***Crear nuevo proyecto***

Al ingresar el comando dotnet new la consola muestra los tipos de proyectos que se pueden iniciar

Ejemplo: dotnet new console crea una aplicación de consola.

Al crear el proyecto, se van a crear automáticamente tres archivos:

* obj
* nombreProyecto.csproj
* Program.cs

Otra forma de generar el proyecto podría ser dotnet new console --output \*nombreCarpeta\*. Esto nos permite indicar el que lugar se desea generar el nuevo proyecto.

***Ejecutar el proyecto***

El comando dotnet run ejecuta la aplicación

***Construir aplicación***

dotnet build analiza y compila todo el código y genera el binario final.

Observaciones:

El *archivo.dll* es el archivo nativo que se puede ejecutar desde cualquier sistema operativo.

dotnet \*archivo.dll ejecuta el programa, pero a diferencia de build, directamente ejecuta sin analizar el código.\*

**Generando ejecutables especializados u optimizados para un sistema en particular.**

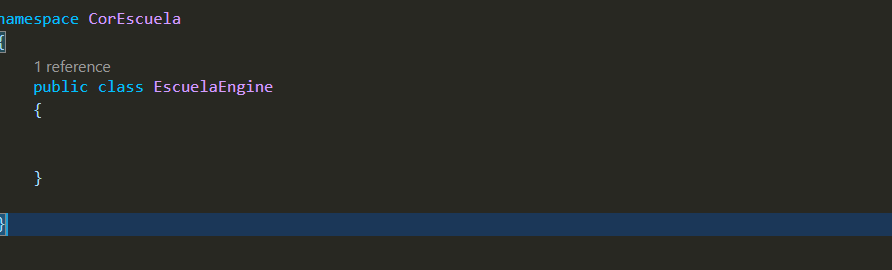
El flag ***-c*** se utiliza para especificar si será una versión de *producción* o *desarrollo. Release* es para producción.

El flag **-r** determina el sistema operativo objetivo y versión para la cual se desea optimizar el ejecutable.

**Ejemplo**: dotnet build -c Release -r win10-x64

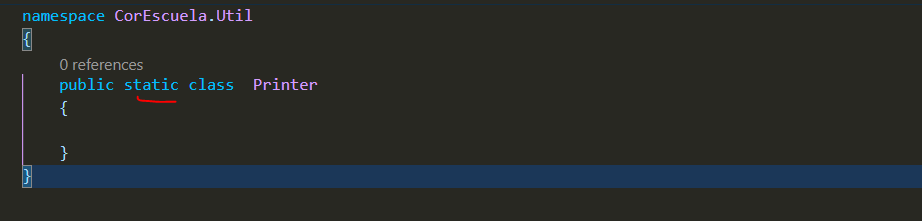
**Creación de clases.**

* Clase normal donde se debe instanciar un objeto a la clase



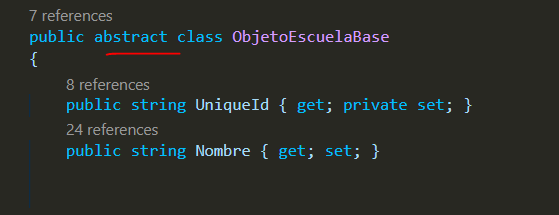
* Clase estática donde solo se usa ella misma como una instancia. Se usa la palabra clase static.

Una *clase estática* es básicamente lo mismo que una clase no estática, pero hay una diferencia, una clase estática no puede ser instanciada. En otras palabras, no puede usar el operador new para crear una variable del tipo de clase. Como no hay una variable de instancia, puede acceder a los miembros de una clase estática utilizando el nombre de la clase en sí.

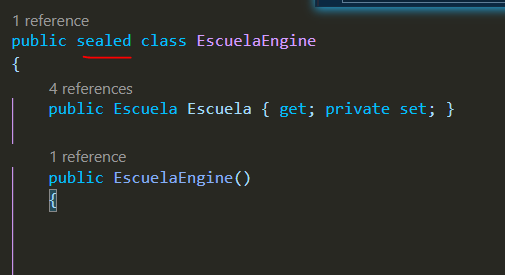


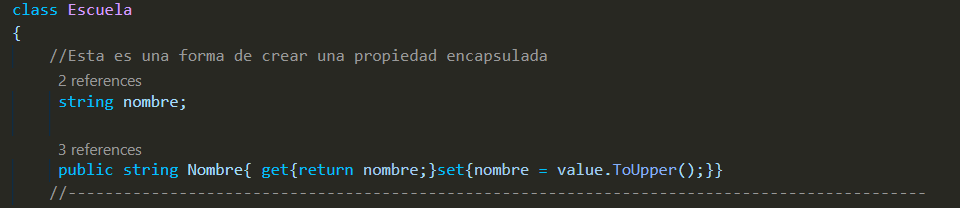
**Modificadores de Tipo a las clases.**

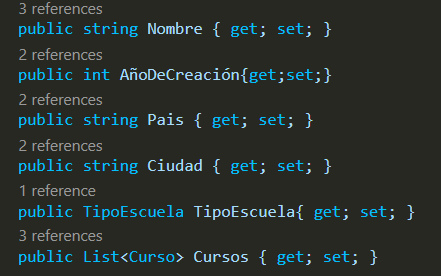
"abstract" Pemite que la clase no se permita instanciar.

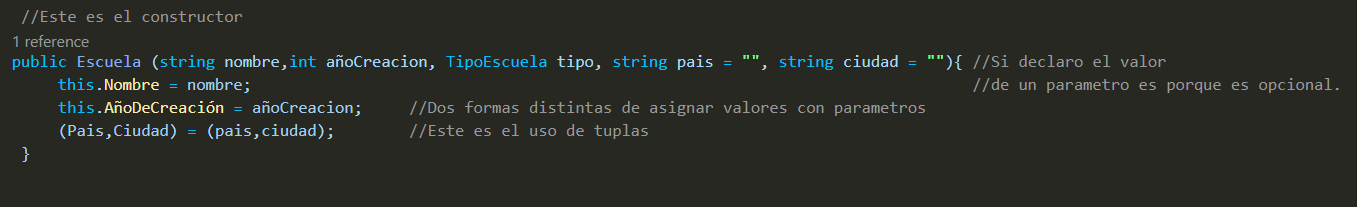
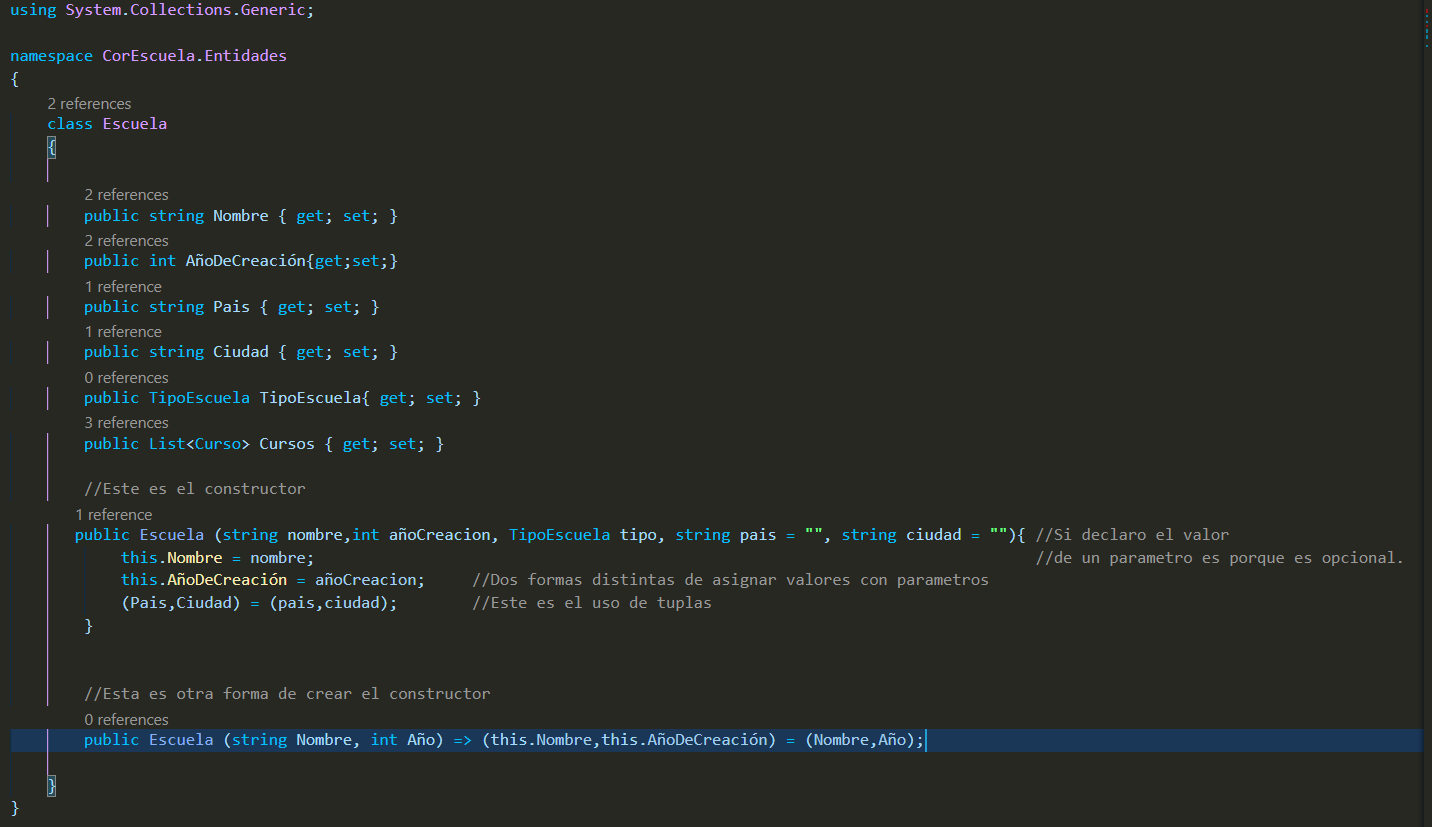


"Sealed" Sella la clase para que permita instanciar pero no heredar.



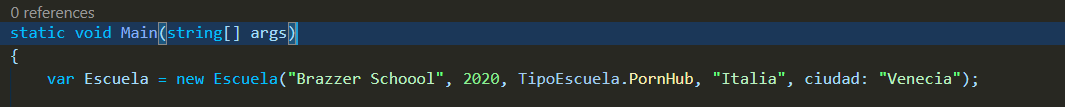
**Creación de Propiedades en una clase.**



**Constructores en una clase.**

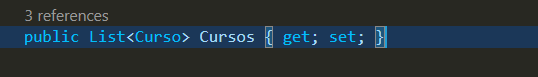
* En la clase se pueden crear tantos constructores sean requerido para la necesidad del programa.

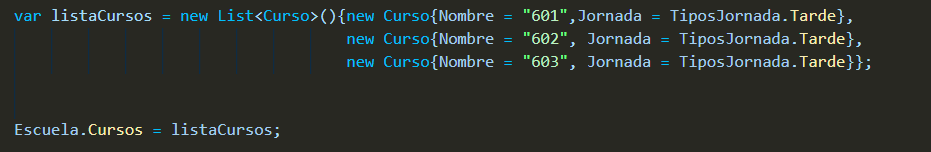
**Instanciar una clase.**

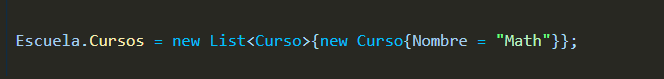
Se usa la palabra clave **New** y el nombre del objeto que quiero Instanciar.

**Asignar clases como tipo de datos.**

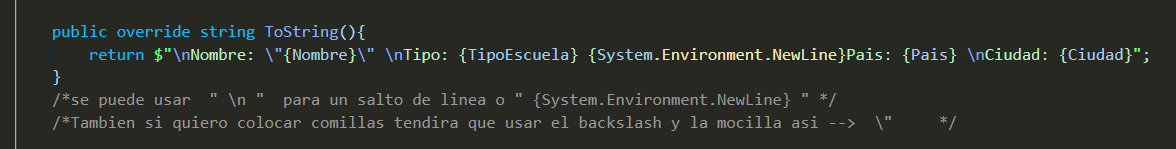
En este caso se añade la clase **Curso** como propiedad que recibe una lista de Cursos.







**Formas de sobrescribir los métodos de una clase.**

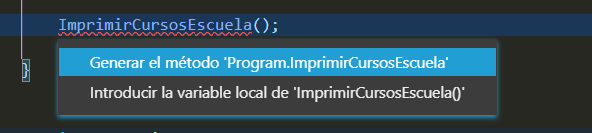
En este caso se sobrescribió el método ToString(). Esto se incluye dentro de la clase.

**Generar un método automáticamente.**

* Creo la sentencia :

*ImprimirCursosEscuela();*

* Luego poniendo el cursor sobre la sentencia oprimo:
  + CONTROL + (TECLA PUNTO)



* Le doy como se muestra en la captura y listo.

**Seleccionar varias líneas seguidas Visual Code.**

ALT + CONTROL+ ↑/↓

**Seleccionar varias líneas clickeando Visual Code.**

ALT + CLICK (en la línea)

**Formatear Código Visual Code.**

SHIFT + ALF + F

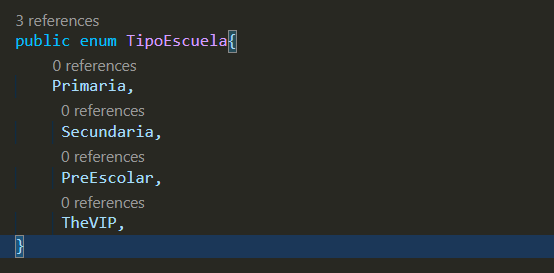
**Buscar archivo en Visual Code.**

CONTROL + P

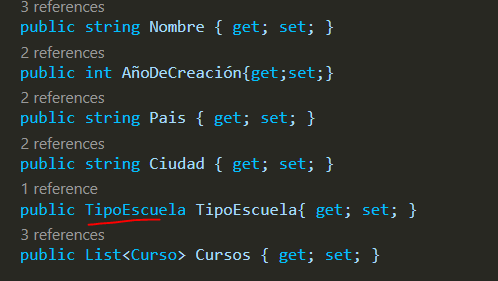
**Abrir archivos pantalla dividida en Visual Code.**

ALT + CONTROL+ ↔ (left or right)

**Crear Enumeraciones.**



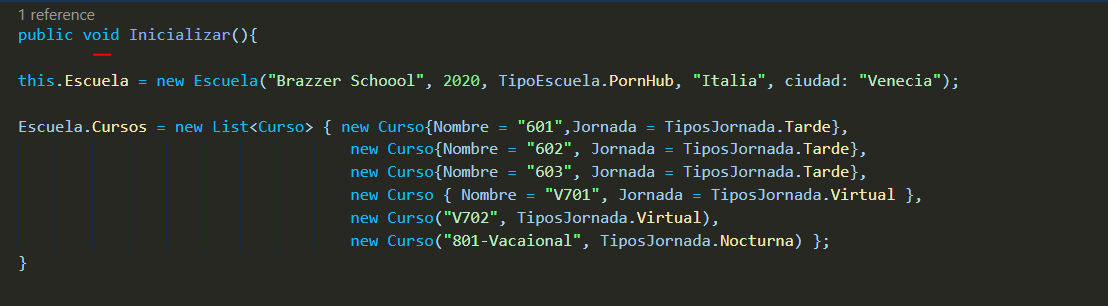
**Asignar Enumeración como Tipo de Dato en una propiedad de un clase.**

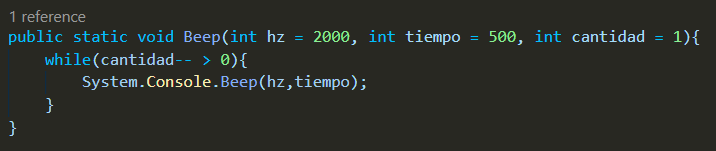


**Creación de Métodos.**

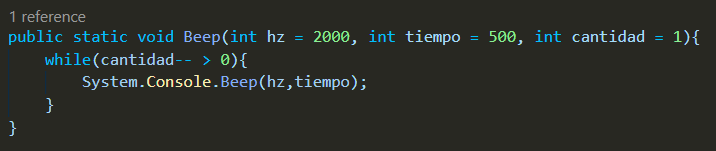
Cuando queremos crear un método que nos va a devolver nada vamos a utilizar la palabra clave **void.**

**Ejem. 1**





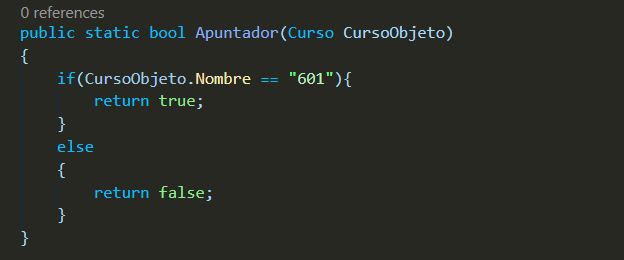
**Ejem. 2**





**Creación de Funciones.**

Cuando creamos una función debemos especificar el tipo de dato que va a ser retornado. En este caso tiene que retornar un booleano por lo que usaremos la palabra clave **bool.**



**Condiciones.**

El operador **if** nos indica si se cumple o no una condición, si la condición se cumple entonces ejecuta el bloque de código, en caso de no cumplirse no se ejecuta. La sentencia **else** se ejecuta cuando un if no se cumple y la sentencia **else if** lo que hace es en caso de no cumplirse un if, pregunta por una nueva condición.

Existen diferentes operadores que resultan útiles para verificar condicionales, algunos de ellos son:

* || o or.
* && o and
* % operador MOD, retorna el residuo de una división.

# Uso del While.

# 

# Uso del Do While.

# 

# Uso del For.

# 

# Uso del ForEach.

# 

# Uso del If -Else

# 

# Crear Arrays. (Arreglos)

# Se utiliza palabra clave New Dentro de los símbolos [] especifico la cantidad de posiciones que va a tener el arreglo y anteriormente el tipo de dato que va a almacenar el array.

# *New* TipoDato[Cantidad];

# 

# 

# 

# Asignar valores en las posiciones del Array.

# 

# Creación de Colecciones.

# En este caso se crea una colección de cursos con la palabra clave new List.

# Dentro de los <> se escribe el tipo de dato que va a almacenar la colección. En este caso de tipo Curso

# New List <tipo de dato>

# 

# Adicionar ítems.

# Se debe usar el método .add()

# 

# 

# Eliminar Items.

# 

# Agregar una colección a otra colección.

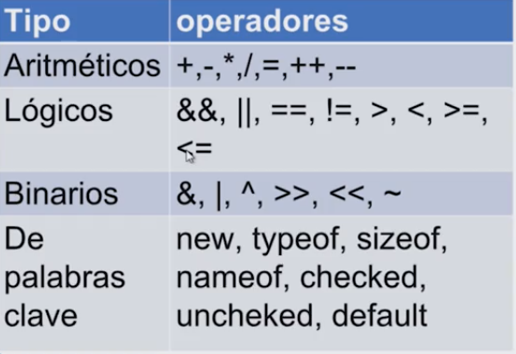
# Se debe usar el método .addRange()

# 

# Limpiar o eliminar datos de una colección.

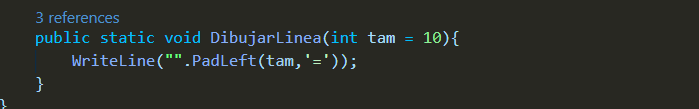
# 

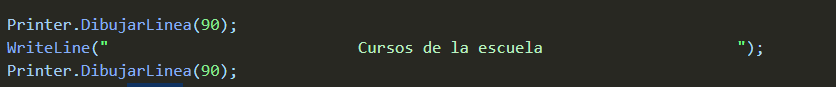
# Operadores en C#.



**Para saber:**

Función que rellena hacia la Izquierda o Derecha una cadena. Se debe especificar con cuantas repeticiones va a rellenar y cuál es el caracteres que se va a repetir.

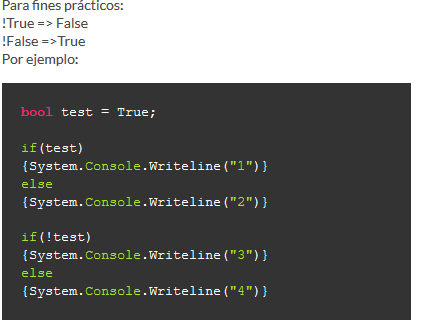




***Impresión.***

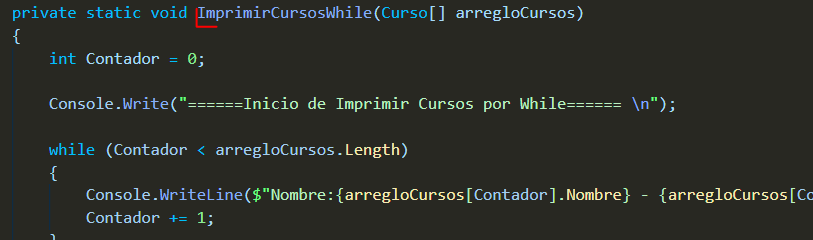


***Static***: palabra clave utilizada para declarar clases o métodos que no pueden ser instanciados, solo exponen unos métodos que pueden ser usados, no necesita que sus datos permanezcan en el tiempo entre instancias; una clase puede tener métodos estáticos que deben ser usados sin necesidad de instanciar una clase.



Resultado en consola: 1 4

Por convención el nombre de un método debe Iniciar con Mayúscula.



-----------------------------------------------------------------------------------------------------

**HashCode** → el hash es un propiedad de todos los objetos de .Net, es un identificador del objeto; cuando se emplea el método Remove de las colecciones esto lo que hace es comparar los objetos por su HashCode para identificar cual es el que debe eliminar

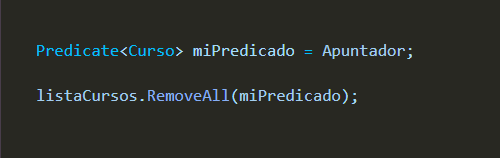
Las expresiones lambda son formas reducidas de escribir algoritmos sin necesidad de especificar un método por completo ni diseñar una clase que lo encapsule.

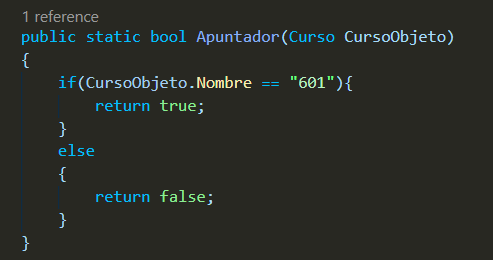
***Predicado parte 1.***

Un predicado no es mas que una funcion que pasamos como argumento en métodos que incluyen nuestras colecciones para ser recorridas como RemoveAll(***predicado***), findAll(***predicado***).

Todo esto es muy parecido a lo que conocemos como arrow function dentro de nuestros métodos de arrays en JS. (map(***arrowfunction***), filter(**arrowfunction**),etc)

En este caso creamos un predicado que es un método que devuelve un Booleano y que recibe como parámetro el tipo de dato especificado.

Un delegado nos permite pasar un método como parámetro de otro método.

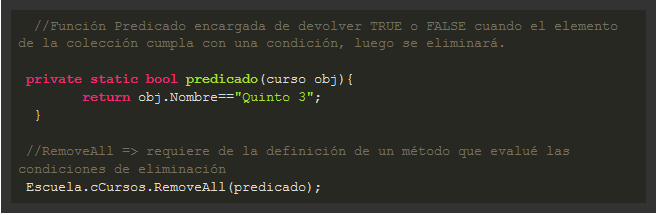


En este este ejemplo lo que se requiere es eliminar el curso 601.

* Para este caso uso el método RemoveAll();
* Como parámetro le envió “miPredicado” que simplemente es una variable de Tipo Predicado.
* Al momento de crear esta variable de Tipo Predicado le especifico el Tipo de dato que va a evaluar que en este caso es de tipo “Curso” y en la asignación solo me puede devolver un booleano .
* En este caso asigno un método que me devuelve el objeto que cumpla con las condiciones que requiero (En este ejemplo que el curso sea el 601)

***Predicado parte 2.***

Aquí como podemos ver vamos a usar un predicado en la utilización del método **RemoveAll()** de la colección:



1. Delegados: Especifican que parámetros de Entrada y de Salida debe tener una función, Ejemplo (Pasos):

Definición del delegado:

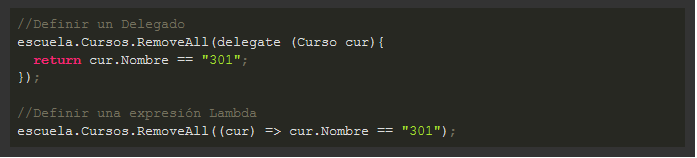
PalabraClave<tipoDatoEntrada> nomDelegado = metodoDelDelegado;  
**Ej**: Predicate<string> isUpper = IsUpperCase;

Definición del método:

En su definición solo retorna TRUE or FALSE => Evalúa una condición

***static bool*** *IsUpperCase(****string*** *str){* ***return*** *str.****Equals****(str.ToUpper());  
}*

Uso del delegado, en el return será igual a true dado que es un String el parámetro pasado.



***Sin embargo un alternativa a los Predicate son las expresiones lambda.***

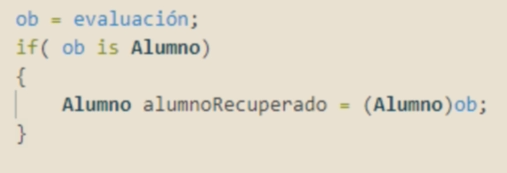
**Validaciones de tipo de objetos.**

**En caso de usar una lista de objetos poliformica vamos a tener que aprender a validar los tipos de objetos.**

Vamos a crear un método que nos regrese todos los objetos Escuela de la escuela a través de un List de ObjetoEscuelaBase.

En C# podemos realizar validaciones sobre el tipo de objeto que estamos manejando, usando las siguientes palabras clave:

* **is**, para verificar si un objeto es de un tipo en específico.
* **as**, para tratar un objeto como un tipo específico, en caso de no poder convertir el objeto entonces va a asignar un valor null.

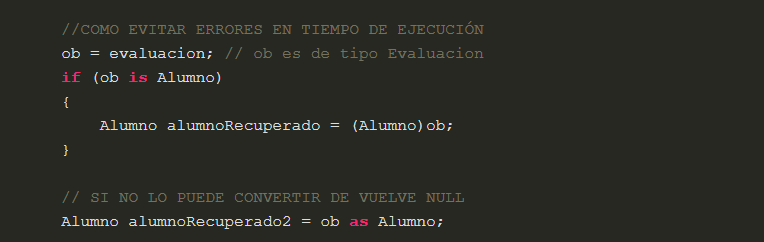


De esta manera se puede evaluar que “Ob” sea de tipo “Alumno”. Aquí en caso de que el objeto sea de tipo alumno lo va recuperar en una variable llama “alumnoRecuperado”.

“(Alumno) ob ” es una forma de Castear o Parsear en este caso “ob” en un “Alumno”

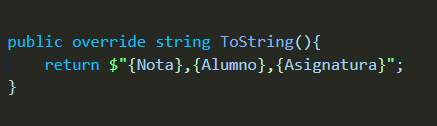


De esta otra manera en caso de que la sentencia “ob as Alumno” se cumpla lo devolverá como AlumnoRecuperado2. De no ser asi simplemente alumnoRecuperado2 será null.



# Utilizando ToString para mejorar la depuración.

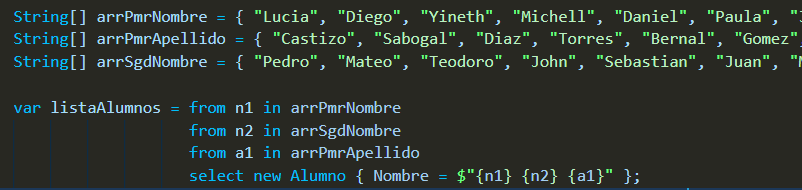
Vamos a sobrescribir el método ToString para mejorar la depuración de nuestro proyecto. Gracias a la herencia podemos sobrescribir el método en nuestra clase padre y todas sus clases hijas también tendrán sobrescrito el método.



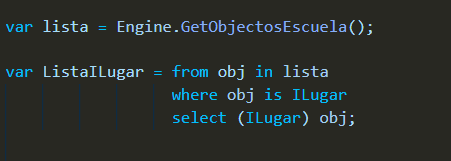
Sobrescribiendo el método ToString() para que al momento de ver los Objetos en el debugger se muestren justo como los necesitamos. En este caso se sobrescribió el método en la Clase “Evaluación” para que me imprima exactamente la nota, el alumno y la asignatura.

# *Creación de Linq*

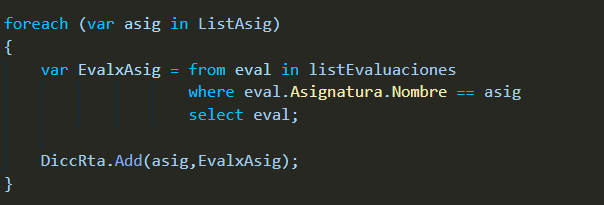
# *Ejemplo 1*



# *Ejemplo 2*



# *Ejemplo 3*



# *Ejemplo 4*

# 

# *Ejemplo 5*

# 

# *SQL LINQ*

SELECT FROM

FROM WHERE or GROUP BY

WHERE SELECT

Links:

<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries>

<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/linq/basic-linq-query-operations>

# Uso de interfaces

Desde el punto de vista de la programación orientada a objetos cualquier objeto puede ser visto como una interfaz.

Haciendo uso de Linq vamos a seleccionar de nuestra lista de objetos todos los objetos y les haremos casting con ILugar. Si no declaramos que nos retorne sólo los objetos que sean de tipo ILugar, entonces el programa al realizar el casting va a causar errores silenciosos.

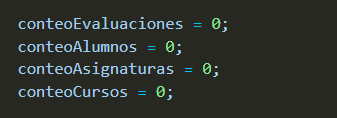
# Creación y uso de regiones

Una clase con muchas responsabilidades será un indicio de que nuestro código no está totalmente optimizado.

Las regiones no cambian a nivel de programación, pero si en la utilización del IDE. Situando código entre **#region** y **#endregion** el IDE va a ser capaz de colapsar todo ese bloque de código mejorando la legibilidad de este.

# Asignación múltiple de variables

En lugar de realizar esto:



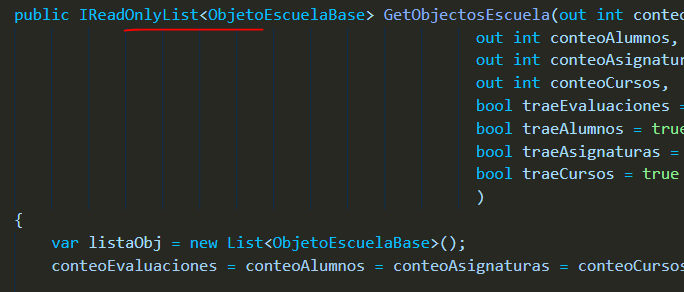
Se puede realizar de esta manera:



# Listas de solo Lectura

Nuestro método GetObjetoEscuela nos regresa una lista que debe utilizarse solo para lectura de objetos, sin embargo, al regresar un objeto tipo List es posible para el desarrollador añadir nuevos objetos. Podemos solucionar esto cambiando el tipo **List** por **IReadOnlyList**.

Recuerda que una buena práctica para las listas de datos que son de manera pública es regresar un tipo de lista genérico, como IEnumerable y en los casos donde serán de solo lectura regresar un tipo IReadOnlyList.



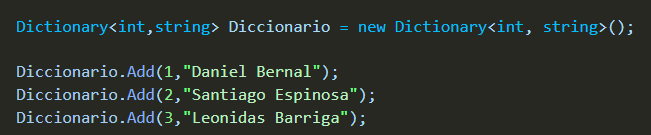


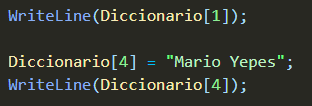
# Diccionario Polimórfico.

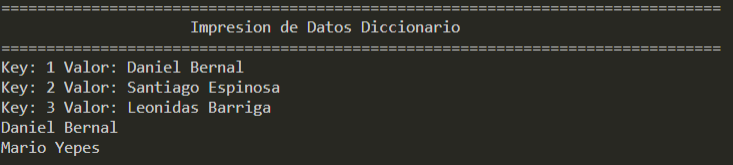
El objeto tipo Dictionary necesita dos parámetros, el primero es la llave y el segundo es el contenido. Haciendo que la instancia de un Dictionary quede de la siguiente forma:

Dictionary<int, string> diccionario = new Dictionary<int, string>();

Las llaves en los diccionarios son únicas.







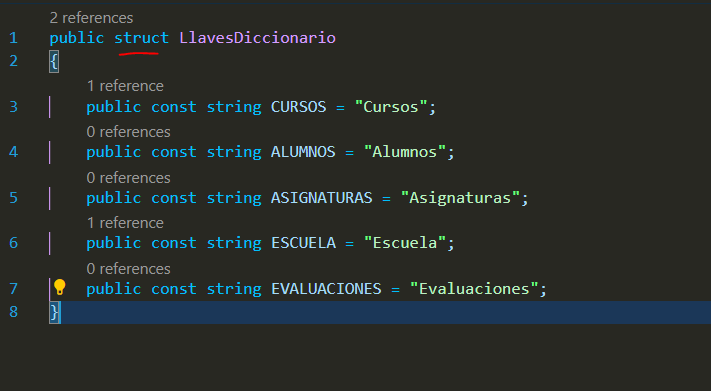
# Estructuras Simples en lugar de clases.

En este caso se crean unas constantes que pueden ser creadas también en una clase. Pero en lugar de crearlas como una clase se puede crear como se visualiza en la imagen con la palabra clave “***struct***”

Las estructuras son un tipo de clase que pueden ser usadas para almacenar objetos que directamente contienen datos, uno de sus usos es pequeñas estructuras de datos y para los diccionarios como es el caso de la clase.

Estas son algunas de las características de las estructuras:

• No pueden tener un constructor sin parámetros.  
• Guardan los valores y no la referencia a la memoria como es el caso de las clases.  
• Pueden ser instanciadas sin usar new.  
• No pueden heredar de otras estructuras o clases pero si implementar interfaces.  
• Los modificadores abstract y sealed no pueden usarse en ellas.  
• Se les puede asignar null como valor.

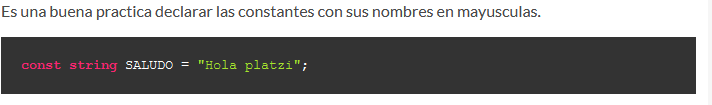


# Uso de numeraciones en diccionario

Teniendo un diccionario que sus llaves sean de tipo string vamos a encontrarnos con un problema al momento de programar: podríamos equivocarnos en el nombre de la llave escribiendo “cursos” en lugar de “Cursos”, esto ocasionaría que estemos accediendo a un lugar distinto del diccionario.

Para solucionar nuestro problema podemos hacer uso de variables constantes, pero solamente funcionaria dentro del fragmento de código actual. Una mejor solución sería hacer uso de un **enum** con los distintos nombres de las llaves y cambiar el tipo de la llave del diccionario de string a enum.

# Declaración de constantes

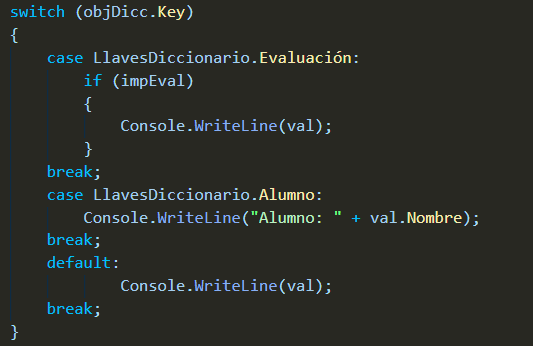


# Uso de la sentencia switch

La sentencia switch es más fácil de leer y por lo tanto una mejor opción cuando empiezas a tener muchas comparaciones con sentencias if.

En switch debes elegir sobre que variable vas a hacer las comparaciones, luego programas cada caso de la variable dentro del bloque ***case*** y cierras este bloque con la palabra ***break***.

Por último, al final del switch debes marcar el caso por ***default*** que se ejecutara cuando no coincida con ningún caso que programaste.



**Diferencia entre Math y MathF**

**Math y MathF:** Proporciona constantes y métodos estáticos para funciones trigonométricas, logarítmicas y otras funciones matemáticas comunes.

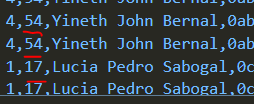
**MathF:** Los campos y métodos estáticos de la *clase MathF* corresponden a los de la *clase Math* , excepto que sus parámetros son de tipo Single en lugar de Double , y devuelven valores Single en lugar de Double .



En este caso MathF.Round recibe dos parámetros el valor a redondear y la cantidad de valores que quiero dejar.

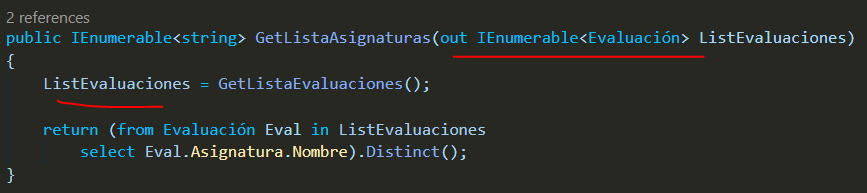
MathF.Round( ***Valor***, ***CantidadDecimales*** )

Impresión:



**Declaración de variables de salida**

Dentro de un método puedo devolver una variable que se usó dentro del mismo. Para esto uso la palabra clase **out** luego el **tipo de variable** y el **nombre de la variable** que se usó dentro de la función o método.



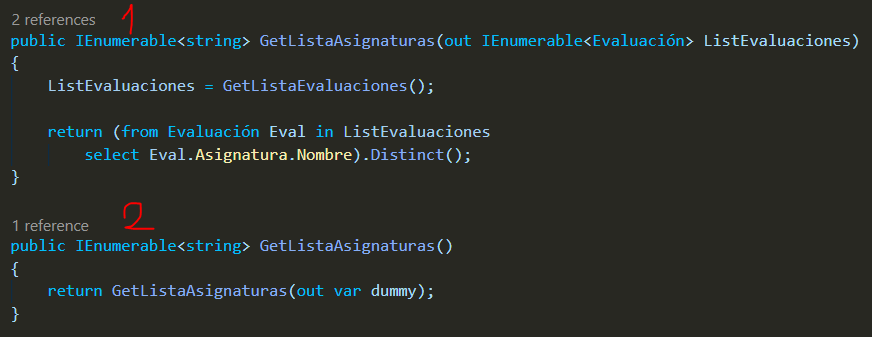
En el llamado del método quedaría de la siguiente forma.



**Sobrecarga de Métodos**

Sin embargo al crear un método del que no siempre podría ser necesario este parámetro de salida es necesario realizar la sobrecarga del método que es la creación del mismo sin que me pida el parámetro de salida obligatoriamente.

Para ello creo el mismo método sin el parámetro de salida y en ***return*** invoco el método que me pide el parámetro de salida pero le envió una variable **dummy**. Quedaría de la siguiente manera:



**Finally en los Try Catch.**

