

Antes de empezar lea lo siguiente:

Para los ejercicios de esta guía, 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

1. Ingresar 5 números por consola, guardándolos en una variable escalar. Luego calcular y mostrar: el valor máximo, el valor mínimo y el promedio.
2. 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.
3. 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 (%).
4. 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 (%).
5. 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 (%).
6. 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 (%).

7. 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.

8. 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.

9. 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.

MÉTODOS ESTÁTICOS (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 valor, int min, int max)**:

- valor: dato a validar
- min y max: rango en el cual deberá estar la variable valor.

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 "¿Continuar? (S/N)".

En el método estático **ValidaS_N(char c)** de la clase **ValidarRespuesta**, se validará el ingreso de

opciones.

El método devolverá un valor de tipo booleano, *TRUE* si se ingresó una 'S' y *FALSE* si se ingresó cualquier otro valor.

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**) que realicen el cálculo del área que corresponda:

a. **double CalcularCuadrado(double)**

b. **double CalcularTriangulo(double, double)**

c. **double CalcularCirculo(double)**

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 cuándo finalizar el programa.

Crear una clase llamada **Calculadora** que posea tres métodos estáticos (de clase):

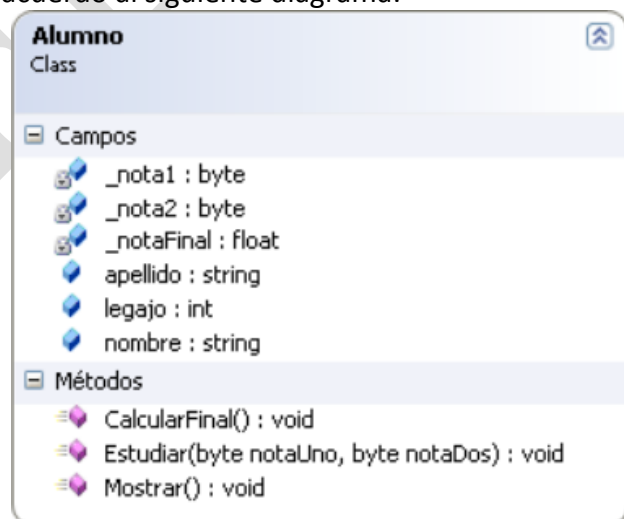
a. **Calcular** (público): Recibirá tres parámetros, el primer número, el segundo número y la operación matemática. El método devolverá el resultado de la operación.

b. **Validar** (privado): Recibirá como parámetro el segundo número. Este método se debe utilizar sólo cuando la operación elegida sea la DIVISIÓN. Este método devolverá *TRUE* si el número es distinto de CERO.

c. **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:



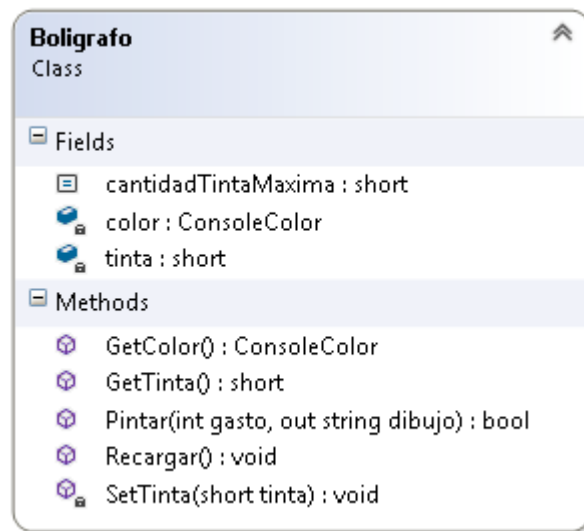
a. Se pide crear 3 instancias de la clase Alumno (3 objetos) en la función Main. Colocarle nombre, apellido y legajo a cada uno de ellos.

b. Sólo se podrá ingresar las notas (nota1 y nota2) a través del método **Estudiar**.

c. 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**.

- d. El método **Mostrar**, expondrá en la consola todos los datos de los alumnos. La nota final se mostrará sólo si es distinto de -1, caso contrario se mostrará la leyenda "Alumno desaprobado".

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

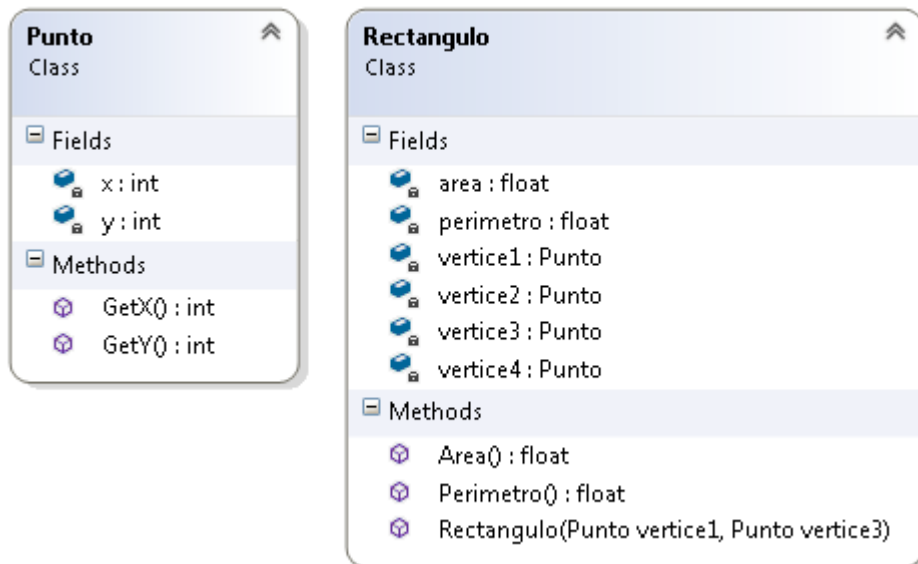


- La cantidad máxima de tinta para todos los bolígrafos será de 100. Generar una constante en el Bolígrafo llamada `cantidadTintaMaxima` donde se guardará dicho valor.
- Generar los métodos ***GetColor*** y ***GetTinta*** para los correspondientes atributos (sólo retornarán el valor del mismo).
- Generar el método privado `SetTinta` que valide el nivel de tinta y asigne en el atributo.
- `Recargar()` colocará a tinta en su nivel máximo de tinta. Reutilizar código.
- En el Main, crear un bolígrafo de tinta azul (***ConsoleColor.Blue***) y una cantidad inicial de tinta de 100 y otro de tinta roja (***ConsoleColor.Red***) y 50 de tinta.
- El método `Pintar(int, out string)` restará la tinta gastada, sin poder quedar el nivel en negativo, avisando si pudo pintar (nivel de tinta mayor a 0). También informará mediante el out string tantos "*" como haya podido "gastar" del nivel de tinta. O sea, si nivel de tinta es 10 y gasto es 2 valdrá "***" y si nivel de tinta es 3 y gasto es 10 "*****".
- Utilizar todos los métodos en el Main.
- Al utilizar `Pintar`, si corresponde, se deberá dibujar por pantalla con el color de dicho bolígrafo.

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

18. 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).
- Realizar los métodos *getters* para los atributos privados lado, área y perímetro.
- Los atributos área ($base * altura$) y perímetro ($((base + altura)*2)$) se deberán calcular sólo una vez, al llamar por primera vez a su correspondiente método *getter*. Luego se deberá retornar siempre el mismo valor.

En el espacio de nombres **PruebaGeometria** es donde se escribirá una clase con el método **Main**.

- Probar las funcionalidades de las clases Punto y Rectangulo.
 - Generar un nuevo Rectangulo.
 - Imprimir por pantalla los valores de área y perímetro.
- Desarrollar un método de **clase** que muestre todos los datos del rectángulo que recibe como parámetro.

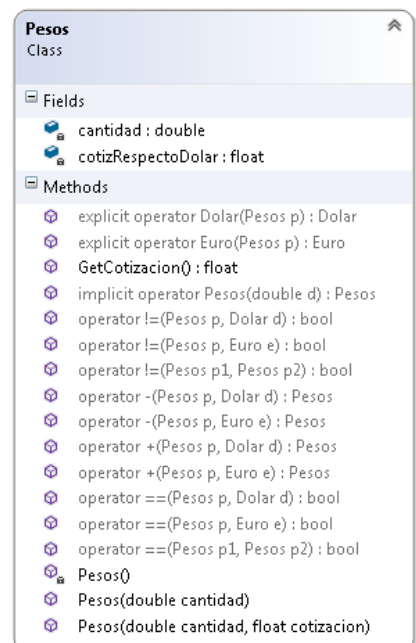
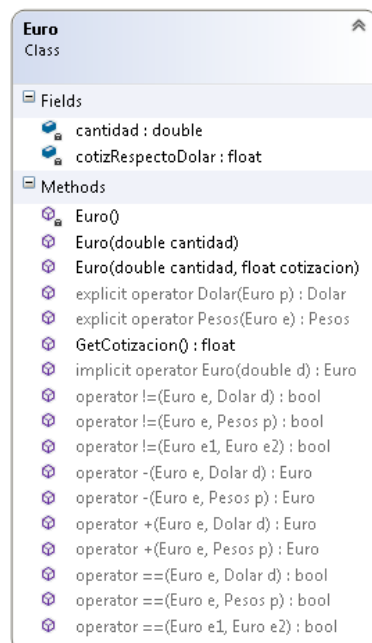
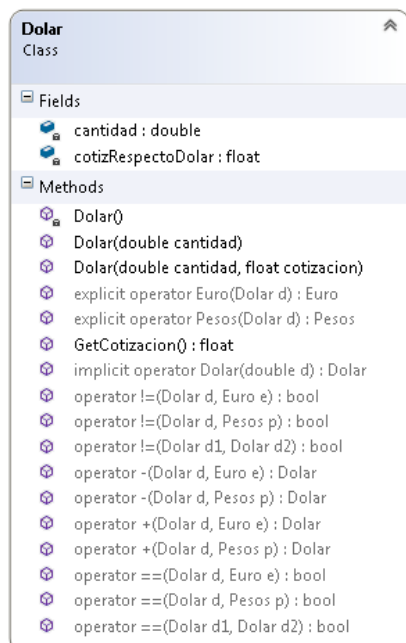
SOBRECARGA DE OPERADORES

19. Realizar una aplicación de consola. Agregar la clase Sumador.



- a. Crear dos constructores:
 - i. Sumador(int) inicializa cantidadSumas en el valor recibido por parámetros.
 - ii. Sumador() inicializa cantidadSumas en 0. Reutilizará al primer constructor.
 - b. El método Sumar incrementará cantidadSumas en 1 y adicionará sus parámetros con la siguiente lógica:
 - i. En el caso de Sumar(long, long) sumará los valores numéricos
 - ii. En el de Sumar(string, string) concatenará las cadenas de texto.
- Antes de continuar, agregar un objeto del tipo Sumador en el Main y probar el código.
- c. Generar una conversión explícita que retorne cantidadSumas.
 - d. Sobrecargar el operador + (suma) para que puedan sumar cantidadSumas y retornen un long con dicho valor.
 - e. Sobrecargar el operador | (pipe) para que retorne True si ambos sumadores tienen igual cantidadSumas.
- Agregar un segundo objeto del tipo Sumador en el Main y probar el código.

20. Generar un nuevo proyecto del tipo Console Application. Construir tres clases dentro de un namespace llamado Billetes: **Pesos, Euro y Dolar**.



- a. Se debe lograr que los objetos de estas clases se puedan sumar, restar y comparar entre sí con total normalidad como si fueran tipos numéricos, teniendo presente que 1 Euro equivale a 1,3642 dólares y 1 dólar equivale a 17,55 pesos.
- b. Sobrecargar los operadores **explicit** y/o **implicit** para lograr compatibilidad entre los distintos tipos de datos.
- c. Colocar dentro del Main el código necesario para probar todos los métodos.

21. Crear tres clases: **Fahrenheit**, **Celsius** y **Kelvin**. Realizar un ejercicio similar al anterior, teniendo en cuenta que:

$$F = C * (9/5) + 32$$

$$C = (F-32) * 5/9$$

$$F = K * 9/5 - 459.67$$

$$K = (F + 459.67) * 5/9$$

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

Agregar las clases:

a. **NumeroBinario**:

- i. único atributo número (string)
- ii. único constructor privado (recibe un parámetro de tipo string)

b. **NumeroDecimal**

- i. único atributo número (double)
- ii. único constructor privado (recibe un parámetro de tipo double)

Utilizando los métodos de la clase **Conversor** donde corresponda, agregar las sobrecargas de operadores:

c. **NumeroBinario**:

- i. string + (NumeroBinario, NumeroDecimal)
- ii. string - (NumeroBinario, NumeroDecimal)
- iii. bool == (NumeroBinario, NumeroDecimal)
- iv. bool != (NumeroBinario, NumeroDecimal)

d. **NumeroDecimal**:

- i. double + (NumeroDecimal, NumeroBinario)
- ii. double - (NumeroDecimal, NumeroBinario)
- iii. bool == (NumeroDecimal, NumeroBinario)
- iv. bool != (NumeroDecimal, NumeroBinario)

Agregar conversiones **implícitas** para poder ejecutar:

- e. NumeroBinario objBinario = "1001";
- f. NumeroDecimal objDecimal = 9;

Agregar conversiones **explícitas** para poder ejecutar:

- g. (string)objBinario
- h. (double)objDecimal

Generar el código en el Main para instanciar un objeto de cada tipo y operar entre ellos, imprimiendo cada resultado por pantalla.

23. Tomar el Ejercicio 20. Crear una Biblioteca de Clases llamada Moneda y colocar dentro las clases Pesos, Euro y Dólar. Crear un proyecto de Windows Form donde se realizará un formulario con el siguiente formato:



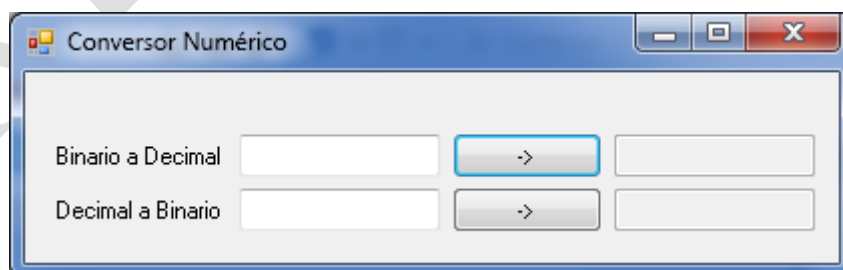
Implementarlo de tal forma que al ingresar un valor válido en la primer casilla (txtEuro, txtDolar y txtPesos respectivamente) y presionar el botón del medio (btnConvertEuro, btnConvertDolar y btnConvertPesos) el resultado de la conversión se vea reflejado en las casillas de la derecha (txtEuroAEuro, txtEuroADolar, txtEuroAPesos, txtDolarAEuro, txtDolarADolar, txtDolarAPesos, txtPesosAEuro, txtPesosADolar y txtPesosAPesos), las cuales no podrán ser editadas/escritas por el usuario.

24. Tomar el Ejercicio 21. Realizar un Formulario con el siguiente formato:



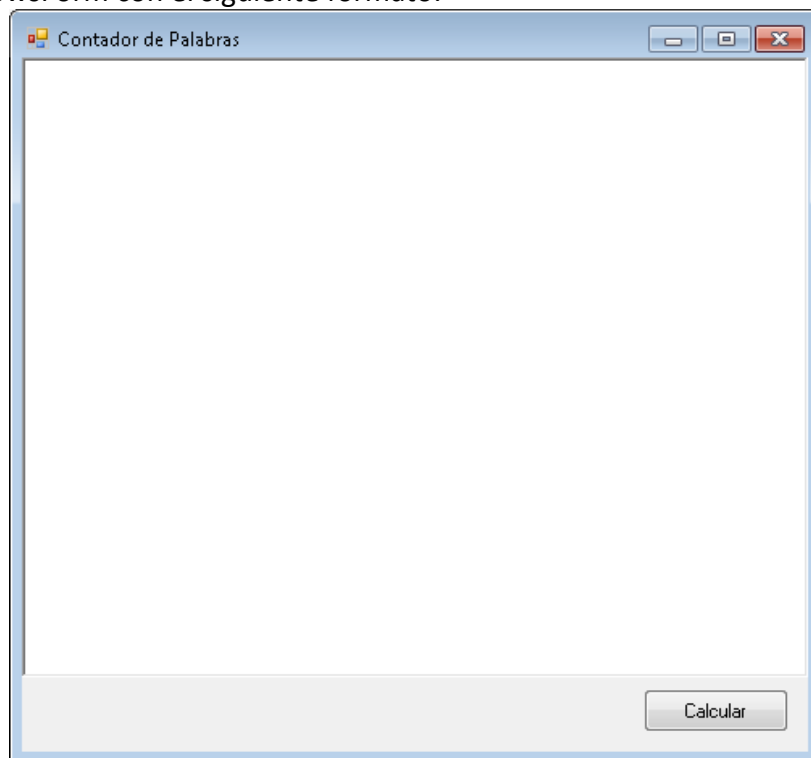
Implementarlo con la misma lógica que el ejercicio anterior.

25. Tomar el ejercicio 22. Realizar un Formulario con el siguiente formato:



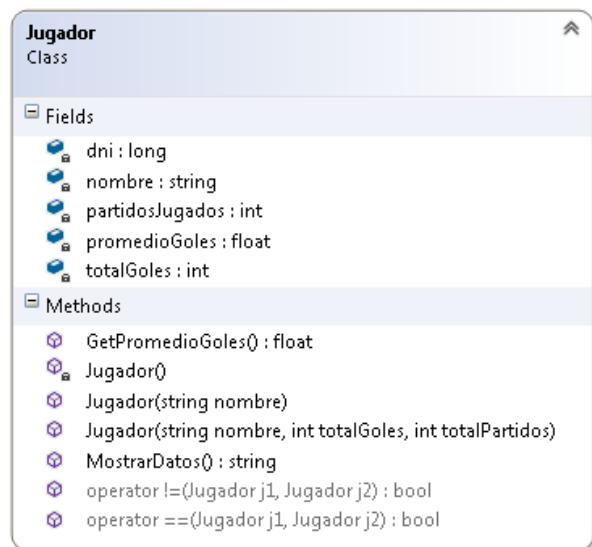
Implementarlo de tal forma que al ingresar un valor válido en la primer casilla (txtBinario y txtDecimal respectivamente) y presionar el botón del medio (btnBinToDec y btnDecToBin) el resultado de la conversión se vea reflejado en las casillas de la derecha (txtResultadoDec y ResultadoBin), las cuales no podrán ser editadas/escritas por el usuario.

26. Crear una aplicación de consola que cargue 20 números enteros (positivos y negativos) distintos de cero de forma aleatoria utilizando la clase Random.
- Mostrar el vector tal como fue ingresado
 - Luego mostrar los positivos ordenados en forma decreciente.
 - Por último, mostrar los negativos ordenados en forma creciente.
27. Realizar el ejercicio anterior pero esta vez con las siguientes colecciones: **Pilas**, **Colas** y **Listas**.
28. Generar un WindowsForm con el siguiente formato:



Utilizar Diccionarios (Dictionary<string, int>) para realizar un contador de palabras, recorriendo el texto palabra por palabra se deberá lograr que si se trata de una nueva palabra, se la agregará al diccionario e inicializar su contador en 1, caso contrario se deberá incrementar dicho contador. Ordenar los resultados de forma descendente por cantidad de apariciones de cada palabra. Informar mediante un MessageBox el TOP 3 de palabras con más apariciones.

29. Crear una Clase llamada **Jugador** y otra Equipo con la siguiente estructura:



Jugador:

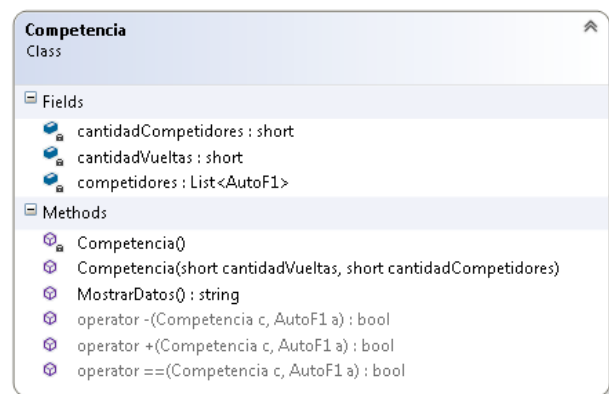
- Todos los datos estadísticos del Jugador se inicializarán en 0 dentro del constructor privado.
- El promedio de gol sólo se calculará cuando invoquen al método GetPromedioGoles.
- MostrarDatos retornará una cadena de string con todos los datos y estadística del Jugador.
- Dos jugadores serán iguales si tienen el mismo DNI.

Equipo:

- La lista de jugadores se inicializará sólo en el constructor privado de Equipo.
- La sobrecarga del operador + agregará jugadores a la lista siempre y cuando este no exista aun en el equipo y la cantidad de jugadores no supere el límite establecido por el atributo cantidadDeJugadores.

Generar los métodos en el Main para probar el código.

30. Diseñar una clase llamada **Competencia** y otra **AutoF1** con los siguientes atributos y métodos:



AutoF1:

- Al generar un auto se cargará enCompetencia como falso y cantidadCombustible y vueltasRestantes en 0.
- Dos autos serán iguales si el número y escudería son iguales.
- Realizar los métodos getters y setters para cantidadCombustible, enCompetencia y vueltasRestantes.

Competencia:

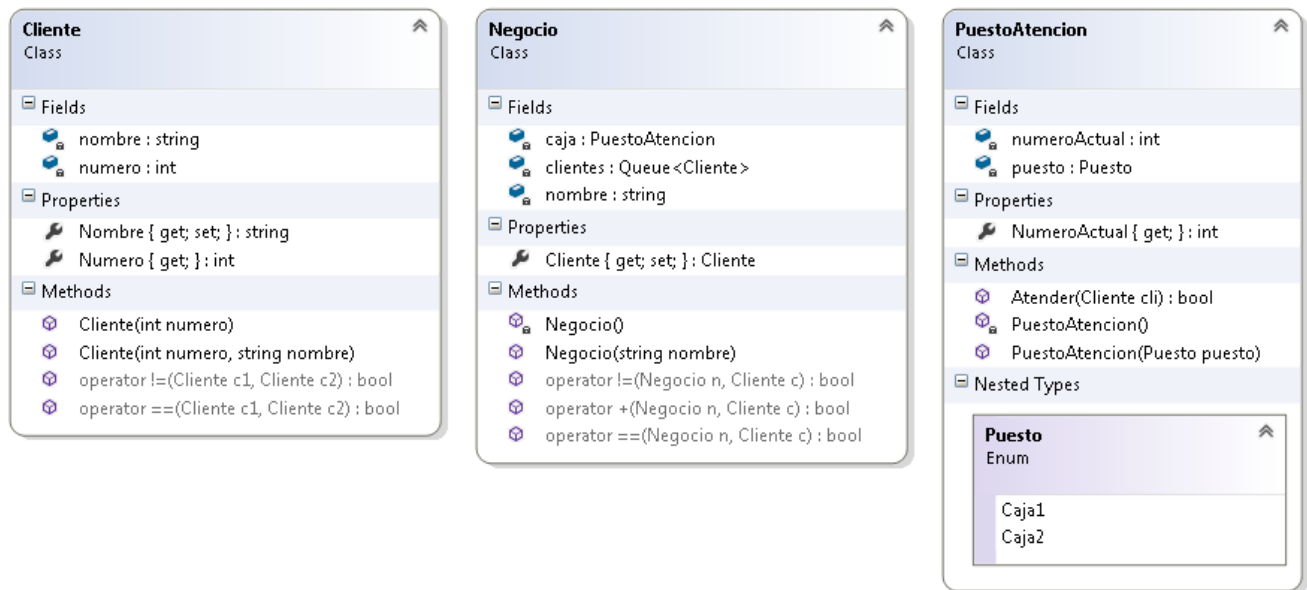
- El constructor privado será el único capaz de inicializar la lista de competidores.

- e. La sobrecarga + agregará un competidor si es que aún hay espacio (validar con cantidadCompetidores) y el competidor no forma parte de la lista (== entre Competencia y AutoF1).
- f. Al ser agregado, el competidor cambiará su estado enCompetencia a verdadero, la cantidad de vueltasRestantes será igual a la cantidadVueltas de Competencia y se le asignará un número random entre 15 y 100 a cantidadCombustible.

Generar los métodos en el Main para probar el código.

ENCAPSULAMIENTO

31. Generar un sistema de atención al cliente mediante las clases Cliente, Negocio y PuestoAtencion:



Puesto Atención:

- a. numeroActual es estático y privado. Se inicializará en el constructor de clase con valor 0.
- b. El método Atender simulará un tiempo de atención a un cliente: recibirá un cliente, simulará un tiempo de atención mediante el método de clase Sleep de la clase Thread y retornará true para indicar el final de la atención.
- c. NumeroActual es una **propiedad** estática, encargada de incrementar en 1 al atributo numeroActual y luego retornarlo.

Cliente:

- d. Dos clientes serán iguales si tienen el mismo número.

Negocio:

- e. El constructor privado inicializará la colección y el puesto de atención como Caja1.
- f. El operador + será el único capaz de agregar un Cliente a la cola de atención. Sólo se agregará un Cliente si este ya no forma parte de la lista. Reutilizar el == de Cliente.
- g. La propiedad Cliente retornará el próximo cliente en la cola de atención en el get. El set deberá controlar que el Cliente no figure ya en la cola de atención, caso contrario lo agregará.
- h. El operador == retornará true si el cliente se encuentra en la colección.
- i. El operador bool ~(Negocio) generará una atención del próximo cliente en la cola, utilizando la propiedad Cliente y el método Atender de PuestoAtencion.

Generar los métodos en el Main para probar el código.

32. Tomar el ejercicio 29 como base.

- Agregar propiedades de sólo lectura a los atributos partidosJugados, promedioGoles y totalGoles de Jugador, y de lectura/escritura a nombre y dni.
- Quitar el atributo promedioGoles de jugador. Calcular dicho promedio dentro de la propiedad de sólo lectura PromedioDeGoles.
- Quitar el método GetPromedioGoles, colocando dicha lógica a la respectiva propiedad.
- Realizar todos los cambios necesarios para que vuelva a funcionar como antes.

33. Crear la clase Libro:



El indexador leerá la página pedida, siempre y cuando el subíndice se encuentre dentro de un rango correcto, sino retornará un string vacío "". En el **set**, la asignará si esta ya existe. Si el índice es superior al máximo existente, agregará una nueva página.

Generar el código en el Main para probar la clase.

HERENCIA

34. Crear las clases Automovil, Moto y Camion.

- Crear un enumerado Colores { Rojo, Blanco, Azul, Gris, Negro }
- Camión tendrá los atributos cantidadRuedas : short, cantidadPuertas : short, color : Colores, cantidadMarchas : short, pesoCarga : int.
- Automovil: cantidadRuedas : short, cantidadPuertas : short, color : Colores, cantidadMarchas : short, cantidadPasajeros : int.
- Moto: cantidadRuedas : short, cantidadPuertas : short, color : Colores, cilindrada : short.
- Crearle a cada clase un constructor que reciba todos sus atributos.
- Crear la clase VehiculoTerrestre y abstraer la información necesaria de las clases anteriores. Luego generar una relación de herencia entre ellas, según corresponda.
- VehiculoTerrestre tendrá un constructor que recibirá todos sus atributos. Modificar las clases que heredan de ésta para que lo reutilicen.

Generar el código en el Main para probar las clases.

35. Tomar el ejercicio 32 de esta guía.

- Crear una clase Persona.
- Generar una nueva clase DirectorTecnico
- DirectorTecnico y Jugador deberán heredar de Persona, quedando el siguiente formato:



- d. Realizar las modificaciones necesarias para reutilizar código según la nueva estructura (constructores, métodos de exposición de datos).
- e. Realizar las propiedades necesarias en Persona y DirectorTecnico.

36. Tomar como base el ejercicio 30 de esta guía. Agregar la clase VehiculoDeCarrera y la clase MotoCross:

- a. Mover toda la información pedida a la clase VehiculoDeCarrera, modificando AutoF1, generando sus correspondientes propiedades.
- b. Dos VehiculoDeCarrera son iguales si coincide su número y escudería.
- c. Dos AutoF1 serán iguales cuando, además de coincidir los datos contenidos en VehiculoDeCarrera, coincide el atributo caballosDeFuerza.
- d. Dos MotoCross son iguales si coincide cuando, además de coincidir los datos contenidos en VehiculoDeCarrera, coincide el atributo cilindrada.
- e. El método Mostrar de VehiculoDeCarrera será el único capaz de exponer información de este tipo de objetos.
- f. En la clase Competencia cambiar el tipo de la lista por VehiculoDeCarrera.



- g. Si la competencia es declarada del tipo CarreraMotoCross, sólo se podrán agregar vehículos del tipo MotoCross. Si la competencia es del tipo F1, sólo se podrán agregar objetos AutoF1. Colocar dicha comparación dentro del == de la clase Competencia.
- h. Modificar todo lo que sea necesario para que el sistema siga comportándose de la misma forma, aceptando también vehículos del tipo MotoCross en la competencia.

37. Centralita – Herencia

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

- a. Crear en una solución llamada **CentralTelefonica** un proyecto de tipo **Consola** nombrado como **CentralitaHerencia** que contenga la siguiente jerarquía de clases:



Llamada:

- La **clase Llamada**, tendrá todos sus atributos con el modificador **protected**. Crear las propiedades de sólo lectura para exponer estos atributos.
- OrdenarPorDuracion es un método de clase que recibirá dos Llamadas. Se utilizará para ordenar una lista de llamadas de forma ascendente.
- Mostrar es un método de instancia. Utiliza StringBuilder.

Local:

- Herederá de Llamada.
- Método Mostrar expondrá, además de los atributos de la clase base, la propiedad CostoLlamada. Utilizar StringBuilder.
- CalcularCosto será privado. Retornará el valor de la llamada a partir de la duración y el costo de la misma.
- La propiedad CostoLlamada retornará el precio, que se calculará en el método CalcularCosto.

Provincial:

- i. Hederará de Llamada
- j. Método Mostrar expondrá, además de los atributos de la clase base, la propiedad CostoLlamada y franjaHoraria. Utilizar StringBuilder.
- k. CalcularCosto será 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.

Centralita:

- l. CalcularGanancia será privado. Este método recibe un Enumerado TipoLlamada y retornará el valor de lo recaudado, según el criterio elegido (ganancias por las llamadas del tipo Local, Provincial o de Todas según corresponda).
- m. El método Mostrar expondrá la razón social, la ganancia total, ganancia por llamados locales y provinciales y el detalle de las llamadas realizadas.
- n. La lista sólo se inicializará en el constructor por defecto Centralita().
- o. Las propiedades GananciaPorTotal, GananciaPorLocal y GananciaPorProvincial retornarán el precio de lo facturado según el criterio. Dichos valores se calcularán en el método CalcularGanancia().

Generar un nuevo proyecto del tipo Console Application llamado Test. El namespace también deberá llamarse Test. Agregar el siguiente código en el Main para probar la Centralita.

```
// Mi central
Centralita c = new Centralita("Fede Center");

// Mis 4 llamadas
Local l1 = new Local("Bernal", 30, "Rosario", 2.65f);
Provincial l2 = new Provincial("Morón", Provincial.Franja.Franja_1, 21, "Bernal");
Local l3 = new Local("Lanús", 45, "San Rafael", 1.99f);
Provincial l4 = new Provincial(Provincial.Franja.Franja_3, l2);

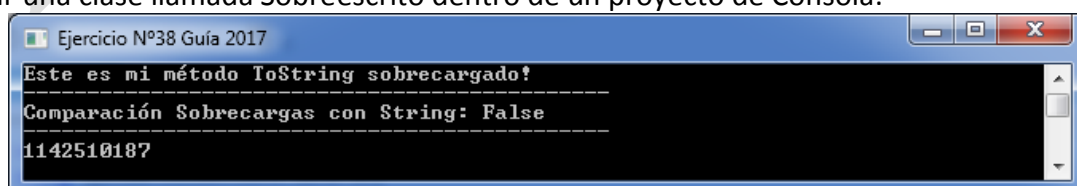
// Las llamadas se irán registrando en la Centralita.
// La centralita mostrará por pantalla todas las llamadas según las vaya registrando.
c.Llamadas.Add(l1);
Console.WriteLine(c.Mostrar());
c.Llamadas.Add(l2);
Console.WriteLine(c.Mostrar());
c.Llamadas.Add(l3);
Console.WriteLine(c.Mostrar());
c.Llamadas.Add(l4);
Console.WriteLine(c.Mostrar());

c.OrdenarLlamadas();
Console.WriteLine(c.Mostrar());

Console.ReadKey();
```

SOBRECARGA DE MÉTODOS, POLIMORFISMO, ABSTRACT y VIRTUAL

38. Crear una clase llamada Sobreescrito dentro de un proyecto de Consola:



- a. Sobreescibir el método ToString para que retorne "¡Este es mi método ToString sobreescrito!".

- b. Sobreescribir el método Equals para que retorne true si son del mismo tipo (objetos de la misma clase), false en caso contrario.
- c. Sobreescribir el método GetHashCode para que retorne el número 1142510187.
- d. Main:

```

Console.Title = "Ejercicio N°38 Guía 2017";
Sobreescrito sobrecarga = new Sobreescrito();

Console.WriteLine(sobrecarga.ToString());

string objeto = "¡Este es mi método ToString sobreescrito!";

Console.WriteLine("-----");
Console.WriteLine("Comparación Sobrecargas con String: ");
Console.WriteLine(sobrecarga.Equals(objeto));

Console.WriteLine("-----");
Console.WriteLine(sobrecarga.GetHashCode());

Console.ReadKey();

```

- 39. Tomar el ejercicio anterior. Sobreescrito será una clase abstracta.
 - a. Agregar un atributo miAtributo del tipo string, con visibilidad **protected**.
 - b. Generar un constructor de instancia que inicialice miAtributo con "Probar abstractos".
 - c. Agregará propiedad abstracta MiPropiedad de sólo lectura. Una vez implementada, retornará el valor de miAtributo.
 - d. Crear un método abstracto MiMetodo que retorne un string. Una vez implementada, retornará
 - e. Agregar una clase llamada SobreSobreescrito que herede de Sobreescrito. Implementar el código necesario para que todo funcione correctamente.
 - f. Modificar el Main para probar las modificaciones.
- 40. Centralita aplicando clases abstractas y polimorfismo: partiendo del ejercicio 37 (CentralitaHerencia) modificar la jerarquía de clases para obtener:



Llamada:

- La clase Llamada ahora será abstracta. Tendrá definida la propiedad de sólo lectura CostoLlamada que también será abstracta.
- Mostrar deberá ser declarado como virtual, protegido y retornará un string que contendrá los atributos propios de la clase Llamada
- El operador == 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.

Local:

- Sobreescribir el método Mostrar. Será 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.

f. ToString reutilizará el código del método Mostrar.

Provincial:

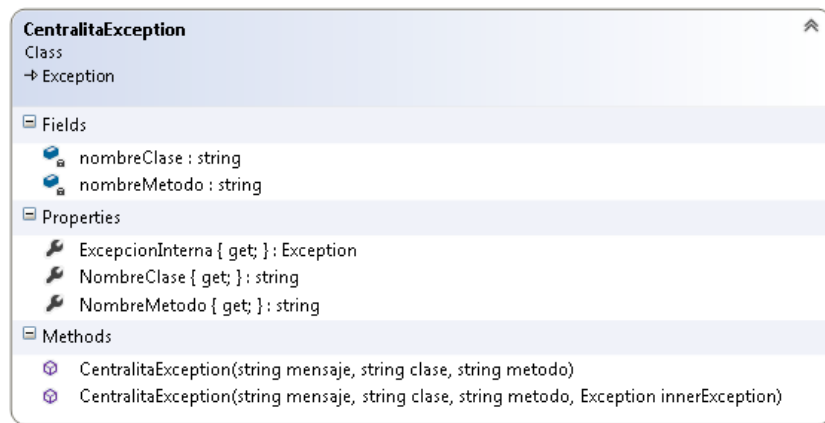
- g. El método Mostrar será protegido. Reutilizará el código escrito en la clase base y agregará `_franjaHoraria` y `CostoLlamada`, utilizando un `StringBuilder`.
- h. `Equals` retornará `true` sólo si el objeto que recibe es de tipo `Provincial`.
- i. `ToString` reutilizará el código del método `Mostrar`.

Centralita:

- j. Se reemplaza el método `Mostrar` por la sobrescritura del método `ToString`.
- k. `AgregarLlamada` es privado. Recibe una `Llamada` y la agrega a la lista de llamadas.
- l. El operador `==` retornará `true` si la `Centralita` contiene la `Llamada` en su lista genérica. Utilizar sobrecarga `==` de `Llamada`.
- m. El operador `+` invocará al método `AgregarLlamada` sólo si la llamada no está registrada en la `Centralita` (utilizar la sobrecarga del operador `==` de `Centralita`).

EXCEPCIONES

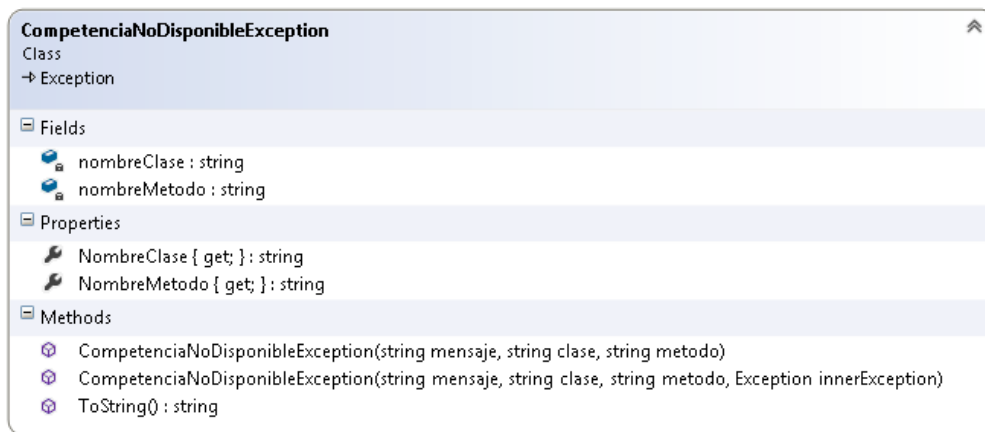
41. Centralita con excepciones: partiendo del ejercicio 40.



- a. Agregar la clase `CentralitaException`, la cual deriva de `Exception`.
- b. En el operador `+` de `Centralita`, lanzar la excepción `CentralitaException` en el caso de que la llamada se encuentre registrada en el sistema.

42. Crear el código necesario para lanzar una excepción `DivideByZeroException` en un método estático, capturarla en un constructor de instancia y relanzarla hacia otro constructor de instancia que creará una excepción propia llamada `UnaException` (utilizar `innerException` para almacenar la excepción original) y volver a lanzarla. Luego pasará por un método de instancia que generará una excepción propia llamada `MiException`. `MiException` será capturada en el `Main`, mostrando el mensaje de error que esta almacena y los mensajes de todas las excepciones almacenadas en sus `innerException`.

43. Tomar el ejercicio 36 de esta guía. Agregar la excepción `CompetenciaNoDisponibleException`.



- La sobrescritura del método ToString retornará un mensaje con el siguiente formato por líneas:
 - "Excepción en el método {0} de la clase {1}:"
 - Mensaje propio de la excepción
 - Todos los InnerException con una tabulación ('\t').
- La excepción CompetenciaNoDisponibleException será lanzada dentro de == de Competencia y Vehiculo con el mensaje "El vehículo no corresponde a la competencia", capturada dentro del operador + y vuelta a lanzar como una **nueva** excepción con el mensaje "Competencia incorrecta". Utilizar innerException para almacenar la excepción original.
- Modificar el Main para agregar un Vehiculo que no corresponda con la competencia, capturar la excepción y mostrar el error por pantalla.

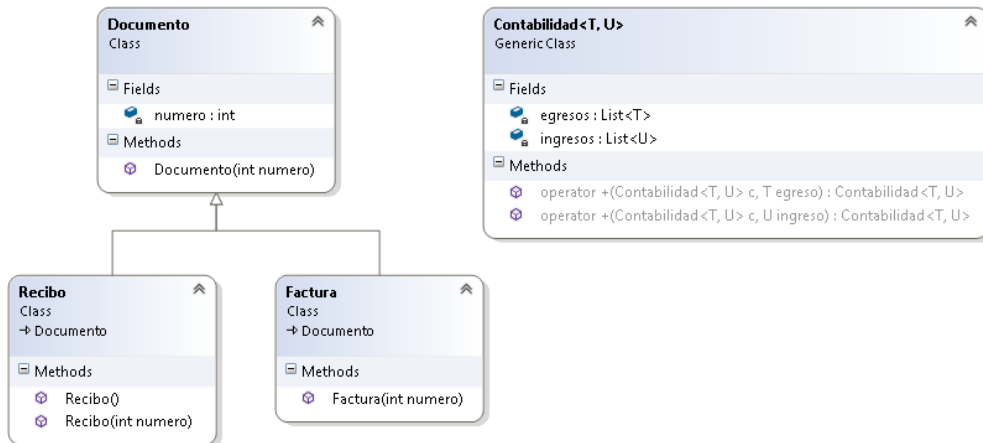
TEST UNITARIOS

- Tomar el ejercicio 41:
 - Crear un test unitario que valide que la lista de llamadas de la centralita esté instanciada al crear un nuevo objeto del tipo Centralita.
 - Controlar mediante un nuevo método de test unitario que la excepción CentralitaException se lance al intentar cargar una segunda llamada con solamente los mismos datos de origen y destino de una llamada Local ya existente.
 - Controlar mediante un nuevo método de test unitario que la excepción CentralitaException se lance al intentar cargar una segunda llamada con solamente los mismos datos de origen y destino de una llamada Provincial ya existente.
 - Dentro de un método de test unitario crear dos llamadas Local y dos Provincial, todos con los mismos datos de origen y destino. Luego comparar cada uno de estos cuatro objetos contra los demás, debiendo ser iguales solamente las instancias de Local y de Provincial entre sí.
- Tomar el ejercicio 42:
 - Realizar un test unitario para cada método y/o constructor.
 - Cada test deberá validar que el método lance la excepción que le corresponde.
- Tomar el ejercicio 43:
 - Agregar una propiedad que haga pública la lista de Vehiculos de Competencia.
 - Crear un test unitario que valide que la lista de vehículos de la competencia esté instanciada al crear un nuevo objeto.
 - Realizar un test unitario que controle que la excepción CompetenciaNoDisponible se lance al querer cargar un AutoF1 en una competencia del tipo MotoCross.

- d. Realizar otro test que controle que la excepción CompetenciaNoDisponible no se lance al querer cargar un objeto del tipo MotoCross en una competencia del tipo MotoCross.
- e. Comprobar que al cargar un nuevo vehículo en la competencia esté figure en la lista. Utilizar el operador + y el ==.
- f. Comprobar que al quitar un vehículo existente en la competencia esté ya no figure en la lista. Utilizar el operador - y el !=.

GENERICICS

47. Crear la clase Contabilidad para que tenga dos tipos genéricos:



- a. Crear un constructor que no reciba parámetros en Contabilidad para inicializar las listas.
- b. El constructor sin parámetros de Recibo asignará 0 como número de documento.
- c. Tanto el tipo genérico T como el U deberán ser del tipo Documento o uno de sus derivados.
- d. El tipo U deberá tener una restricción que indique que deberá tener un constructor por defecto (sin argumentos).
- e. El operador + entre Contabilidad y T agregará un elemento a la lista egresos.
- f. El operador + entre Contabilidad y U agregará un elemento a la lista ingresos.
- g. Generar el código necesario para probar dichas clases.

48. Tomar el ejercicio 46.

- a. La clase Competencia deberá tener un tipo genérico, el cual sólo podrá ser del tipo VehiculoDeCarrera o uno de sus derivados.
- b. La lista de competidores cambiará del tipo VehiculoDeCarrera al tipo genérico.
- c. Realizar todos los cambios necesarios para que esto funcione correctamente.

49.

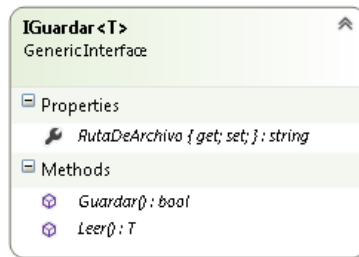
INTERFACES

50. Crear la siguiente estructura de datos:



- La interfaz IGuardar será implementada por ambas clases.
- El método Guardar simplemente retornará True.
- El método Leer retornará la leyenda "Texto Leído" para GuardarTexto y "Objeto Leído" para Serializar. Se deberá utilizar Convert.ChangeType (investigar).

51. Tomar el ejercicio 44:



- Implementar la interfaz en la clase Centralita para datos del tipo String:
 - Guardar tomará el objeto y consultará todos sus datos, luego retornará true.
 - Leer lanzará la excepción NotImplementedException.
- Implementar la interfaz en las llamadas del tipo Local y Provincial para los tipos de datos Local y Provincial respectivamente.
 - Tanto Leer como Guardar lanzarán la excepción NotImplementedException.

52. Generar el siguiente esquema de clases:



- En Lapiz:

- i. Escribir reducirá la mina en 0.1 por cada carácter escrito.
 - ii. Recargar lanzará NotImplementedException.
 - iii. El color será gris (grey), sin posibilidad de morificarlo. El set lanzará NotImplementedException.
- b. En Boligrafo:
 - i. Escribir reducirá la tinta en 0.3 por cada carácter escrito.
 - ii. Recargar incrementará tinta en tantas unidades como se agreguen.
- c. En ambas clases el método ToString retornará un texto informando que es (Lapiz o Boligrafo), color de escritura y nivel de tinta.
- d. Probar el siguiente código en el Main:

```
ConsoleColor colorOriginal = Console.ForegroundColor;
```

```
Lapiz miLapiz = new Lapiz(10);
```

```
Boligrafo miBoligrafo = new Boligrafo(20, ConsoleColor.Green);
```

```
EscrituraWrapper eLapiz = miLapiz.Escribir("Hola");
```

```
Console.ForegroundColor = eLapiz.color;
```

```
Console.WriteLine(eLapiz.texto);
```

```
Console.ForegroundColor = colorOriginal;
```

```
Console.WriteLine(miLapiz);
```

```
EscrituraWrapper eBoligrafo = miBoligrafo.Escribir("Hola");
```

```
Console.ForegroundColor = eBoligrafo.color;
```

```
Console.WriteLine(eBoligrafo.texto);
```

```
Console.ForegroundColor = colorOriginal;
```

```
Console.WriteLine(miBoligrafo);
```

```
Console.ReadKey();
```

ARCHIVOS DE TEXTO

53. Tomar el ejercicio 45 y agregarle un nuevo proyecto llamado IO. Dentro de este proyecto crear la clase ArchivoTexto estática:
 - a. El método Guardar agregará información al archivo de texto ubicado en la ruta pasada como parámetro. También recibirá un string con la información a guardar.
 - b. El método Leer retornará el contenido del archivo ubicado en la ruta pasada como parámetro. En caso de no existir, lanzará la excepción de sistema FileNotFoundException.
 - c. Modificar en el Main donde se captura la excepción. Quitar los Console.WriteLine y guardar todos los datos del error en un archivo de texto, cuyo nombre será la fecha y hora actual como indica este ejemplo: '20171012-1316.txt' (año mes día – hora minutos).
 - d. Luego, fuera del catch, utilizar el método Leer para mostrar por pantalla los mensajes de error.
54. Tomar el ejercicio 51 de la guía:
 - a. El método Guardar de la implementación de IGuardar en Centralita deberá guardar en un archivo de texto a modo de bitácora fecha y hora con el siguiente formato “Jueves 19 de octubre de 2017 19:09hs – Se realizó una llamada”; para lo cual este método deberá ser

llamado desde el operador + (suma). En caso de no poder guardar, igualmente agregar la llamada a la Centralita y luego lanzar la excepción **FallaLogException**.

- b. El método Leer deberá obtener los datos de un archivo dado y retornarlos.
- c. En el método Main modificar el código para que, antes de salir, muestre el log.

55. Crear un proyecto de formularios capaz de abrir, editar y guardar archivos de texto, tal como un editor simple de texto como ser el Notepad de Windows.



- a. Agregar un menú superior (MenuStrip) con las opciones de Archivo->Guardar, Archivo->Guardar como... y Archivo->Abrir.
- b. Agregar una barra de estado en la parte inferior (StatusStrip). Se debe informar el total de caracteres del archivo.
- c. Al pulsar en el menú "Abrir" o "Guardar como...", se deberá abrir una ventana para seleccionar los archivos (ver nota al pie).
- d. Al hacer click sobre "Guardar", se deberá guardar el mismo archivo abierto.
- e. El editor (RichTextBox) deberá estar acoplado (Dock) al formulario.

NOTA: Utilizar OpenFileDialog y SaveFileDialog para buscar los archivos en las carpetas del sistema.

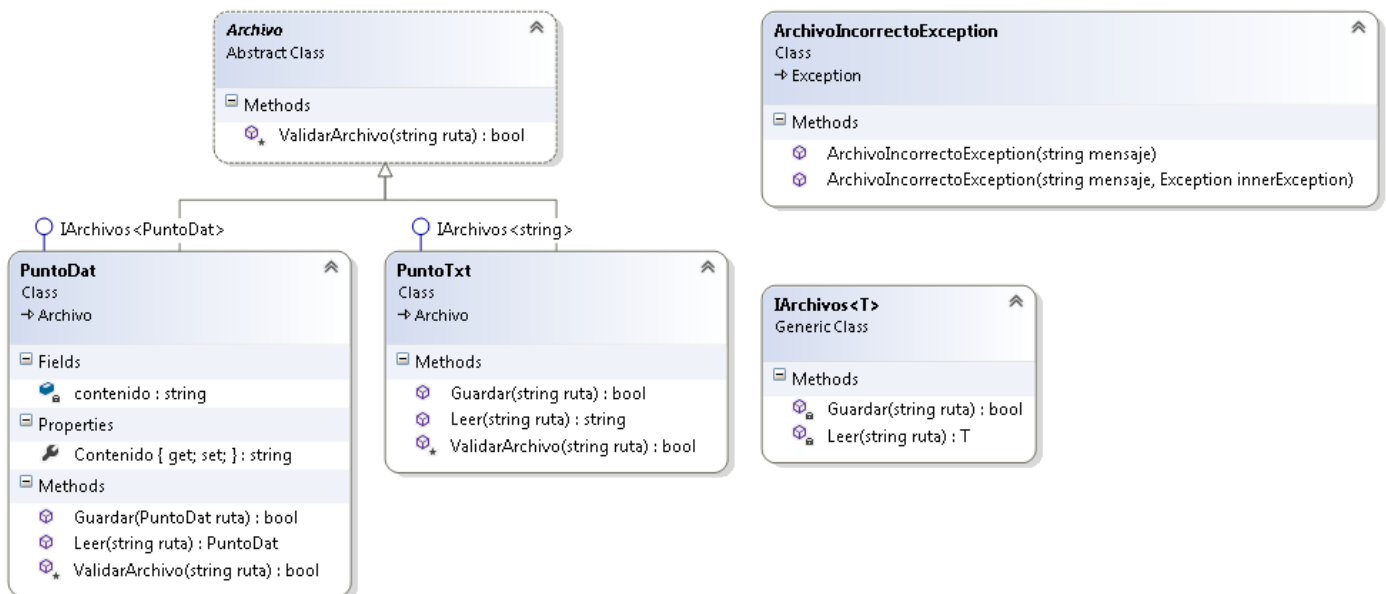
SERIALIZACIÓN

56. Crear un nuevo proyecto del tipo Consola.

- a. Crear una clase Persona con dos atributos privados del tipo string, nombre y apellido.
 - i. Agregarle un constructor que reciba ambos parámetros.
 - ii. Crear un método estático llamado Guardar que reciba una persona y la serialice en un archivo.
 - iii. Crear un método estático Leer que deserialice un archivo y retorne una Persona.
 - iv. Sobrecargar el método ToString para mostrar los datos de la persona.
- b. En el Main instanciar un objeto del tipo Persona e intentar serializarlo.
- c. Luego intentar leer ese objeto serializado en una nueva instancia de persona y lo muestre por pantalla.
- d. Por cada excepción que lance la aplicación:
 - i. Generar un **catch** que la capture y la maneje.
 - ii. Luego corregir el problema que genera la excepción.
 - iii. Repetir el proceso hasta capturar todas las excepciones de forma individual (no utilizar Exception).

- iv. Los datos de la Persona guardada deben ser el nombre y el apellido, y coincidir con los de la Persona leída.

57. Tomar el formulario y comportamiento del ejercicio 55. Crear un proyecto llamado IO y agregar las siguientes clases e interfaces:



- En Archivo, ValidarArchivo comprobará que el archivo exista. De no existir, lanzará la excepción FileNotFoundException.
- En PuntoDat, ValidarArchivo comprobará que el archivo exista y que su extensión sea del tipo ".dat".
- En PuntoTxt, ValidarArchivo comprobará que el archivo exista y que su extensión sea del tipo ".txt".
- En ambas validaciones:
 - De no ser la extensión correcta, lanzará la excepción ArchivoIncorrectoException con el mensaje "El archivo no es un dat."
 - De no existir, capturará la excepción y la relanzará como ArchivoIncorrectoException, incorporando en su innerException a la original, con el mensaje "El archivo no es correcto."
- En el formulario, al ir a los menú de "Abrir" o "Guardar como...", el cuadro de dialogo deberá ofrecer la posibilidad de abrir "Archivos de texto (.txt)" o "Archivos de datos (.dat)". Para esto, utilizar la propiedad Filter.
- La serialización para PuntoDat será binaria.
- Realizar las modificaciones necesarias para que funcione correctamente.

58. Tomar el ejercicio 54 de la guía, y serializar mediante XML:

- Los métodos de la implementación de IGuardar en Local deberán obtener los datos de un archivo dado, comprobar que estos sean del tipo Local y retornar un nuevo objeto de este tipo. En caso de que no sea del tipo Local, lanzará la excepción InvalidCastException.
- Los métodos de la implementación de IGuardar en Provincial deberán guardar y obtener los datos de un archivo dado, comprobar que estos sean del tipo Provincial y retornar un nuevo objeto de este tipo. En caso de que no sea del tipo Provincial, lanzarán la excepción InvalidCastException.

59. Crear un proyecto de Windows Form, y utilizando las bases de datos de prueba de Microsoft SQL Server (<https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/adventure-works>) crear un programa que permita agregar, modificar y eliminar elementos de la tabla Production.Product.
60. En un nuevo proyecto de Windows Form:
- Crear una tabla en una base de datos llamada Persona con los campos:
 - ID, autoincremental y entero.
 - Nombre, varchar(50).
 - Apellido, varchar(50).
 - Crear una clase Persona con nombre y apellido como atributos privados.
 - Crear un constructor que reciba ambos parámetros.
 - Crear una clase llamada PersonaDAO y agregarle 5 métodos:
 - Guardar: guardará una nueva persona en la base de datos.
 - Leer: retornará la lista de personas de la base de datos.
 - LeerPorID: traerá una persona, filtrando por ID.
 - Modificar: modificará una persona a partir de su ID.
 - Borrar: eliminará una persona de la base de datos a partir de su ID.
 - Armar un formulario con dos TextBox (txtNombre y txtApellido), un ListBox (lstPersonas) y 4 botones (btnGuardar, btnModificar, btnEliminar y btnLeer).
 - ListBox mostrará la lista de Personas devuelta por el método Leer de Persona, invocado al presionar el botón btnLeer.
 - btnModificar actualizará la información de la Persona que se seleccionó con doble click en el ListBox.
 - Al hacer doble click sobre una persona, esta se deberá cargar en los TextBox.
 - btnGuardar agregará una Persona en la base de datos.
 - btnEliminar borrará de la base a la persona seleccionada en el ListBox.
61. Tomar el ejercicio 58. Crear un nuevo proyecto llamado EntidadesDAO, dónde se crearán las clases LocalDAO y ProvincialDAO.
- Ambas clases implementaran la interfaz IGuardar, proveyendo el acceso a base de datos para guardar y leer los datos de esas llamadas.
 - Al querer leer una llamada desde la base, se hará el filtro a partir de los datos de duración, tipo, origen y destino.
 - La única tabla en la base de datos será Llamadas con los campos:
 - ID: entero, autoincremental y clave primaria.
 - Duración: entero.
 - Origen: varchar(50).
 - Destino: varchar(50).
 - Tipo: 0 (local) o 1 (provincial).

THREADS

62. Tomar el ejercicio XX de esta guía (AutoF1). Hacerlos correr.
63. Simulador de llamadas para centralita

EVENTOS

- 64.

PRELIMINAR