de tipo* ***TimeSpan*** *para alojar la diferencia entre las fechas y de acuerdo al valor del enumerado* ***ETiempo***, *utilizar las propiedades correspondientes de dicha variable.*

MODELOS PRIMER PARCIAL

33. BOLIGRAFO (2009)

1) Realizar la clase Bolígrafo que posea tres atributos privados _marca (String), _color (string), _canitdadTinta (int).
   a. Colocarla en un **"namespace"** distinto al Program
   b. Realizar 4 (cuatro) sobrecargas del constructor, todas tienen como parámetro color y marca, las que no reciben como parámetro una cantidad, deberán inicializar ese atributo en 0 (cero).
2) Crear el método ***privado*** de ***instancia*** llamado "Mostrar" (sin parámetros) que mostrará todos los atributos del objeto bolígrafo por consola.
3) Método "MostrarBoligrafos": método público de ***clase*** que tiene como parámetro una lista genérica de bolígrafos y que nos mostrará los atributos de cada objeto contenido en la lista llamando al método "Mostrar" creado en el punto anterior .
4) Método "Escribir": método de instancia, público, que genera un consumo de tinta en el bolígrafo, la cantidad es recibida por parámetro y si la tinta que contiene el bolígrafo no es suficiente se deberá informar por consola los datos del bolígrafo utilizando el método "Mostrar".
5) Método "RecargarTinta" (Sobrecargado): método de instancia, público, sin parámetros, permite recargar (poner en 100) la cantidad de tinta del bolígrafo sólo si su tinta es menor a 50, este método retornará **"true"** si pudo recargar y **"false"** si no pudo realizar la acción.
   a. Sobrecargar: realiza la misma acción pero recibe como parámetro la cantidad de tinta que se le agregara al objeto solo si su tinta es mayor a 50.
6) Sobrecarga del operador "==" permite saber si dos objetos comparten la misma marca y el mismo color.
7) Sobrecarga del operador "+", recibe como parámetros una lista de bolígrafos y un bolígrafo, agregar ese bolígrafo a la lista y retorna la lista.

```
miLista = miLista + Otrobligrafo;
```

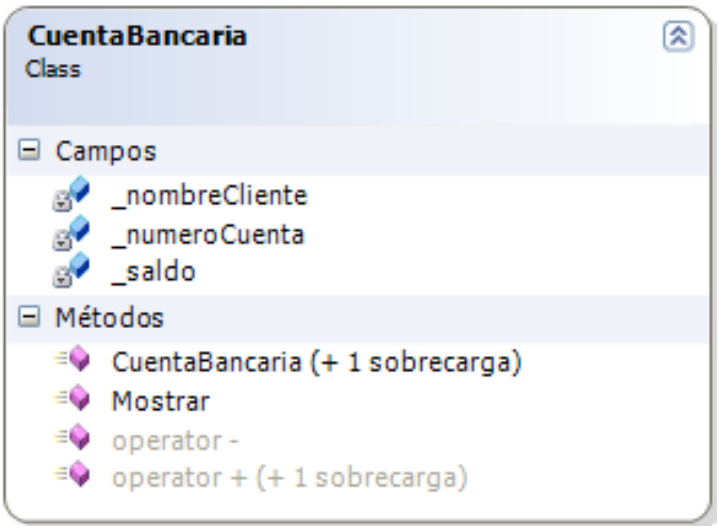
8) Sobrecarga del operador "-", recibe como parámetros una lista de bolígrafos y un bolígrafo, eliminar el objeto bolígrafo de la lista que coincida con los datos del objeto recibido por parámetros (utilizar el operador "==") y retorna la lista.

```
miLista = miLista - BoligrafoTres;
```

```
static void Main(string[] args) {

    List<Boligrafo> miLista = new List<Boligrafo>();
    Boligrafo Unboligrafo = new Boligrafo("blanco","bic",60);
    Unboligrafo.Escribir(70);

    if (Unboligrafo.RecargarTinta(90))
        Console.WriteLine("Boligrafo Recargado");
    else
        Console.WriteLine("No se pudo recargar");

    Unboligrafo.Escribir(30);

    if (Unboligrafo.RecargarTinta())
        Console.WriteLine("Boligrafo Recargado");
    else
        Console.WriteLine("No se pudo recargar");

    miLista = miLista + Unboligrafo;

    Boligrafo boligrafo2 = new Boligrafo("rojo","Silvapen",50);
    miLista.Add(boligrafo2);
    Console.WriteLine("                Muestro :");
    Boligrafo.MostrarBoligrafos(miLista);
    Console.ReadLine();
    Console.WriteLine("                 Sumo :");
    Boligrafo Otrobligrafo = new Boligrafo("verde", "Parker", 60);
    miLista = miLista + Otrobligrafo;
    Boligrafo.MostrarBoligrafos(miLista);
    Console.ReadLine();
    Console.WriteLine("                  Resto :");
    Boligrafo BoligrafoTres = new Boligrafo(40, "verde", "Parker");
    miLista = miLista - BoligrafoTres;
    Boligrafo.MostrarBoligrafos(miLista);
    Console.ReadLine();

}
```

34. CUENTA BANCARIA (2010)

Realizar una clase llamada **CuentaBancaria** que posea los siguientes atributos privados:



- _nombreDelCliente (String)
- _numeroCuenta (Int32)
- _saldo (Double)

Sobrecargar su constructor para poder instanciar objetos pasándole como parámetros:

a) El número de cuenta y el nombre del cliente (primera sobrecarga)
b) El número de cuenta, el nombre del cliente y el saldo (segunda sobrecarga).

Realizar un método de **clase** llamado "Mostrar" que muestre todos los atributos del objeto CuentaBancaria que recibirá dicho método por parámetro.

Sobrecargar el operador "+" para que permita sumar dos cuentas bancarias, sólo si son del mismo cliente (clientes con el mismo nombre), de lo contrario informarlo por consola, y que retorne un Double con la suma de los importes de cada cuenta bancaria. Ejemplo:

```
Double importeDouble = primeraCuenta + segundaCuenta;
```

Sobrecargar el operador "+" para que me permita sumar un importe (de tipo Double) a un objeto CuentaBancaria. Ejemplo:

```
Cuenta primeraCuenta = primeraCuenta + 1600;
```

Sobrecarga el operador "-" para poder restar un importe (de tipo Double) a un objeto CuentaBancaria. Dentro de la sobrecarga del operador se deberá verificar que se disponga del saldo suficiente para poder realizar dicha operación y de no ser así informar por consola. Ejemplo:

```
Cuenta primeraCuenta = primeraCuenta - 1000;
```

Nota: La clase CuentaBancaria deberá estar en un namespace formado por el apellido y el nombre de pila del alumno, separados por un punto. Ejemplo Perez.Juan.
La clase Program (que contiene el método Main()) deberá estar en un namespace distinto al de la clase CuentaBancaria.
Realizar las propiedades que considere necesarias.

35. AUTO - GARAGE – con ARRAY (2011)

1) Realizar una clase llamada "**Auto**" que posea los siguientes atributos **privados**:
   a. _color (String)
   b. _precio (Double)
   c. _marca (Enum EMarcas{Ford, Ferrari, Lotus}).
   d. _fecha (DateTime)

   a. La clase "**Auto**" deberá estar en un **namespace** formado por el apellido y el nombre de pila del alumno, separados por un punto. Ejemplo Perez.Juan.
      La clase Program (que contiene el método Main) deberá estar en un **namespace** distinto al de la clase "**Auto**".

   b. Sobrecargar su constructor para poder instanciar objetos pasándole como parámetros:
      i. La marca y el color (primera sobrecarga)
      ii. La marca, color y el precio (segunda sobrecarga).
      iii. Y tres sobrecargas más, de las cuales dos deben tener como parámetro la fecha.

2) Realizar un método de **instancia** llamado "**AgregarImpuestos**", que recibirá un doble por parámetro y que se sumará al precio del objeto.

3) Realizar un método de **clase** llamado "**MostrarAuto**", que recibirá un objeto de tipo "**Auto**" por parámetro y que mostrará todos los atributos de dicho objeto.

4) Sobrecargar el operador **"=="** que permita comparar dos objetos de tipo **"Auto"**. Sólo devolverá *TRUE* si ambos **"Autos"** son de la misma marca.

5) Sobrecargar el operador **"+"** para que permita sumar dos objetos **"Auto"** (sólo si son de la misma marca, utilizar la sobrecarga del ==, y del mismo color, de lo contrario informarlo por consola) y que retorne un **Double** con la suma de los precios o cero si no se pudo realizar la operación.

Ejemplo: importeDouble = AutoUno + AutoDos;

6) En el Main:

a. Crear **dos** objetos **"Auto"** de la misma marca y distinto color.
b. Crear **dos** objetos **"Auto"** de la misma marca, mismo color y distinto precio.
c. Crear tres objetos **"Auto"** utilizando las **tres** sobrecargas restantes.
d. Utilizar el método **"AgregarImpuesto"** en los últimos tres objetos, agregando $ 1000 al atributo precio.
e. Colocar esta línea en el Main:

importeDouble = AutoUno + AutoDos;

y mostrar el resultado por consola.
f. Comparar el primer **"Auto"** con el segundo y quinto objeto e informar si son iguales o no.
g. Utilizar el método de clase **"MostrarAuto"** para mostrar cada los objetos impares (1, 3, 5, 7).

Bonus

Crear la clase Garage que posea como atributos privados:
a. _razonSocial (String)
b. _precioPorHora (Double)
c. _autos (Autos[])

Sobrecargar su constructor para poder instanciar objetos pasándole como parámetros:
i. La razón social y precio por hora (primera sobrecarga)
ii. La razón social, y el precio por hora (segunda sobrecarga).
iii. Y dos sobrecargas más.

Nota: Todas las sobrecargas deben inicializar el Array de Autos con 5 elementos. El constructor por defecto debe ser **privado**.

Realizar un método de **instancia** llamado **"MostrarGarage"**, que no recibirá parámetros y que mostrará todos los atributos del objeto.

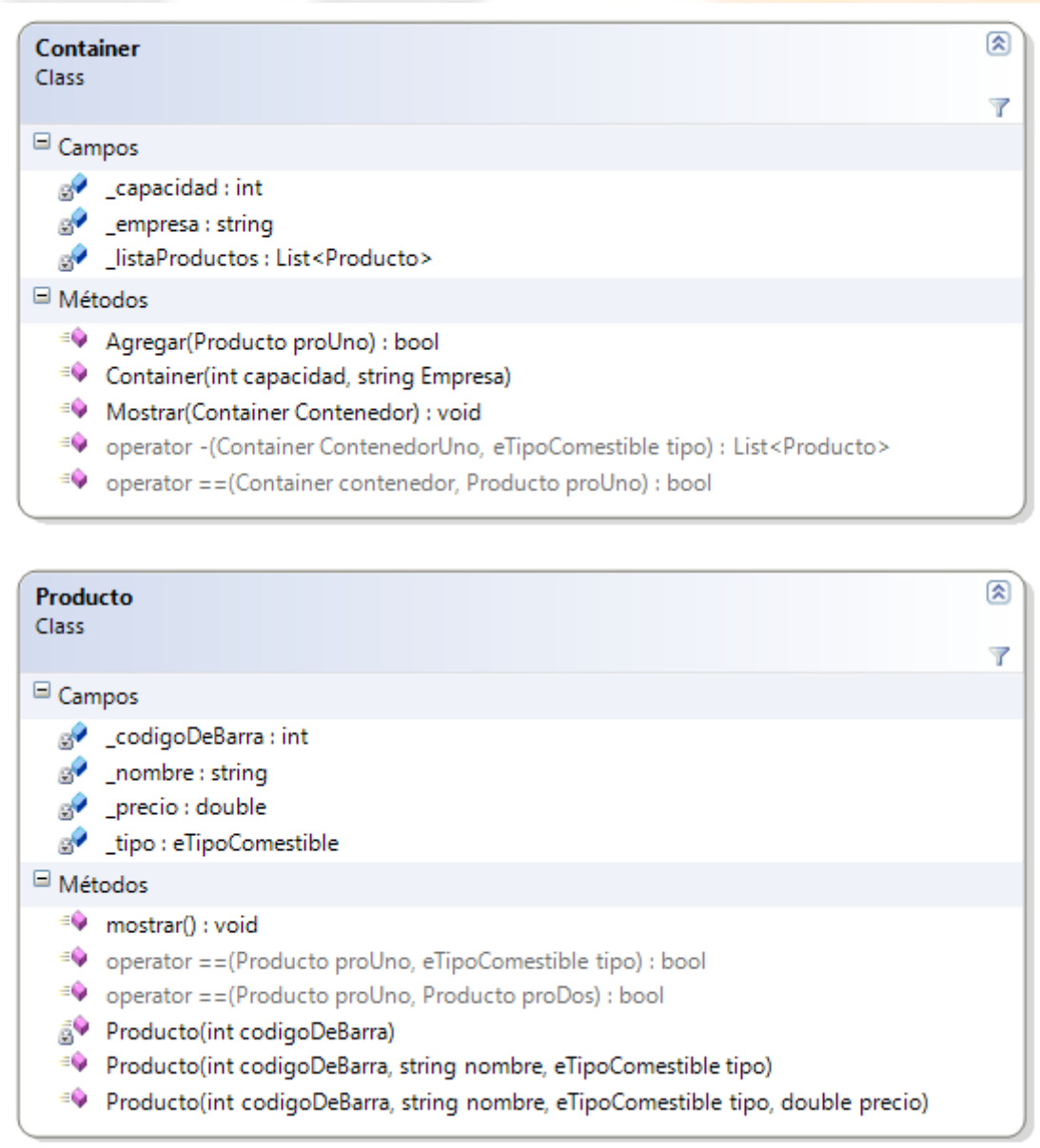
Sobrecargar el operador **"=="** que permita comparar al objeto de tipo **Garaje** con un objeto de tipo **Auto**. Sólo devolverá *TRUE* si el auto está en el garaje.

Sobrecargar el operador **"+"** para que permita sumar un objeto **"Auto"** al **"Garage"** (sólo si el auto **no** está en el garaje, utilizar la sobrecarga del ==, de lo contrario informarlo por consola) y que retorne un **Garage.**

Ejemplo: miGarage += AutoUno;

36. CONTAINER – PRODUCTO (2011)

1) Crear las siguientes clases



Teniendo en cuenta que

- El constructor Producto(int, string, eTipoComestible) **sólo** inicializará el nombre y el tipo del producto
- El constructor Producto(int, string, eTipoComestible, double) **sólo** inicializará el precio del producto
- Todas las instancias de producto deben de alguna forma inicializar el atributo *_codigoDeBarra*
- El constructor de la clase Container deberá instanciar la lista

NO SE PUEDEN USAR PROPIEDADES, LA ASIGNACIÓN DE LOS ATRIBUTOS SE DEBE HACER MEDIANTE LOS CONSTRUCTORES

CLASE PRODUCTO

2) Crear el método de instancia *mostrar()* que muestre por pantalla todos los atributos del producto.

3)

a) Crear una sobrecarga del operador "==" que permita comparar dos productos y devuelva verdadero sólo cuando estos sean iguales

b) Crear otra sobrecarga del operador "==" que devuelva verdadero sólo si el producto pertenece al tipo de comestible con el que se compara

CLASE CONTAINER

4) Crear un método de clase *Mostrar (Container)* que muestre los datos del contenedor pasado por parámetro. Se deberá mostrar también la lista completa de sus productos.

5) Crear la sobrecarga del operador "==" de modo que devuelva verdadero sólo si el producto se encuentra en la lista.

6) Sobrecargar el operador "-" para que retorne una lista con todos los productos que pertenecen al tipo indicado.

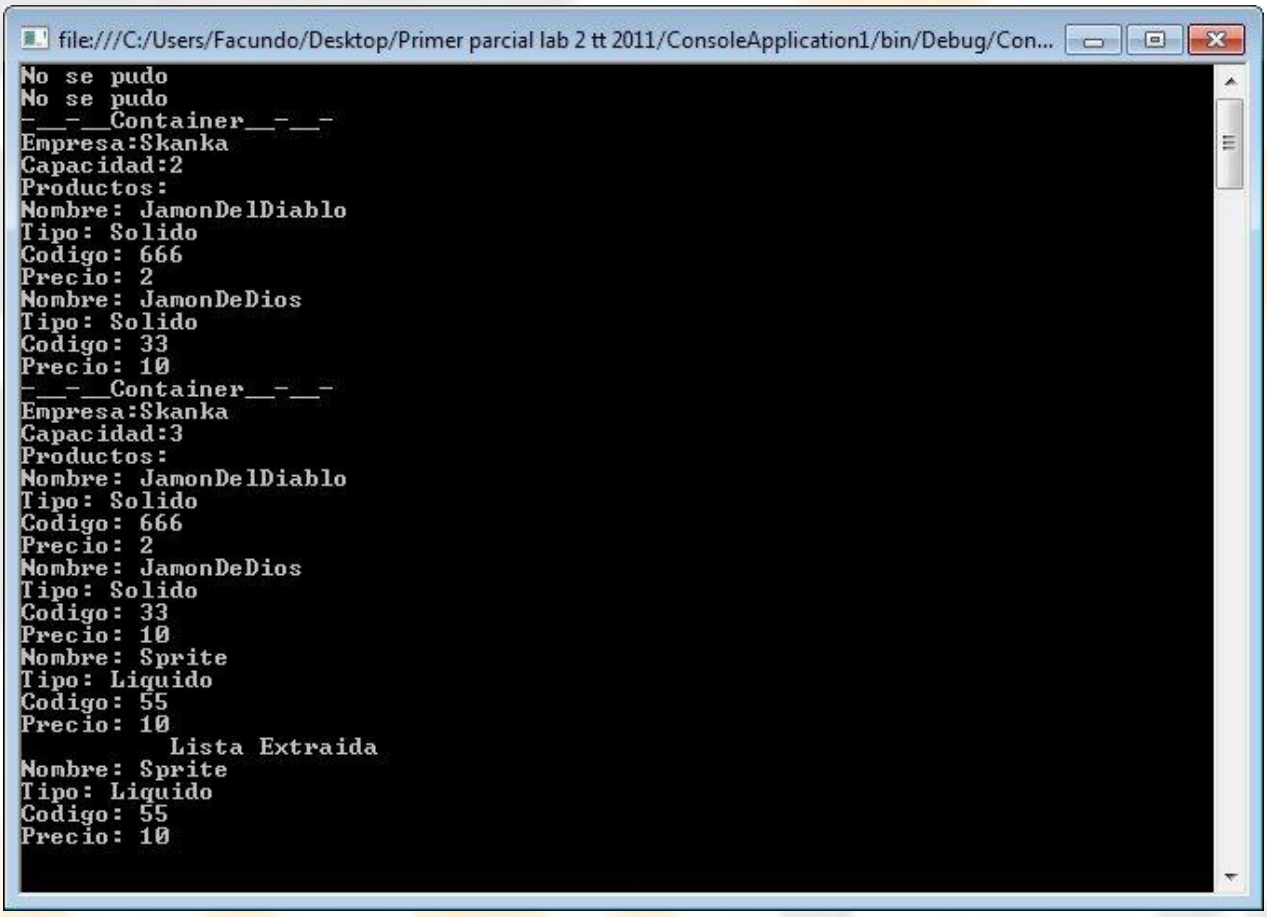
7) Crear el método de instancia *Agregar (Producto)* que agregue a la lista de productos del contenedor el producto pasado por parámetro sólo si la capacidad lo permite y el producto no se encuentra ya en la lista.

El alumno deberá copiar las siguientes líneas correspondientes al método Main de la clase Program. Dichas líneas **NO PORDRÁN SER MODIFICADAS**

```
static void Main(string[] args)
{

   Producto ProductoUno = new Producto(666,"JamonDelDiablo", eTipoComestible.Solido,2);
   Producto ProductoDos = new Producto(33, "JamonDeDios", eTipoComestible.Solido,10);
   Producto ProductoTres = new Producto(55, "Sprite", eTipoComestible.Liquido, 10);
   Producto ProductoCuatro = new Producto(666, "JamonDelDiablo", eTipoComestible.Solido);

   Container ContenerdorUno = new Container(2, "Skanka");
   Container ContenerdorDos = new Container(3, "Skanka");
   List<Producto> listaProducto = new List<Producto>();

   ContenerdorUno.Agregar(ProductoUno);
   ContenerdorUno.Agregar(ProductoDos);
   if(! ContenerdorUno.Agregar(ProductoTres)) {
      Console.WriteLine("No se pudo");
   }

   ContenerdorDos.Agregar(ProductoUno);
   ContenerdorDos.Agregar(ProductoDos);

   if (!ContenerdorDos.Agregar(ProductoCuatro)) {
       Console.WriteLine("No se pudo");
   }

   ContenerdorDos.Agregar(ProductoTres);
```

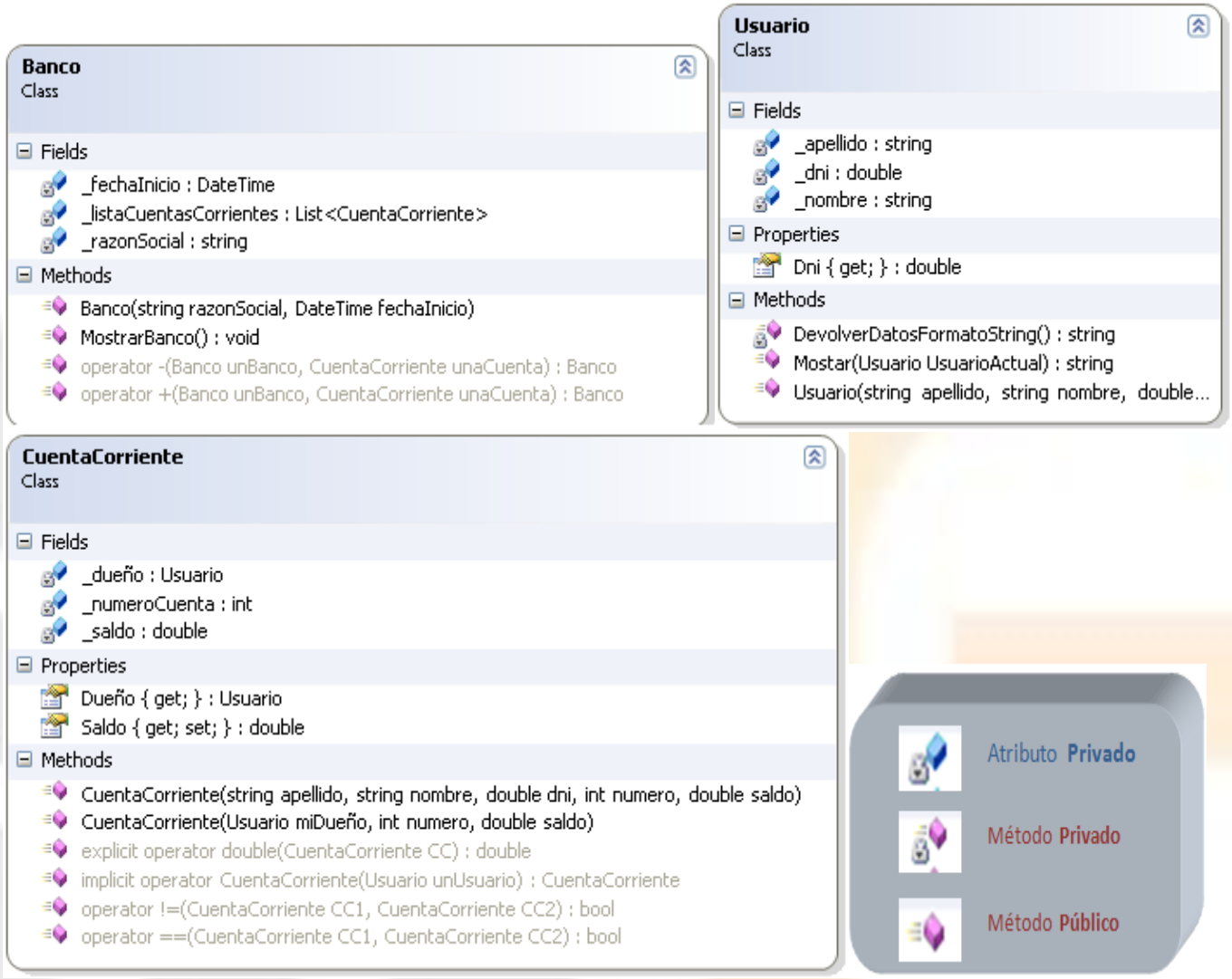
```
   Container.Mostrar(ContenerdorUno);
   Container.Mostrar(ContenerdorDos);

   listaProducto = ContenerdorDos - eTipoComestible.Liquido;

   Console.WriteLine("          Lista Extraida             ");
   foreach (Producto item in listaProducto) {
      item.mostrar();
   }

   Console.ReadLine();
}
```

Y la salida por consola deberá ser **EXCACTAMENTE** la siguiente



37. CUENTA CORRIENTE – BANCO (2011)

Siguiendo a este diagrama de clases como guía, se deberán crear los miembros de las clases que aquí se detallan:



Importante: No se podrán agregar métodos y tampoco se podrán cambiar los modificadores de visibilidad. Respetar los nombre del diagrama de clases.

1- **Constructores:** Realizar los constructores de las clases "***Usuario***", "***CuentaCorriente***" y "***Banco***", siguiendo el detalle del diagrama de clases (Misma firma y mismos nombres de parámetros), son 4 (cuatro) constructores en total.

2- **Métodos de instancia:**

   a. En la clase ***Usuario*** crear el método privado **"DevolverDatosFormatoString"** que retornará un dato de tipo **"String".** Usar la clase ***StringBuilder***.

   b. El método **"Mostrar"** de la clase ***Banco***, permite mostrar los datos del banco además de la lista de cuentas corrientes actuales. Tener en cuenta que por cada cuenta se debe mostrar los datos del usuario. Usar la clase ***StringBuilder***.

3- **Métodos de clase:**

   a. En la clase ***Usuario*** se creará un método de clase público llamado **"Mostrar"** que tiene como parámetro un objeto de tipo usuario (como lo muestra el diagrama de clases), dentro del método se **invocará** al método de instancia **"DevolverDatosFormatoString".**

4- **Sobrecarga de Operadores:**

   a. Se necesita comparar dos objetos de tipo CuentaCorriente, y retornar true sólo si las dos cuentas tienen el mismo usuario (**Dos usuarios son iguales si y solo si comparten el mismo DNI**). Para el operador != solo se permite escribir una línea de código.

   b. **Operadore + y –** de la clase ***Banco***, permiten agregar y eliminar una cuenta corriente de la lista de cuentas de la clase. Realizan estas operaciones si y solo si existe la cuenta corriente en la lista. Informar de lo acontecido.

5- **Operadores de Conversión:**

   a. Realizar el operador para convertir ***explícitamente*** un objeto de tipo ***CuentaCorriente*** en un dato de tipo **Double**, para poder poner estas líneas en el método **"MostrarBanco"** de la clase ***Banco***.

```
sb.AppendLine("Saldo: " + (double) CC);
```

   b. Realizar el operador de conversión que me permita asignar un usuario a una cuenta, retornando una cuenta con el usuario como "dueño" y todos los demás datos en 0 (cero). Para poder poner estas líneas en el Main:

```
Usuario dos = new Usuario("Mercurio", "Alfredo", 222);
CuentaCorriente objCC4 = dos;
```

6- **Realizar las siguientes propiedades:**

   a. Dni (sólo lectura)

   b. Saldo (lectura/escritura) deberá ***acumular*** el saldo con un valor especificado

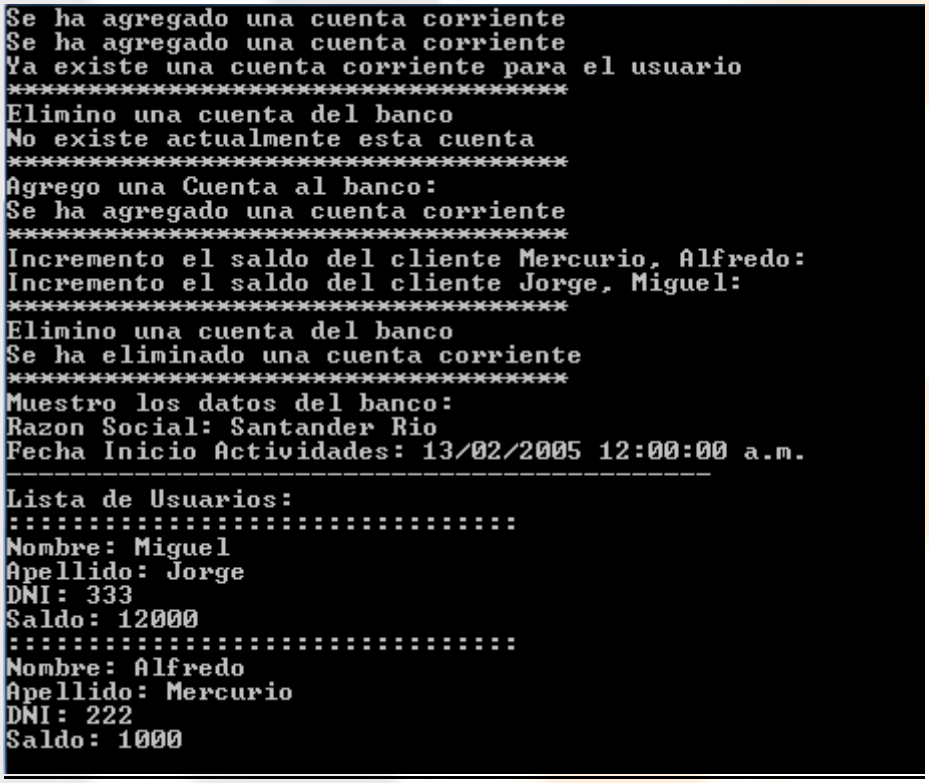
   c. Dueño(sólo lectura)

Este es el Main:

```
//Instancia de la clase Usuario: uno("Agua","Rogelio",111)
//Instancia de la clase Usuario: dos("Mercurio","Alfredo",222)

//Instancia de la clase Banco: unBanco("Santander Rio", 13/02/2005)

//Instancia de la clase CuentaCorriente: objCC1(uno, 789, 1000)
//Instancia de la clase CuentaCorriente: objCC2("Jorge", "Miguel", 333, 798, 5000)
//Instancia de la clase CuentaCorriente: objCC3(uno, 799, 1000)
//Instancia de la clase CuentaCorriente: objCC4=dos

            unBanco += objCC1;
            unBanco += objCC2;
            unBanco += objCC3;

            Console.WriteLine("**********************************");
            Console.WriteLine("Elimino una cuenta del banco ");
            unBanco -= objCC4;
            Console.WriteLine("**********************************");
            Console.WriteLine("Agrego una Cuenta al banco: ");
            unBanco += objCC4;
            Console.WriteLine("**********************************");
            Console.WriteLine("Incremento el saldo del cliente Mercurio, Alfredo: ");
            objCC4.Saldo = 1000;
            Console.WriteLine("Incremento el saldo del cliente Jorge, Miguel: ");
            objCC2.Saldo =7000;
            Console.WriteLine("**********************************");
            Console.WriteLine("Elimino una cuenta del banco ");
            unBanco -= objCC1;
```

```
Console.WriteLine("**********************************");

Console.WriteLine("Muestro los datos del banco: ");
unBanco.MostrarBanco();

Console.ReadLine();
```

Esta debe ser la salida exacta del programa:



38. PASAJERO – VUELO (2012)

Dadas las siguientes clases (que deberán estar en distintos espacios de nombres):

Pasajero

a. Atributos **privados**: _apellido (string), _nombre (string), _dni (string), _esPlus (bool)
b. Crear 2 sobrecargas del constructor que reciban 4 parámetros cada una.
c. Sobrecargar el operador ==, permitirá comparar 2 objetos Pasajero. Retornará verdadero cuando el _dni sea igual.
d. Agregar una propiedad de sólo lectura llamada InfoPasajero, que retornará una cadena de caracteres con los atributos concatenados.
e. Agregar un método de clase llamado MostrarPasajero que mostrará los atributos por consola, utilizar la propiedad de sólo lectura.

Vuelo

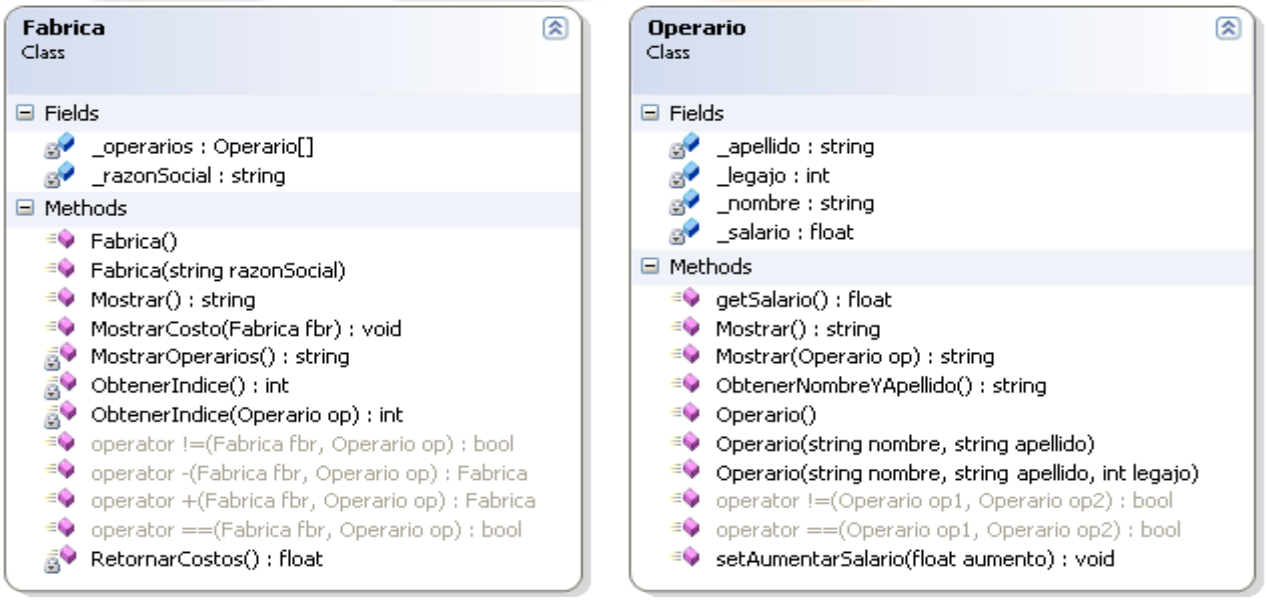
f. Atributos **privados y de instancia**: _fecha (DateTime), _empresa (string) _precio (decimal), _listaDePasajeros (lista genérica de elementos tipo

Pasajero), _cantMaxima (propiedad int de sólo lectura). Tanto _listaDePasajero como _cantMaxima sólo se inicializarán en un constructor.

g. Crear 2 sobrecargas del constructor, que reciban 3 parámetros como mínimo.
h. Agregar una propiedad de sólo lectura, que devuelva en una cadena de caracteres toda la información de un vuelo: fecha, empresa, precio, cantidad máxima de pasajeros, y toda la información de **todos** los pasajeros.
i. Crear un método de **instancia** llamado AgregarPasajero, en el caso que no exista en la lista, se agregará (utilizar operador == para verificar existencia). Además tener en cuenta la capacidad del vuelo. El valor de retorno de este método indicará si se agregó o no.
j. Agregar un método de instancia llamado MostrarVuelo, que mostrará la información de un vuelo.
k. Sobrecargar el operador + para que permita sumar 2 vuelos. El valor devuelto deberá ser de tipo decimal, y representará el valor recaudado por el vuelo. Tener en cuenta que si un pasajero es **Plus**, se le hará un descuento del 20% en el precio del vuelo.
l. Agregar el código necesario en la clase vuelo para realizar, por ejemplo, la siguiente asignación: montoTotalVuelo = unVuelo; siendo montoTotalVuelo una variable de tipo decimal, y unVuelo es de tipo Vuelo. Luego de realizar la asignación, montoTotalVuelo deberá tener el valor total del vuelo (también tener en cuenta el criterio para pasajeros Plus)

39. FABRICA – OPERARIO - con ARRAY (2012)

Se desea realizar el control de operarios de una fábrica, para eso deberemos crear las siguientes clases:



1) Métodos getters y setters (en ***Operario***):
getSalario: Sólo retorna el salario del operario.
setAumentarSalario: Sólo permite asignar un nuevo salario al operario. La asignación consiste en incrementar el salario de acuerdo al porcentaje que recibe como parámetro.

2) Constructores: realizar los constructores para cada clase (**Fabrica** y **Operario**) con los parámetros que se detallan en la imagen.
En la clase **Operario**, por cualquier sobrecarga que se utilice, el salario debe inicializarse en $1500. No repetir líneas de código.
En la clase **Fabrica**, por cualquier sobrecarga que se utilice, la cantidad máxima de operarios será de 5. No repetir líneas de código.

3) Métodos (en ***Operario***)
ObtenerNombreYApellido (de instancia): Retorna un ***String*** que tiene concatenado el nombre y el apellido del operario separado por una coma.
Mostrar (de instancia): Retorna un ***String*** con toda la información del operario. Utilizar el método *ObtenerNombreYApellido*.
Mostrar (de clase): Recibe un ***operario*** y retorna un ***String*** con toda la información del mismo (utilizar el método *Mostrar* de instancia)

4) Métodos (en ***Fabrica***)
RetornarCostos (de instancia, privado): Retorna el dinero que la fábrica tiene que gastar en concepto de salario de todos sus operarios.
ObtenerIndice (de instancia, privado): Devuelve el valor del primer índice 'libre' (elemento en ***null***) del Array de Operario, -1 si no encuentra ninguno.
ObtenerIndice (de instancia, privado): Recibe un operario y retorna el índice donde se encuentra dicho operario en el Array, -1 si no está.
MostrarOperarios (de instancia, privado): Recorre el Array de operarios de la fábrica y muestra el nombre, apellido y el salario de cada operario (utilizar el método *Mostrar* de operario).
MostrarCosto (de clase): muestra la cantidad total del costo de la fábrica en concepto de salarios (utilizar el método *RetornarCostos*).

5) Sobrecarga de operadores (en ***Operario***)
Operador == (Igual): Retronará un booleano informando si el nombre, apellido y el legajo de los operarios coinciden al mismo tiempo.

6) Sobrecarga de operadores (en ***Fabrica***)
Operador == (Igual): Retronará un booleano informando si el operario se encuentra en la fábrica o no (utilizar la *sobrecarga ==* de operario).
O**perador +** (Adición): Agrega un operario al Array de tipo ***Operario***, siempre y cuando haya lugar disponible en la fábrica (utilizar el método *ObtenerIndice*) y el operario no se encuentre ya ingresado (utilizar *sobrecarga ==*)
Ej. Utilización:

```
miFabrica = miFabrica + miOperario;
            ó
miFabrica += miOperario;
```

O**perador –** (Sustracción): Saca a un operario de la fábrica, siempre y cuando el operario se encuentre en el Array de tipo Operario.
Ej. Utilización:

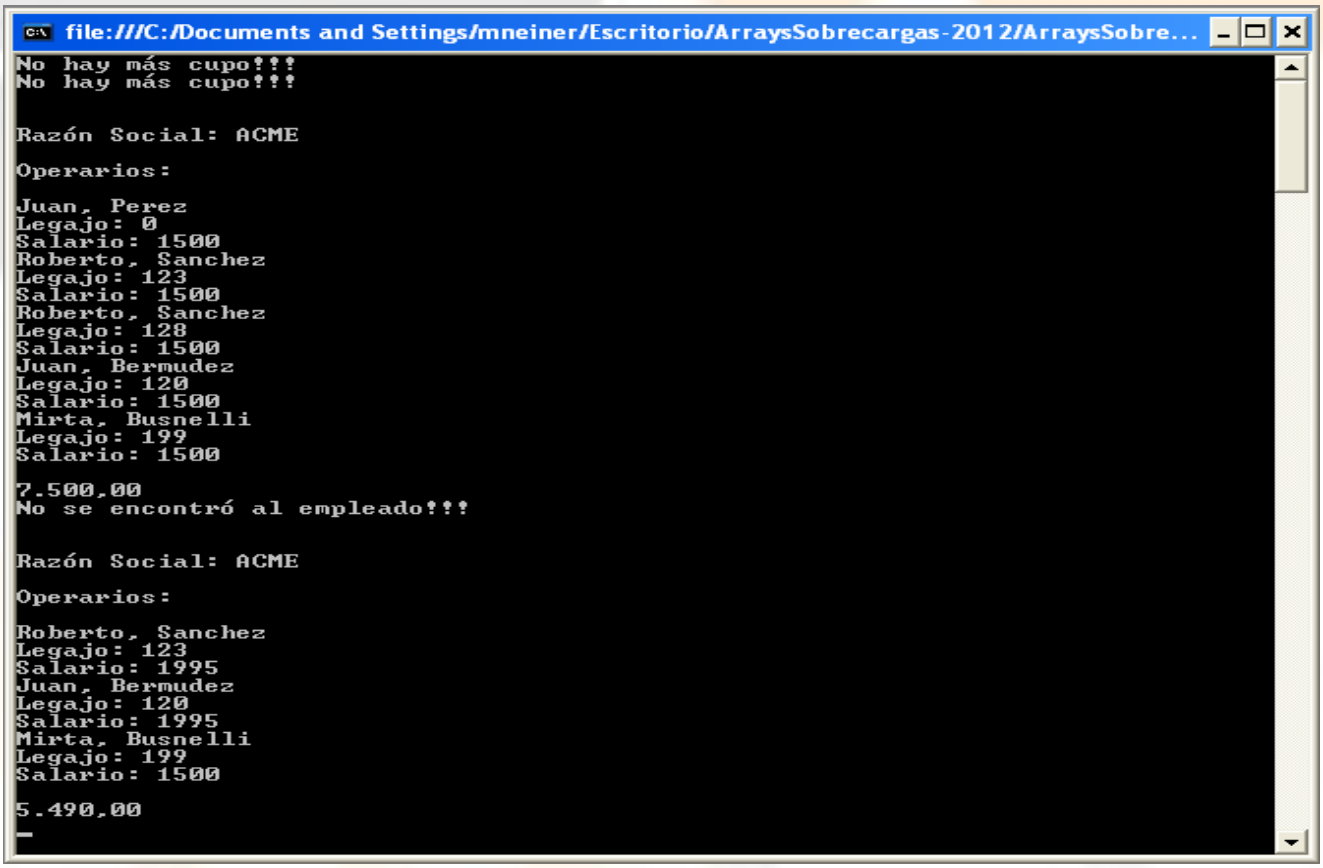
```
miFabrica = miFabrica - miOperario;
            ó
miFabrica -= miOperario;
```

Este es el Main (el alumno lo debe copiar y no se podrá modificar):

```
static void Main(string[] args) {

      Fabrica miFabrica = new Fabrica("ACME");
      Operario op1 = new Operario("Juan", "Perez");
      Operario op2 = new Operario("Roberto", "Sanchez", 123);
      Operario op3 = new Operario("Roberto", "Sanchez", 128);
      Operario op4 = new Operario("Juan", "Bermudez", 120);
```

```
            Operario op5 = new Operario("Mirta", "Busnelli", 199);
            //AGREGO OPERARIOS A LA FABRICA
            miFabrica += op1;
            miFabrica += op2;
            miFabrica += op3;
            miFabrica += op4;
            miFabrica += op5;
            miFabrica += op1;
            miFabrica += op3;
            //MUESTRO LA FABRICA
            Console.WriteLine(miFabrica.Mostrar());
            //MUESTRO EL COSTO
            Fabrica.MostrarCosto(miFabrica);
            //SACO OPERARIOS
            miFabrica -= op1;
            miFabrica -= op3;
            miFabrica -= op1;
            //AUMENTO EL SUELDO A LOS OPERARIOS
            op2.setAumentarSalario(33);
            op4.setAumentarSalario(33);
            //MUESTRO LA FABRICA
            Console.WriteLine(miFabrica.Mostrar());
            //MUESTRO EL COSTO
            Fabrica.MostrarCosto(miFabrica);
            Console.ReadLine();
        }
```

Resultado de consola:



40. ALUMNO – CURSO – con ARRAY (2012)

```
class Alumno {
        private string _apellido;
        private string _nombre;
        private string _dni;
        private int _legajo;
```

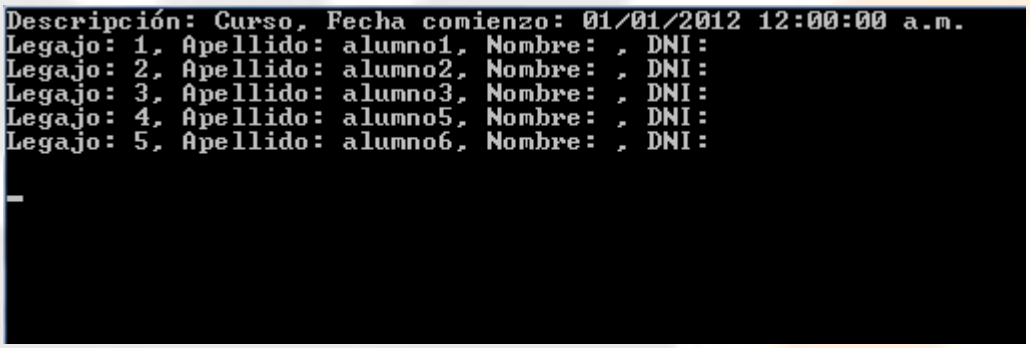
```
        public string Info() {
            return "Legajo: "+ _legajo + ", Apellido: " + _apellido + ",
            Nombre: " + _nombre + ", DNI: " + _dni;
        }
        public Alumno(int legajo, string apellido)  {
            _legajo = legajo;
            _apellido = apellido;
        }
        //Punto 2 (operador igual)...
    }
    class Curso {
        private string _descripcion;
        private Alumno[] _alumnos;
        private DateTime _fechaComienzo;
        public Curso() {
            //...
        }
        //Punto 8 (operador +)...
        private int HayLugar() {
            //...
        }
        private bool ExisteAlumno(Alumno alumno)  {
            //...
        }
        public string Info(){
            //...
        }

    }
```

1. Incluir las clases Alumno y Curso en espacios de nombres diferentes.
2. El método **Info** de la clase Alumno, retornará un string con TODA LA INFORMACIÓN de un alumno.
3. En la clase Alumno, en total deberán existir 3 sobrecargas del constructor, todas recibirán parámetros. No repetir código.
4. Sobrecargar el operador == en la clase Alumno. 2 alumnos serán iguales si tienen igual número de legajo.
5. En la clase Curso, deberán existir 2 sobrecargas del constructor, una de ellas sin parámetros (que sólo inicializará el array (para poder almacenar 5 alumnos como máximo), y será en el único lugar donde se realizará dicha acción). La otra sobrecarga recibirá 2 parámetros (un string y una fecha). No repetir código.
6. En la clase Curso definir un método de instancia llamado HayLugar, retornará un entero, y no recibirá parámetros. Si en el array existe alguna referencia null, retornará la posición en la que se encuentre esa referencia; de lo contrario retornará -1.
7. En la clase curso el método ExisteAlumno retornará verdadero si en la colección existe el alumno que se recibe por parámetro. El criterio de igualdad entre 2 alumnos será el legajo.
8. Sobrecargar el operador + en la clase Curso, para poder sumar un Curso y un Alumno.
Retornará un valor booleano indicando si se agregó el alumno o no. Se agregará un alumno a la colección, siempre y cuando exista un lugar disponible, y el alumno no esté previamente en dicha colección. Utilizar los 2 puntos anteriores.
9. Escribir el código correspondiente en la clase que corresponda para poder realizar la siguiente asignación:
Alumno pepito;
int valor = 100;
pepito = valor; //luego de esta asignación, pepito deberá referenciar a un objeto Alumno con legajo 100.
Un posible Main podría ser el siguiente (no es necesario que utilice necesariamente este ejemplo):

```
    static void Main(string[] args) {
        int valor = 6;
        Alumno alumno1 = new Alumno(1, "alumno1");
        Alumno alumno2 = new Alumno(2, "alumno2");
        Alumno alumno3 = new Alumno(3, "alumno3");
        Alumno alumno4 = new Alumno(1, "alumno4");
        Alumno alumno5 = new Alumno(4, "alumno5");
        Alumno alumno6 = new Alumno(5, "alumno6");
        Alumno alumno7 = new Alumno(6, "alumno7");
        Alumno alumno8 = valor;
```

```
            Curso curso = new Curso("Curso", new DateTime(2012, 1, 1));
            bool retorno;
            retorno = curso + alumno1;
            retorno = curso + alumno2;
            retorno = curso + alumno3;
            retorno = curso + alumno4;
            retorno = curso + alumno5;
            retorno = curso + alumno6;
            retorno = curso + alumno7;
            retorno = curso + alumno8;
            Console.WriteLine(curso.Info());
        }
```


Cuya salida por consola sería:



Observaciones importantes: En caso que fuese necesario, agregar lo que haga falta. Las clases Alumno y Curso al comienzo del enunciado deben ser utilizadas como base. Utilizar propiedades públicas, atributos públicos, o métodos públicos para modificar atributos privados, restará puntos a la nota final del parcial.

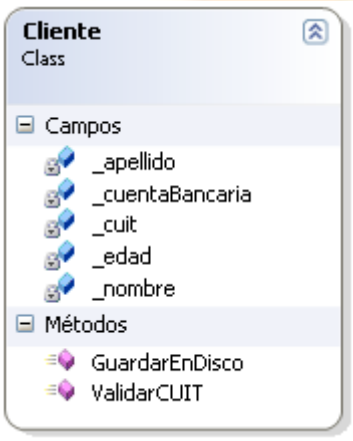
HERENCIA

41. Se requiere crear la clase **SerHumano** que posea los siguientes atributos privados:
    - _nombre (String)
    - _peso (Single)
    - _altura (Single)
    - _sexo (String)

    Además tendrá los siguientes métodos:
    - Comer(String)
    - Dormir

    A partir de la clase SerHumano, se pide:
    Crear dos clases (que hereden de la anterior) llamadas **Gerente** y **Empleado.**
    Cada una de dichas clases poseerá atributos y métodos característicos (por ejemplo, la clase Empleado tendrá como atributo sueldo y como método Trabajar).
    Generar, en el método Main, las sentencias necesarias para probar los miembros de ambas clases.

42. Crear la clase **Cliente** que posea al menos estos cinco atributos privados:



    Todos los atributos deben ser tener **propiedades** de lectura/escritura, salvo _cuentaBancaria que será de sólo lectura.
    Dicha clase, inicializará con su constructor, tanto el número de CUIT como el número de Cuenta Bancaria.
    También tendrá un método estático **ValidarCUIT**, que recibirá el CUIT a validar y devolverá 'True' si el CUIT es válido o 'False' caso contrario. (Sólo se deberá validar que contenga 11 números y dos guiones, ##-########-#).
    Un método ***virtual*** GuardarEnDisco(), que guardará la información del cliente, por defecto se guardará con formato de texto y en el directorio raíz.
    Diseñar la clase ***ClientePlus***, que *herede de Cliente*, que posea el campo privado '_puntos' (lectura/escritura), y que guarde los datos de ese cliente en un archivo de texto diferente que el anterior.

En el método Main se deberán crear distintos tipos de clientes.

43. Crear la clase **Colectivo** que posea: patente, marca, cantidad de asientos. Un método estático **ValidarAsientos**, un método virtual **SetearKm** y un método abstracto **indicarVelocidad**.
Además crear tres clases (CortaDistancia, MediaDistancia y LargaDistancia) que hereden de la primera y que contengan atributos y métodos propios.
Se pide que por medio de un menú el ingreso de pasajeros para los micros, ingresar destinos, ingresar precio de pasajes, calcular mejor recaudación, mostrar datos de los micros.

44. Diseñar un programa que utilice una jerarquía de distintos dispositivos de reproducción de video, creando atributos, métodos, instancias, etc. (Por ejemplo: Vhs, Dvd, OchoMm, etc.) Utilizar los pilares de la **POO**.

SERIALIZACION

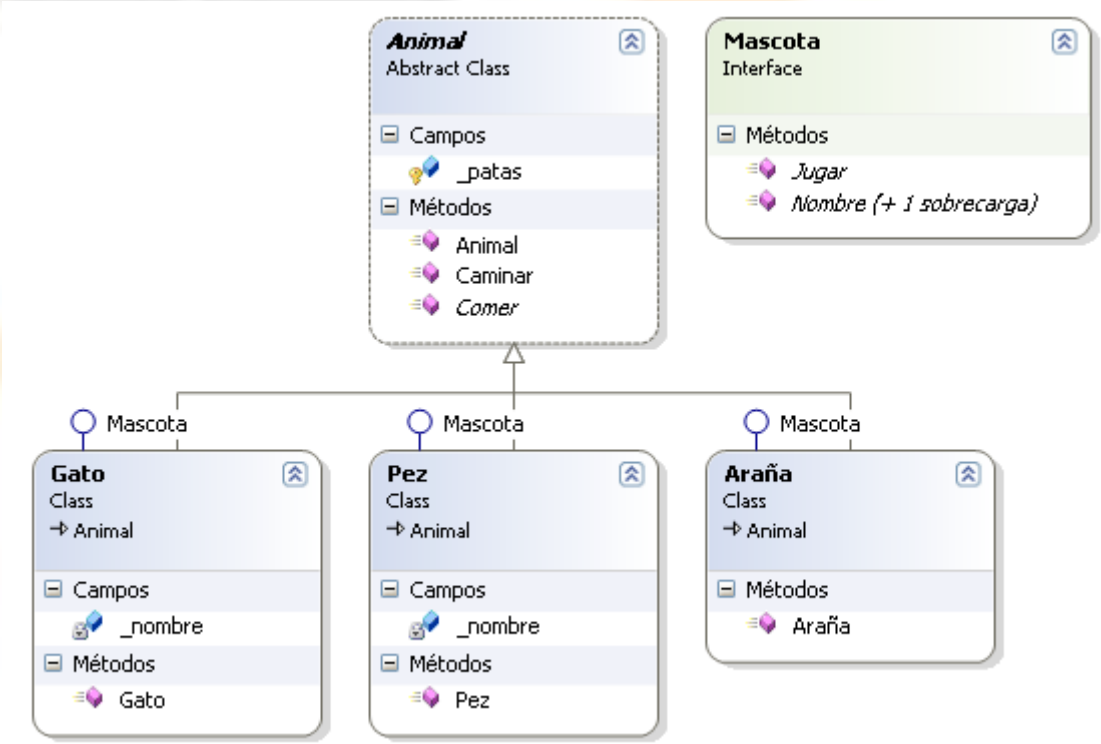
45. Desarrolle una clase llamada **Asignatura** que:
Tenga dos atributos privados, uno de tipo entero (el identificador) y el otro de tipo decimal (la calificación).
Dicha clase tendrá un constructor con un parámetro de tipo entero.
Tomando la clase **SerHumano**, diseñada en el ejercicio 41, se pide: Modificarla para que sea abstracta y 'padre' de las clases **Alumno** y **Profesor**.
La clase Alumno tendrá tres atributos privados de tipo Asignatura, un constructor con tres parámetros de tipo Asignatura que inicialice los tres atributos.
La clase Profesor tendrá un método **CalificarAlumno**, que recibe un parámetro de tipo Alumno y que no devuelve nada. Pondrá una calificación aleatoria a cada una de las asignaturas del alumno, para ello utilizar el método **NextDouble** de la clase **Random**.
Además tendrá un método llamado **CalcularPromedio** que recibe un parámetro de tipo Alumno y devuelve un Double.
Por último, desarrollar una clase llamada Test, que en su método Main:
    - Cree e inicialice tres Asignaturas.
    - Cree un Alumno con las tres Asignaturas.
    - Cree un Profesor que le ponga calificaciones al Alumno y muestre por pantalla el promedio del Alumno.
    - Serializar a XML los objetos creados.

46. Desarrollar un sistema de gestión de garaje siguiendo estas especificaciones:
En el garaje se cambian las ruedas tanto de **coches** como de **motos**.
El precio del cambio de una rueda se fija al abrir el garaje, al igual que la capacidad máxima de **vehículos**, ya sean coches o motos.
Si no hubiese lugar para registrar un nuevo vehículo, habrá que tener contemplado un sistema de aviso para quien este dejando su vehículo.

El sistema de gestión del garaje requiere un mecanismo para ingresar y retirar los vehículos, conocer el número total de vehículos recibidos en ese momento, el precio que supondría cambiar todas las ruedas de todos los vehículos y el kilometraje promedio de todos ellos.
La información que manejaremos de los coches entre otras cosas es la marca y el número de puertas. Mientras que de las motos serán la marca y la cilindrada.
Las clases relacionadas con los vehículos se guardaran en el namespace 'vehículos', mientras que el garaje y la clase que contenga al método Main en el namespace 'GarageTest'.
Dentro del método Main, se creará un garaje y una serie de vehículos que se irán recibiendo en el garaje, y por último se imprimirá por pantalla toda la información general del garaje así como la información de cada vehículo.
Serializar a XML los objetos creados.

INTERFACES

47. En este ejercicio, usted deberá crear clases abstractas e interfaces y explorar las propiedades polimórficas de estos tipos de componentes.
Creará una jerarquía de animales que tienen una clase abstracta Animal como raíz. Varias de las clases de animales implementarán una interfaz llamada Mascota.
En la figura se muestra un diagrama de clases UML de las clases de animales que usted creará.



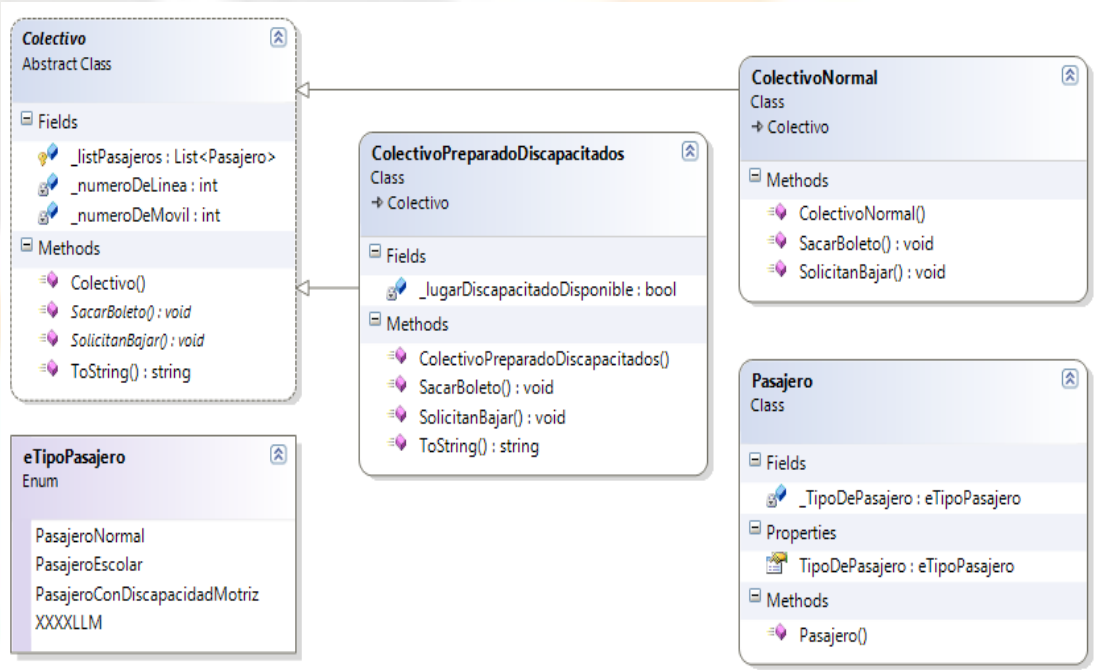
Descripción:
La clase abstracta Animal posee dos métodos, uno 'virtual' llamado 'Caminar()' que deberá mostrar el siguiente mensaje:

"Este animal camina sobre 4 patas", donde 4 será la cantidad de patas que posea el animal. Este método deberá ser sobrescrito en las clases derivadas.
El otro método será abstracto ('Comer()') y deberá describir que es lo que come cada animal.
La interfaz 'Mascota' proporcionará propiedades públicas de lectura/escritura para el nombre de las mascotas y un método llamado 'Jugar()' que mostrará por consola como juega cada mascota.
Serializar a XML los objetos creados.

48. A partir del ejercicio 42, se pide crear una interfaz llamada 'IMostrar' que posea un método llamado 'Mostrar' que, en la clase 'ClientePlus' donde será implementada, muestre la información de ese cliente en la consola.

49. Partiendo de la clase abstracta 'Transporte' (usted definirá los miembros de esta clase), se pide:
    a. Crear dos clases que deriven de esta (Avión y Barco)
    b. Generar una tercera clase llamada 'HidroAvion' que herede de 'Barco' e implemente una interface llamada 'IVolador' que posea los métodos necesarios para poder acelerar, ascender, volar, descender y frenar.
    c. Por último, en el método 'Main', crear instancias de estas clases e interactuar con ellas.
    d. Serializar a XML los objetos creados.

50. Crear las clases que se detallan en el siguiente diagrama:



A continuacion se detallan particularidades de cada una de las clases.

Colectivo

- El constructor inicializa los campos _numeroDeLinea y _numeroDeMovil e instancia _listPasajeros.

- Métodos SacarBoleto y SolicitanBajar, deben ser métodos abstractos y requieren como parámetro un Pasajero.
- Sobre escribir el Método ToString.

ColectivoNormal y ColectivoPreparadoDiscapacitados

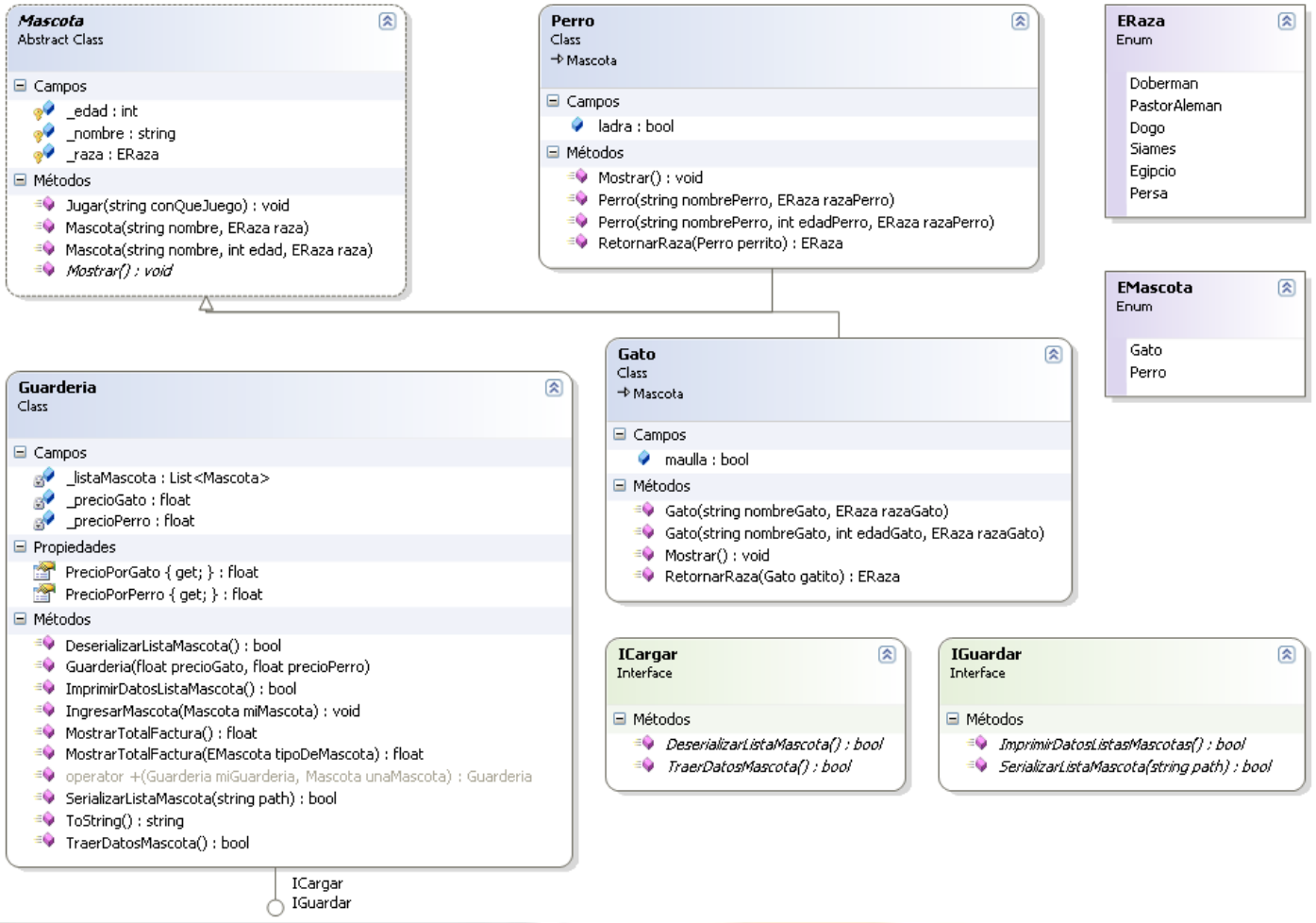
- Deben Heredar de la clase Colectivo.
- El método SacarBoleto agregara a la lista de pasajero (_listPasajeros ya inicializada) el Pasajero enviado como parámetro en caso de que el colectivo sea Normal y el pasajero sea discapacitado motriz deberá lanzar una excepción, indicando que no se puede, en caso que el colectivo lo acepte deberá deshabilitar LugarDiscapacitadosDisponible.
- El método SolicitarBajada en caso que el pasajero sea discapacitado motriz deberá mostrar por pantalla "bajar rampa discapacitados y desciende pasajero" y deberá habilitar LugarDiscapacitadosDisponible, en caso contrario "desciende pasajero".
- El método ToString deberá Mostrar:
  Numero de Línea - Número de Móvil - Cantidad de Pasajeros a Bordo - Lugar Discapacitados Disponible (En caso que corresponda)

Pasajero

- El constructor inicializa los campos _TipoDePasajero con los parámetros solicitados por el mismo (Enum eTipoPasajero, ver grafico).

Ingresar al menos 2 pasajeros por cada uno de los tipos de colectivos posibles (Método SacarBoleto). Solicitar bajada de pasajero al menos una vez (SolicitarBajada). Llamar al Método ToString de los dos colectivos.

51. Realizar la siguiente jerarquía de clases:



La clase Mascota posee un método virtual (Jugar) y otro abstracto (Mostrar). Las clases Perro y Gato deberán implementar el método Mostrar para poder visualizar desde consola todos sus atributos. De la clase Perro **no** se podrá generar herencia.
Deberán tener un método **de Clase** que reciba un PERRO o un GATO, según corresponda, y retorne un enumerado con la raza al que pertenece la mascota.
La clase Guardería que tendrá un atributo **Privado** de tipo lista genérica de Mascotas y además dos atributos **Privados** más: _precioPerro (Double) y _precioGato (Double), estos se inicializarán desde el constructor.
Realizar propiedades de sólo lectura para los dos atributos.
Se realizará la sobrecarga del operador "+" para permitir sumar una mascota a la guardería, retornando una guardería con la mascota agregada en la lista genérica, ejemplo:

Miguarderia = Miguarderia + UnaMascota

Los métodos públicos que tendrá la Guardería son:

- MostrarTotalFacturado: devolverá la ganancia de la guardería (Single), dicho método tendrá una sobrecarga que reciba como parámetro la enumeración EMascota (con Perro y Gato como enumerados) y retornará la ganancia de la Guardería por **tipo** de Mascota.

- IngresarMascota: (que recibe como único parámetro una Mascota), agregará a la lista dicho objeto.
- ToString: deberá devolver en un String los datos de todas las mascotas que tiene la Guardería, este método utilizará un objeto de tipo StringBuilder.

La interface "IGuardar" contiene los siguientes métodos:

- ImprimirDatosListaMascotas
  - Este método guardará sin sobrescribir los datos de las mascotas que tengo en la guardería.
- SerializarListaMascota
  - Serializa la lista de mascotas

La interface "ICargar" contiene los siguientes métodos:

- TraerDatosMascota
  - Este método leerá de un archivo de texto los datos de las mascotas que están en él.
    Lo guardará en un StringBuilder los mostrará por consola.
- DeserealizarListaMascota
  - Deserializa la lista de mascotas

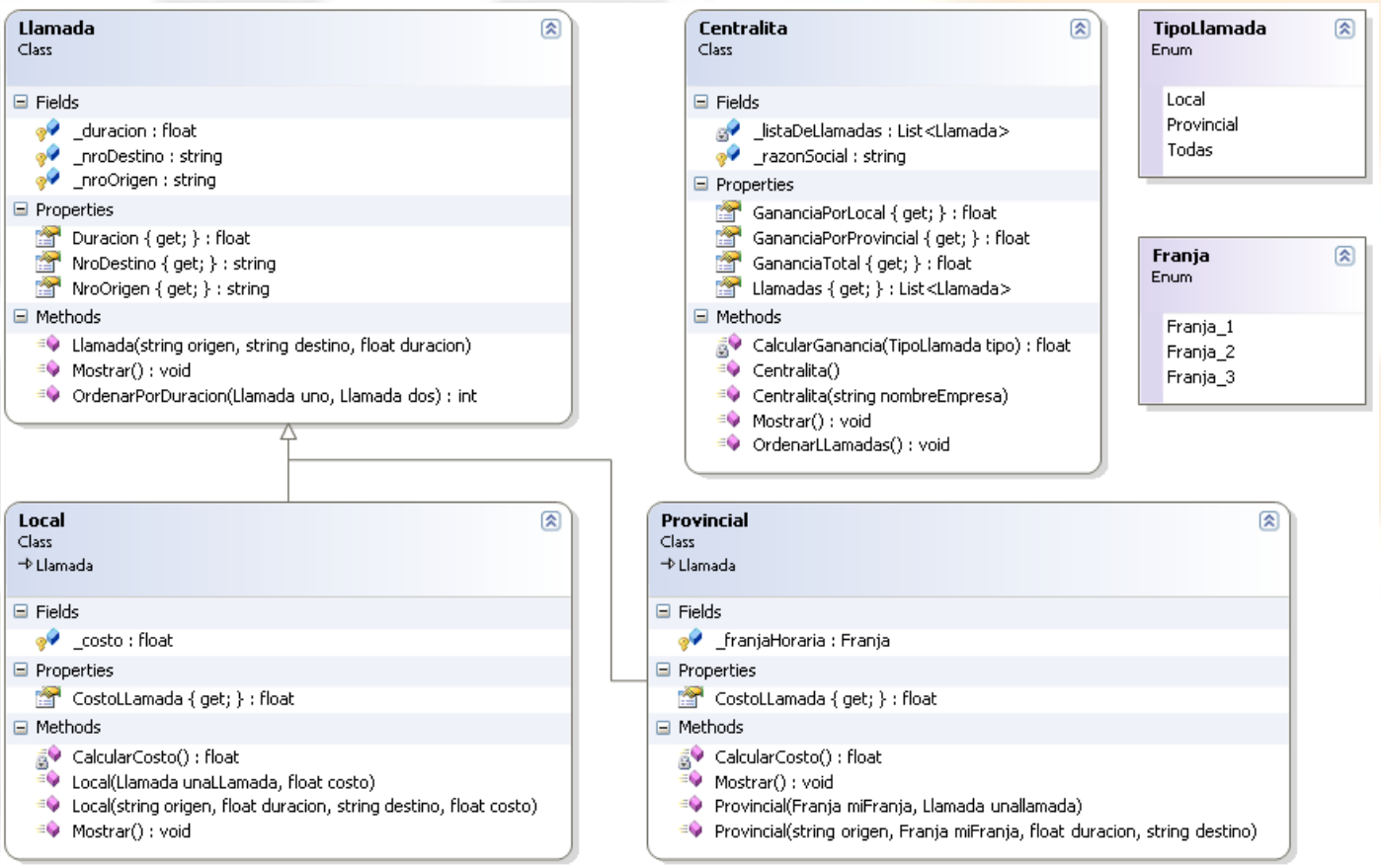
En el Main:

```
// Crear tres gatos
// Crear tres perros
// Crear un Guardería
// Ingresar las mascotas a la guardería
// Mostrar el total Facturado
// Mostrar el total Facturado por Gato
// Mostrar el total Facturado por Perro
// Guarderia.ImprimirDatosListaMascotas
// Guarderia.SerializarListaMascotas
// Guarderia.DeserializarListaMascotas
// Guarderia.TraerDatosListaMascotas

//MOSTRAR POR CONSOLA TODOS LOS ATRIBUTOS DE LAS MASCOTAS
//INGRESADAS EN LA GUARDERIA
```

CENTRALITA

52.Centralita – Herencia

1) Crear en una solución llamada "**CentralTelefonica**" un proyecto de tipo **Consola** nombrado como "**CentralitaHerencia**" que contenga la siguiente jerarquía de clases:



Esta aplicación servirá de control de llamadas realizadas en una central telefónica.

2) La **clase Llamada,** tendrá todos sus atributos con el modificador ***protected***.

Nota: *Realizar propiedades de* ***sólo lectura*** *para cada atributo de la clase* ***Llamada.***

- Métodos
  - OrdenarPorDuracion (método de clase, recibe dos Llamadas).
  - Mostrar (método de instancia, utiliza StringBuilder).
  - Constructor.

Nota: *El método de clase* ***OrdenarPorDuracion*** *será utilizado en el método* ***Sort*** *de la lista genérica de objetos del mismo tipo (en la clase* ***Centralita****).*

Las clases **Local** y **Provincial**, heredarán de ***Llamada*** y poseerán los siguientes miembros:

Local

- Atributos (protegidos)
  - _costo (float)
- Métodos
  - Mostrar (Mostrará, además de los atributos de la clase base, sólo la propiedad ***CostoLlamada***, utiliza StringBuilder).
  - ***CalcularCosto*** *(Privado. Retornará el valor de la llamada a partir de la duración y el costo de la misma).*

Nota: *La propiedad* ***CostoLlamada*** *retornará el precio, que se calculará en el método CalcularCosto().*

Provincial

- Atributos (protegidos)
  - _franjaHoraria (Enumerado de tipo **Franja**).
- Métodos
  - Mostrar (Mostrará, además de los atributos de la clase base, _franjaHoraria y ***CostoLlamada***, utiliza StringBuilder).
  - ***CalcularCosto*** (Privado. Retornará el valor de la llamada a partir de la duración y el costo de la misma. Los valores serán: Franja_1: 0.99, Franja_2: 1.25 y Franja_3: 0.66)

Dentro del este mismo proyecto se deberá agregar el enumerado **TipoLlamada** (Local, Provincial y Todas) que será utilizado posteriormente.

Centralita

- Atributos (privados y protegidos)
  - _listaDeLlamadas (List<Llamada>) (privado)
  - _razonSocial (String) (protegido)
- Métodos
  - Mostrar (Mostrará la razón social, la ganancia total, ganancia por llamados locales y provinciales y el detalle de las llamadas realizadas).

- ***CalcularGanancia*** *(Privado. Recibe un Enumerado de tipo* ***TipoLlamada*** *y retornará el valor de lo recaudado, según el criterio elegido).*
- Constructor (Recibirá la razón social y luego inicializará la lista genérica, en el constructor por defecto)

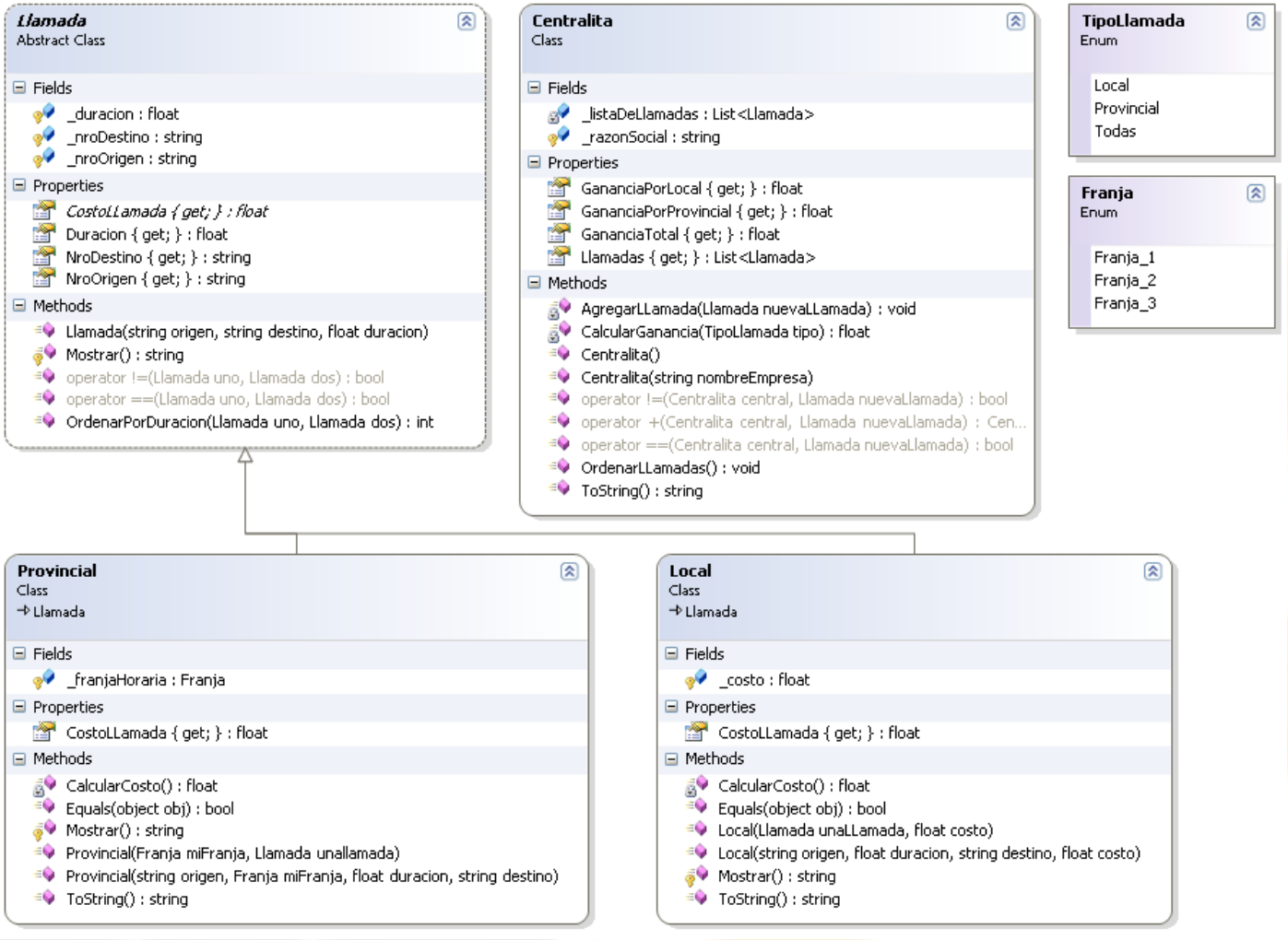
Nota: *Las propiedades* ***GananciaPorTotal, GananciaPorLocal y GananciaPorProvincial*** *retornarán el precio de lo facturado según el criterio. Se calculará en el método* **CalcularGanancia***().*

3) En el Main() se creará una ***Centralita*** ("Telefónica") y cuatro llamadas. La primera llamada será de tipo **Local,** con una duración de 30 seg. y un costo de 2.65. La segunda será **Provincial** (en la franja 1 y con duración de 21 seg.) y las restantes: **Local** (45 seg. y 1.99) y Provincial (que recibe la segunda llamada y franja 3).

4) Las llamadas se irán registrando en la ***Centralita***. Registrar una llamada consiste en agregar a la lista genérica de tipo ***Llamada*** una llamada **Provincial** o una llamada **Local**.
La centralita mostrará por pantalla todas las llamadas según las vaya registrando (método ***Mostrar***).
Luego, se ordenarán las llamadas (método ***OrdenarLlamadas***) y se volverá a mostrar por pantalla el contenido de la Centralita.

53.Centralita – Clase abstracta - polimorfismo

1) Agregar un nuevo proyecto de tipo **Consola** nombrado como "**CentralitaPolimorfismo**", a la solución "**CentralTelefonica**".
Partiendo del proyecto anterior (CentralitaHerencia) modificar la jerarquía de clases para obtener:



2) La clase **Llamada** ahora será ***abstracta***. Tendrá definida la propiedad de sólo lectura **CostoLlamada** que también será ***abstracta***.

   - Métodos
     - Mostrar deberá ser declarado como ***virtual***, protegido y retornará un string que contendrá los atributos propios de la clase ***Llamada*** (utiliza un StringBuilder).

   - Sobrecarga de operadores
     - == Comparará dos llamadas y retornará ***true*** si las llamadas son del mismo tipo (utilizar método ***Equals***, sobrescrito en las clases derivadas) y los números de destino y origen son iguales en ambas llamadas.

3) La clase **Local**

   - Métodos sobrescritos
     - ***Mostrar*** (Protegido. Reutilizará el código escrito en la clase base y además agregará la propiedad ***CostoLlamada***, utilizando un StringBuilder).

- ***Equals*** *(Retornará* **true,** *sólo si el objeto que recibe es de tipo* ***Local****).*
- ***ToString*** (Utilizará al método ***Mostrar***)

4) La clase **Provincial**

- Métodos sobrescritos
  - Mostrar (Protegido. Reutilizará el código escrito en la clase base y agregará _franjaHoraria y ***CostoLlamada***, utilizando un StringBuilder).
  - ***Equals*** *(Retornará* **true,** *sólo si el objeto que recibe es de tipo* ***Provincial****).*
  - ***ToString*** (Utilizará al método ***Mostrar***)

5) **Centralita**

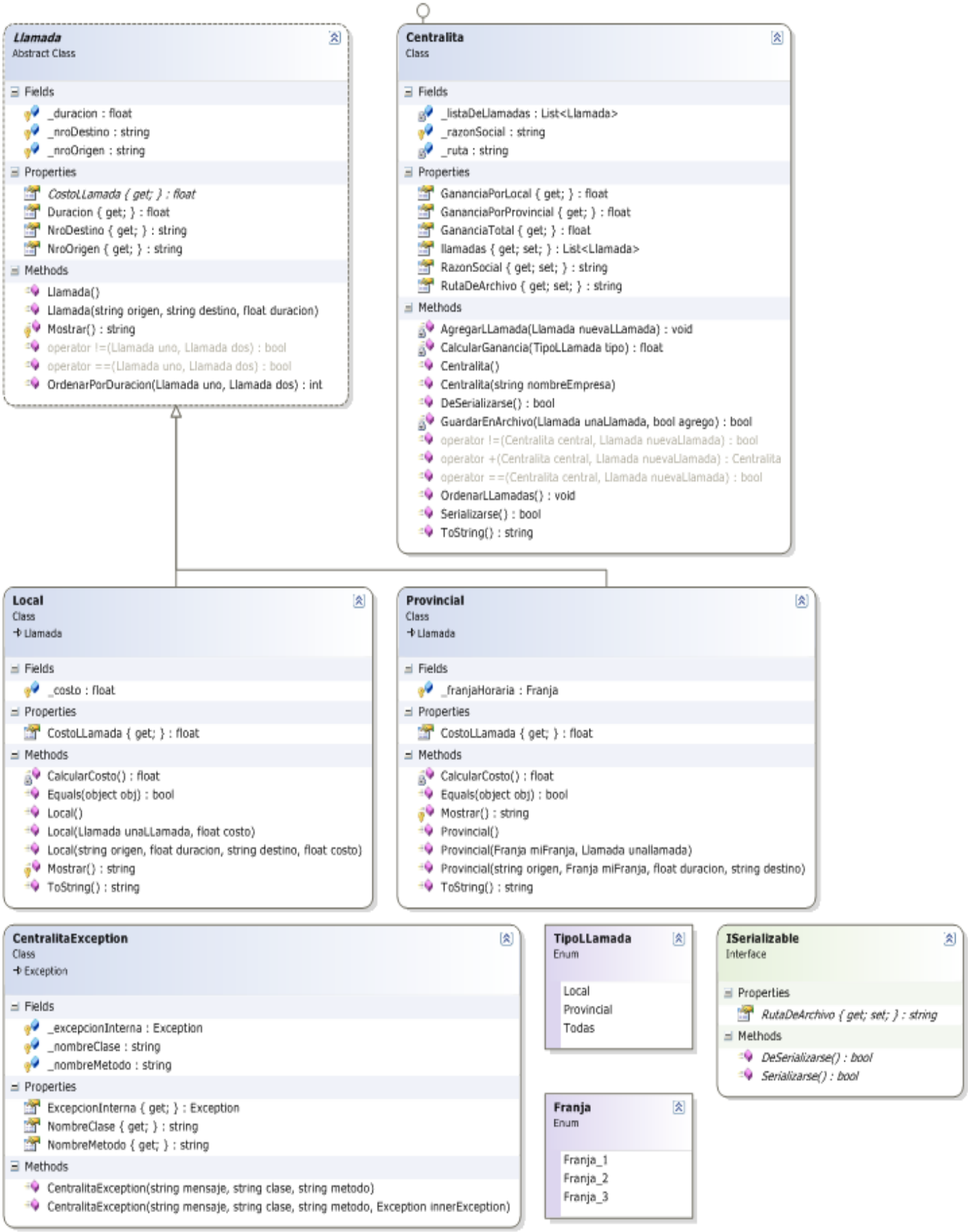
- Métodos
  - Se reemplaza el método Mostrar por la sobrescritura del método ***ToString****.*
  - ***AgregarLlamada*** *(Privado. Recibe una* ***Llamada*** *y la agrega a la lista de llamadas.*

- Sobrecarga de operadores
  - == Retornará ***true*** si la ***Centralita*** contiene la ***Llamada*** en su lista genérica (utilizar sobrecarga ***==*** de Llamada).
  - + Invocará al método AgregarLlamada sólo si la llamada no está registrada en la Centralita, caso contrario mostrará un mensaje (utilizar la sobrecarga del operador == de Centralita).

6) En el Main() se creará una ***Centralita*** ("Telefónica") y cuatro llamadas. La primera llamada será de tipo **Local,** con una duración de 30 seg. y un costo de 2.65. La segunda será **Provincial** (en la franja 1 y con duración de 21 seg.) y las restantes: **Local** (45 seg. y 1.99) y Provincial (que recibe la segunda llamada y franja 3).

7) Las llamadas se irán registrando en la ***Centralita***. Utilizar sobrecarga de operadores. Intentar ingresar llamadas duplicadas.La centralita mostrará por pantalla todas las llamadas según las vaya registrando (ToString).Luego, se ordenarán las llamadas (método ***OrdenarLlamadas***) y se volverá a mostrar por pantalla el contenido de la Centralita.

54.Centralita – Serialización – Excepciones



1) Agregar un nuevo proyecto de tipo **Consola** nombrado como **"CentralitaSerializacion"**, a la solución **"CentralTelefonica"**. Partiendo del proyecto anterior (CentralitaPolimorfismo)

modificar la jerarquía de clases para obtener el anterior diagrama.

2) La clase **Llamada**, **Provincial** y **Local**, además de tener un constructor por defecto, deberán estar marcadas con el atributo ***Serializable*** (para poder serializarlas).

3) **Centralita**, ahora implementará la interface ***ISerializable***.

- Métodos
  - **Serializarse**. Retornará true, si es que se pudo serializar el objeto actual, incluyendo el listado de llamadas del mismo, en el ***path*** indicado por el atributo ***privado*** **_ruta**. En el caso de no poder serializarlo (debido a una excepción) se deberá lanzar una excepción de tipo ***CentralitaException***, indicándole:
    - Mensaje de error.
    - Nombre de la clase que provocó la excepción.
    - Nombre del método que provoco la excepción.
    - El objeto de tipo *Exception*.
  - **DeSerializarse**. Retornará true, si es que se pudo deserializar del archivo .XML al objeto actual, del ***path*** indicado por el atributo ***privado*** **_ruta**. En el caso de no poder deserializarlo (debido a una excepción) se deberá lanzar una excepción de tipo ***CentralitaException***, indicándole:
    - Mensaje de error.
    - Nombre de la clase que provocó la excepción.
    - Nombre del método que provoco la excepción.
    - El objeto de tipo *Exception*.
  - **GuardarEnArchivo**. Este método privado, guardará en un ***archivo de texto*** nombrado como **Llamadas.txt** (ubicado en el *path* indicado por el atributo _ruta).
    En el archivo se agregará/sobrescribirá (según el atributo ***agrego***) un encabezado, la fecha de la llamada (con formato de tiempo largo) y la descripción de la llamada (utilizar polimorfismo).
    Si por algún motivo no se pudiera escribir en el archivo se deberá lanzar una excepción (de tipo CentralitaException) informando:
    - Mensaje de error.
    - Nombre de la clase que provocó la excepción.
    - Nombre del método que provoco la excepción.
    - El objeto de tipo *Exception*.

El tratamiento de esta excepción se realizará en el método privado **AgregarLlamada** (quién será el encargado de invocar al método GuardarEnArchivo).

- Propiedades
  - **RutaDelArchivo.** Permitirá obtener y establecer el path desde dónde se realizará la ***serialización/deserialización*** del objeto 'centralita'.

- Sobrecarga de Operadores
  - **Operador +.** En dicha sobrecarga, si la llamada que se quiere agregar ya fue registrada en la centralita, se deberá lanzar una excepción de tipo ***CentralitaException***, indicándole:
    - Mensaje de error.
    - Nombre de la clase que provocó la excepción.
    - Nombre del método que provoco la excepción.

4) **CentralitaException.** Esta clase deriva de la clase **Exception.**

5) En el Main() se creará una ***Centralita*** ("Telefónica") y cuatro llamadas. La primera llamada será de tipo **Local,** con una duración de 30 seg. y un costo de 2.65. La segunda será **Provincial** (en la franja 1 y con duración de 21 seg.) y las restantes: **Local** (45 seg. y 1.99) y Provincial (que recibe la segunda llamada y franja 3).

6) Una vez instanciada la centralita, se le deberá pasar a la propiedad ***RutaDelArchivo*** el directorio actual de la aplicación concatenado con el nombre de archivo ***Centralita.XML***.

7) Intentar deserializar la centralita informando por consola el resultado de dicha operación.

8) Las llamadas se irán registrando en la ***Centralita***. Utilizar sobrecarga de operadores. Intentar ingresar llamadas duplicadas. La centralita mostrará por pantalla todas las llamadas según las vaya registrando (ToString).
   Luego, se ordenarán las llamadas (método ***OrdenarLlamadas***) y se volverá a mostrar por pantalla el contenido de la Centralita.

9) Antes de finalizar con la aplicación se deberá serializar el contenido de la centralita. Informar del resultado de dicha operación por consola.

10) Recuperar, si es que existe, el archivo **Llamadas.txt** y mostrar su contenido por consola.

55. VEHICULO – LAVADERO (2008)

Realizar la clase abstracta Vehiculo que posea como atributos protegidos:

- patente (string - sólo lectura)
- cantRuedas (Byte)
- marca (enum Marcas, con los siguientes enumerados: Honda, Ford, Zanella y Fiat)

Y los siguientes métodos:

- void Mostrar (abstracto)
- void AcelerarHasta(Byte) (virtual)
- Vehiculo(string, Byte, Marcas) (sin sobrecargas)

Además se pide:
Crear dos clases (Auto y Moto) que hereden de Vehiculo y que posean cantidadAsientos (int) y Cilindrada (int) como atributos respectivamente. Cada una de estas clases deberá implementar el método Mostrar para poder visualizar desde consola todos sus atributos.

Por último se desea construir la clase Lavadero que tendrá una lista genérica de Vehiculos, un atributo cantidadVehiculos (int), precioAuto (float) y precioMoto (float), que se inicializaran desde su constructor.

Los métodos que tendrá Lavadero son:

- MostrarTotalFacturado: devolverá la ganancia del lavadero (Double), dicho método tendrá una sobrecarga que reciba como parámetro la enumeración Vehiculos (con Auto y Moto como enumerados) y retornará la ganancia del Lavadero por tipo de vehículo.
- El método IngresarAlLavadero (que recibe como único parámetro un Vehiculo), agregará a la lista dicho objeto e incrementara su contador de vehículos en uno.

```
static void Main(string[] args)
{
    Auto auto1 = new Auto("ABC123", Marca.Ford, 4, 6);
    Auto auto2 = new Auto("DEF456", Marca.Fiat, 4, 4);
    Moto moto1 = new Moto("GHI789", Marca.Zanella, 2, 1300);
    Moto moto2 = new Moto("JKL000", Marca.Honda, 2, 3600);

    Lavadero miLavadero = new Lavadero(12.5, 9.8);

    //INGRESAR LOS AUTOS Y LAS MOTOS

    double totalPorAuto = 0, totalPorMoto = 0, totalVehiculos = 0;

    totalPorAuto = miLavadero.MostrarTotalFacturado(Vehiculos.Auto);
    totalPorMoto = miLavadero.MostrarTotalFacturado(Vehiculos.Moto);
    totalVehiculos = miLavadero.MostrarTotalFacturado();

    Console.WriteLine("Facturado por Autos: {0}",totalPorAuto);
    Console.WriteLine("Facturado por Motos: {0}",totalPorMoto);
    Console.WriteLine("Facturado por Vehiculos: {0}",totalVehiculos);
    Console.WriteLine("Cantidad total de Vehiculos: {0}",
    miLavadero.cantidadVehiculos);

    Console.WriteLine("Listado de vehiculos");

    //MOSTRAR TODOS LOS ATRIBUTOS DE LOS VEHICULOS INGRESADOS AL LAVADERO
}
```

56. FACTURA – ITEMS (2009)

Una Empresa quiere desarrollar una parte de su sistema de facturación con el siguiente diseño:

Factura

Campos (protegidos)	Propiedad	Acceso	Métodos	Modificador
Numero(long) Fecha(DateTime)			Constructor() void CalcularTotal()	 Abstact
DatosImprenta (string)			String Mostrar()	Virtual

FacturaC : Factura

Campos (privados)	Propiedad	Acceso	Métodos	
Items (<Item>) Total (Double) (……)			Constructor() void CalcularTotal() void AgregarItem() String Mostrar()	 Override Override

Item

Campos (privados)	Propiedad	Acceso	Métodos
Codigo(int)	Codigo	lectura	Constructor()
Cantidad(int)	Cantidad	lectura	String MostrarDetalles()
Detalle(string)	Detalle	lectura	
Precio (double)	Precio	lectura	

El procedimiento es el siguiente: Se crea un objeto FacturaC que por lo menos tendrá dos ítems distintos (se crearán dos objetos de tipo Item distintos).

A la factura se le agregarán los dos ítems (a la lista genérica Items<>), utilizando el método AgregarItem(), pasándole los parámetros que usted crea conveniente.
El método Mostrar() expondrá los datos de la factura, asimismo, el método MostrarDetalles() mostrará todos los datos del ítem en cuestión.
El método CalcularTotal(), calculará el costo de todos los ítems de la factura (precio * cantidad).

Nota: Los métodos Mostrar() y MostrarDetalles() deben utilizar el objeto 'StringBuilder' ubicado en el namespace 'System.Text'.

57. TRANSPORTE (2010)

Se deberá crear un proyecto nombrado con el apellido y nombre del alumno separado por un punto (ej: si el alumno se llama Juan Pérez, el proyecto deberá llamarse Pérez.Juan)

1.- Crear una clase llamada "**Transporte**" que posea los siguientes atributos que **sólo** deberán ser visibles desde las clases que hereden de ella.
_línea (entero)
_interno (entero)
_puertaCerrada (booleano)
_chofer (string)
Y los siguientes métodos:

void IngresarChofer(string) que reciba el nombre del chofer de la unidad y lo guarde en el atributo que corresponda. Este método será **abstracto**.
void SacarBoleto() que indique por consola que el boleto se saca arriba del colectivo. Este método será **virtual** y **deberá ser sobrescrito**.
bool CerrarPuerta() que devolverá TRUE sólo si la puerta se encuentra cerrada. En caso contrario deberá cerrarla.

2.- Crear una interface IServicio que posea el método **void ServirComida (string)** que deberá recibir la comida a servir que mostrará por consola.

3.- Crear una clase llamada "**Corta_Distancia**" que herede de "Transporte" y que posea los siguientes atributos:
_pasajerosABordo (entero): indica la cantidad de pasajeros en el colectivo.

4.- Crear una clase llamada "**Larga_Distancia**" que herede de "Transporte", que implemente "IServicio" y que posea el siguiente atributo:
_personalABordo (entero): cantidad de personal de la empresa a bordo.

5.- Crear constructores para las clases que permitan inicializar todos los atributos.

6.- En el Main:
a) Crear 3 objetos de la clase "Corta_Distancia" y agregarlos a una lista de Transporte.
b) Crear 4 objetos de la clase "Larga_Distancia" y agregarlos a una lista de Transporte. (La misma del punto anterior)
c) Se parar los objetos según su clase en dos listas (utilizar un foreach), una de tipo Corta_Distancia y la otra de Larga_Distancia.
d) Escribir en un archivo de texto todos los atributos de todos los objetos de la lista de corta distancia (Sobrescribir el ToString()).
e) Serializar la lista de larga distancia en un archivo XML.

ACLARACIÓN: todos los métodos y propiedades deberán estar documentados, los atributos y métodos nombrados correctamente respetando tanto el nombre como la forma. Se deberán gestionar los errores.

58. BARCO (2010)

Crear una clase llamada "**Barco**" que posea los siguientes atributos:
(**Sólo** deberán ser visibles desde las clases que hereden de ella).

- _cantMaxPasajeros (entero)
- _motorEncendido (booleano)
- _destino (string)

Realizar una propiedad **abstracta de sólo lectura**:

```
List<Pasajero> Pasajeros
```

Y los siguientes métodos:

void Ingresar(Pasajero) que reciba un objeto de tipo **Pasajero** y lo guarde en el atributo que corresponda, siempre y cuando la cantidad máxima de pasajeros no sea superada, en tal caso, lanzar una excepción de tipo **PasajerosOverFlowException** (que usted deberá crear), la cual será debidamente tratada en el Main. Este método será **abstracto**.

string IndicarDestino() que retorne el valor que hay en el atributo **_destino** para ser mostrado por consola. Este método será **virtual** y **deberá ser sobrescrito**.

bool EncenderMotor() que devolverá TRUE sólo si el motor se encuentra encendido. En caso contrario deberá encenderlo.

2.- Crear una interface **IServicioComedor** que posea el método:
void ServirComida (string) el cual deberá recibir la comida a servir. Se mostrará por consola.

3.- Crear la clase **Pasajero** con los atributos: Apellido (String), Nombre y **Rango** (enumerado Rangos {Camarero, Cocinero, Capitan, Cliente}). Los modificadores de visibilidad se dejarán a su elección.

4.- Crear una clase llamada "**Crucero**" que herede de "Barco" y que posea el siguiente atributo:
_pasajeros (List<Pasajero>): lista genérica de tipo Pasajero. Asociar este atributo con la propiedad Pasajeros.

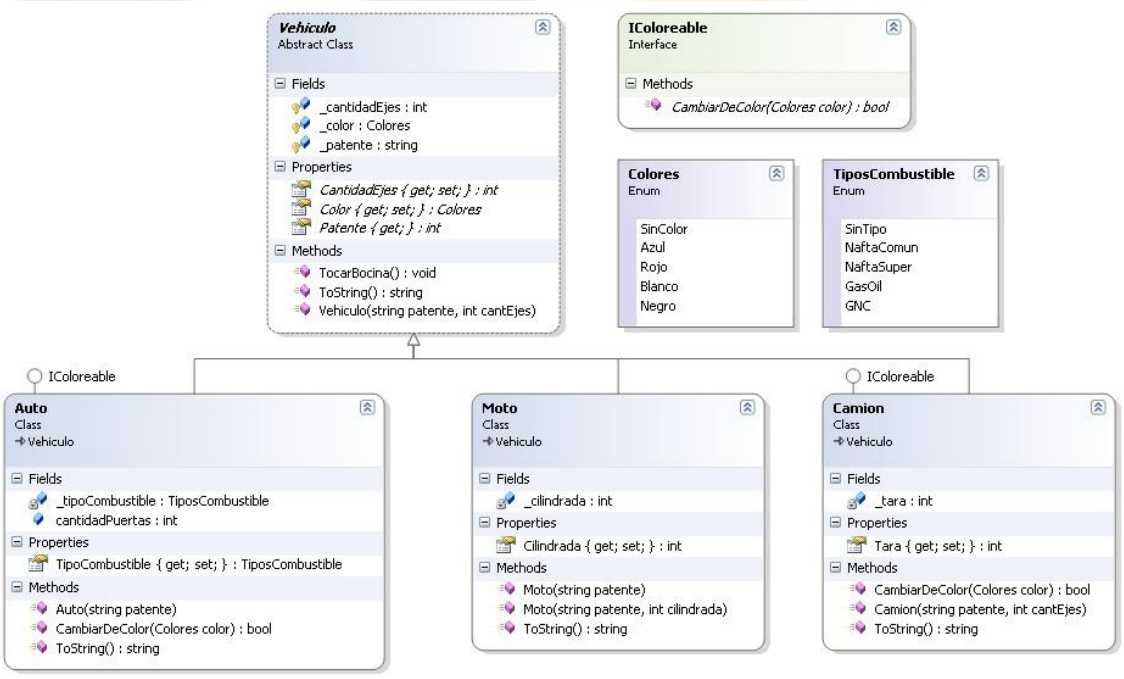
5.- Crear una clase llamada "**Transatlántico**" que herede de "Barco", que implemente "IServicioComedor" y que posea el siguiente atributo:
_pasajeros (List<Pasajero>): lista genérica de tipo Pasajero. Asociar este atributo con la propiedad Pasajeros.

6.- Realizar constructores para permitir inicializar el destino y la cantidad máxima de pasajeros (en Barco y sus derivadas) y los nombres, apellidos y rango (en Pasajero).

7.- En el Main:
a) Crear un crucero y un transatlántico, ambos con una capacidad máxima de 3 pasajeros y como destinos 'Brasil' y 'Australia' respectivamente.
b) Crear 3 objetos de la clase "Pasajero" y agregarlos al crucero.
c) Crear 4 objetos más de la clase "Pasajero" y agregarlos al transatlántico.
d) Al capturar una excepción de tipo **PasajerosOverFlowException** se escribirá en un archivo de texto la fecha y hora de la excepción. Indicando que tipo de barco provocó la excepción.
e) Serializar el transatlántico en un archivo XML. Modifique lo que crea necesario para tal fin.
f) Mostrar los destinos y el listado de pasajeros de ambos barcos. Sobrescribir el método ToString().

ACLARACIÓN: todos los métodos y propiedades deberán estar documentados, los atributos y métodos nombrados correctamente respetando tanto el nombre como la forma. Se deberán gestionar los errores.

59. VEHICULO (2010)



1. Crear una solución nombrada como ***SegundoParcial*** en una carpeta nombrada con su apellido seguido de su nombre (Por ejemplo, PerezJuan).
   Escribir la jerarquía de clases del diagrama anterior, creando un archivo **.cs** por cada clase y/o interface.
2. A excepción de los constructores, el resto de los métodos contendrá sólo un mensaje de salida por consola que indique que método se está ejecutando.
   Asignar valores a los atributos que no los reciban por medio del constructor o de algún otro método.
3. El método CambiarColor() retornará un valor booleano verdadero si el color que recibe como parámetro es distinto al color que ya posee el vehículo y distinto al enumerado SinColor, en tal caso, cambiará el valor del atributo. Caso contrario retornará un valor falso.
4. Desarrollar el Main en una clase que cuyo nombre sea su apellido. Instanciar un objeto de cada clase en las que sea posible hacerlo. Si alguna clase tiene constructores sobrecargados, realizar una instancia por cada constructor.
5. Cada objeto creado se guardará en una lista genérica de tipo **Vehiculo** que se creará en el Main.
6. Las clases que utilicen el método CambiarDeColor() mostrarán los atributos antes de la llamada al método y después de dicha llamada mostrarán el atributo que si se ha modificado, caso contrario mostrar un mensaje informando de la situación.
7. La propiedad TipoCombustible, que recibe el valor del enumerado TiposCombustible, deberá lanzar una excepción si el valor recibido es SinTipo.
8. Al capturar la excepción, se deberá mostrar y guardar en un archivo de texto (nombrado como ***Incidencias.log***) el mensaje de la misma.
9. La excepción lanzada la deberá crear usted y la debe nombrar como **CombustibleException**. Dicha clase poseerá sólo una propiedad llamada MensajeExcepcion, que contendrá la descripción de la excepción.
10. Crear los métodos necesarios en la clase que contiene al Main para serializar y deserializar en XML la lista genérica de tipo **Vehiculo**. Los nombres sugeridos para estos métodos son: SerializarVehiculos y DeserializarVehiculos respectivamente.

Nota: *La clase Vehiculo posee las propiedades **abstractas** CantidadEjes (L/E), Color (L/E) y Patente (L) y un método **virtual** TocarBocina. Las clases Auto y Camion implementan la interface IColoreable.*

60. EXPENDEDORA – BOLETOS (2011)

1 Dadas la clases:

Expendedora

_marca (atrib. string)
_fechaFabricacion (atrib. DateTime)
_transacciones (atrib. lista de transacciones, agregar prop. sólo lectura)
Estado (prop. abstracta, eEstado)

ExpendedoraBoletos

TienePapel (prop. sólo lectura bool)
_mmPapelDisponible (atrib. Int)

ExpendedoraGaseosas

_stock (int)

Transaccion

_fecha (DateTime)
_importe (decimal)

eEstado: enumerado, valores: Disponible, NoDisponible

En la clase Expendedora: Definir un constructor que reciba 2 parámetros (marca y fecha de fabricación).

Definir un método de instancia llamado DatosExpendedora (virtual) para que retorne un string con la marca, fecha de fabricación, transacciones y estado de una expendedora

En las clases ExpendedoraBoletos y ExpendedoraGaseosas (ambas heredan de Expendedora):

Sobrescribir el método ToString() para que retorne un string con información de la expendedora, más la información propia de cada clase.

Tanto el atributo _mmPapelDisponible como _stock, se inicializarán sólo en el constructor.

En la clase Transaccion: sobreescribir el método ToString() para que retorne un string con la fecha e importe de una transacción; definir un constructor que reciba 2 parámetros

2 Definir una interfaz llamada IOperacion que se implementará en las clases ExpendedoraBoletos y ExpendedoraGaseosas, y que tendrá el sig. método:

ImprimirTicket (retorna un bool, recibe un valor de tipo decimal). Mostrará por consola fecha e importe de la transacción, y agregará una transacción a la lista de transacciones, con la fecha actual e importe que recibe por parámetro (en ExpendedoraGaseosas disminuirá el stock en una unidad, en ExpendedoraBoletos disminuirá el atributo _mmPapelDisponible en 15 unidades). Estas acciones se harán siempre y cuando se cumpla lo sig.: en ExpendedoraBoletos, tener estado Disponible y tener papel; y en ExpendedoraGaseosas deberá tener estado Disponible y stock > 0. En el caso que no se cumplan las condiciones antes mencionadas, se deberá cambiar el estado a NoDisponible y arrojar una excepción de tipo ExcepcionAlImprimirTicket, deberá tener en su propiedad Message un mensaje que indique esta situación.

3 En el main, crear una expendedora de cada tipo, imprimir 5 tickets para cada una. En caso que ocurra una excepción de tipo ExcepcionAlImprimirTicket, se deberá guardar un archivo de texto fecha, hora, y el valor que tenga la propiedad Message de la excepción. Nota: si previamente existe el archivo, no se deberá perder la información contenida.

4 Luego crear una lista que almacene objetos de tipo Expendedora, agregar las expendedoras que se hayan creado, y serializar la lista en formato xml

61. TREN (2011)

Se deberá crear un proyecto nombrado con el apellido y nombre del alumno separado por un punto (ej.: si el alumno se llama Juan Pérez, el proyecto deberá llamarse **Pérez.Juan**)

1.- Crear una clase llamada "**Tren**" que posea los siguientes atributos:
(**Sólo** deberán ser visibles desde la propia clase y las clases que hereden de ella).
_cantMaxPasajeros (entero)
_motorEncendido (booleano)
_destino (string)
Realizar una propiedad **abstracta de sólo lectura**:
List<Pasajero> Pasajeros
Y los siguientes métodos:

void Ingresar(Pasajero) que reciba un objeto de tipo **Pasajero** y lo guarde en el atributo que corresponda, siempre y cuando la cantidad máxima de pasajeros no sea superada, en tal caso, lanzar una excepción de tipo **PasajerosOverFlowException** (que usted deberá crear), la cual será debidamente tratada en el Main. Este método será **abstracto**.

string IndicarDestino() que retorne el valor que hay en el atributo **_destino** para ser mostrado por consola. Este método será **virtual** y **deberá ser sobrescrito**.

bool EncenderMotor() que devolverá TRUE sólo si el motor se encuentra encendido. En caso contrario deberá encenderlo.

Polimorfismo en el método **ToString()**. Utilizar el objeto **StringBuilder**.

2.- Crear una interface **IServicioComedor** que posea el método:

void ServirComida (string) el cual deberá recibir la comida a servir, para posteriormente mostrarla por consola.

3.- Crear la clase **Pasajero** con los atributos: Apellido (String), Nombre y **Rango** (enumerado Rangos {Azafata, Cocinero, Maquinista, Cliente}). Los modificadores de visibilidad se dejarán a su elección. Polimorfismo en el método **ToString()**. Utilizar el objeto **StringBuilder**.

4.- Crear una clase llamada "**TrenElectrico**" que herede de "Tren" y que posea el siguiente atributo:

_pasajeros (List<Pasajero>): lista genérica de ***tipo Pasajero***. Asociar este atributo con la propiedad ***Pasajeros***.

Polimorfismo en el método **ToString()**. Utilizar el objeto **StringBuilder**.

5.- Crear una clase llamada "**TrenBala**" que herede de "Tren", que implemente "IServicioComedor" y que posea el siguiente atributo:

_pasajeros (List<Pasajero>): lista genérica de ***tipo Pasajero***. Asociar este atributo con la propiedad ***Pasajeros***.

Polimorfismo en el método **ToString()**. Utilizar el objeto **StringBuilder**.

6.- Realizar constructores para permitir inicializar el destino y la cantidad máxima de pasajeros (en Tren y sus derivadas) y los nombres, apellidos y rango (en Pasajero).

7.- En el Main:

a) Crear un tren eléctrico y un tren bala, ambos con una capacidad máxima de 3 pasajeros y como destinos 'Burzaco' y 'Pinamar' respectivamente.

b) Crear 3 objetos de la clase "Pasajero" y agregarlos al tren eléctrico.

c) Crear 4 objetos más de la clase "Pasajero" y agregarlos al tren bala.

d) Al capturar una excepción de tipo **PasajerosOverFlowException** se escribirá en un **archivo de texto** la fecha y hora de la excepción.

e) Agregar el tren eléctrico y el tren bala a una lista genérica de tipo "Tren". Serializar la lista en un archivo XML. **Agregue** lo que crea necesario para tal fin.

f) Deserializar la lista de trenes, mostrarla por consola y escribirlas en un archivo de texto, nombrado como ***MisTrenes.txt***.

ACLARACIÓN: Los atributos y métodos deberán estar nombrados correctamente, respetando tanto el nombre como la firma. Se deberán gestionar los errores.