Resumen clase a clase

Programación y Laboratorio II

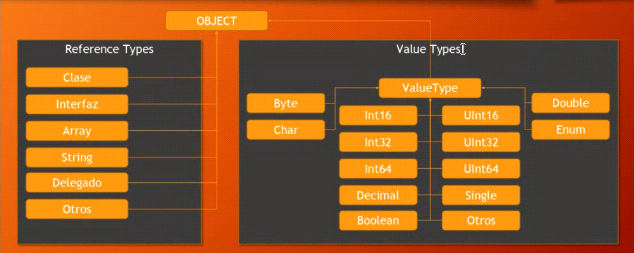
**Clase 1**

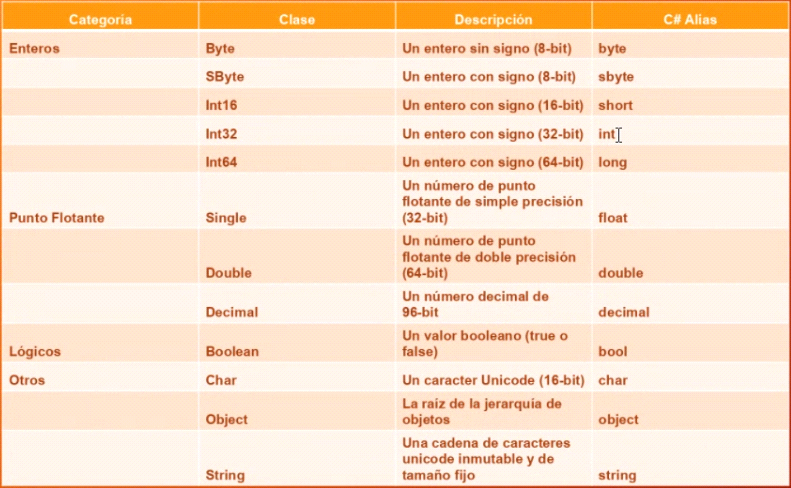
Dentro de este framework (.NET), hay una parte que define los tipos de datos (**Common Type System -CTS-**), que nos va a garantizar que indistintamente del lenguaje que usemos, si el framework es el mismo, entonces los tipos de datos van a ser exactamente iguales.

Aquí, nosotros vamos a utilizar el compilador de C#, pero siempre sobre esta **misma base de datos comunes**.

En el CTS se definen tipos de **Valor**, y también tipos de **Referencia**.

Al mismo tiempo, todo tipo hereda directa o indirectamente del tipo System.Object





**Valores predeterminados (Hacen referencia a atributos dentro de una clase)**

**Enteros**: 0 (cero).

**Punto** **flotante**: 0 (cero).

**Lógicos**: False.

**Referencias**: NULL.

**Conversiones básicas**

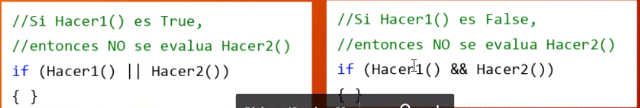
Va a haber dos tipos (que se originan obviamente cuando origen y destino son tipos de datos diferentes)

**Implícitas**: Lo genera automáticamente el compilador, y se da cuando el tipo de dato que obtiene el receptor es más grande que el que se le esta asignando (Por ejemplo, cuando le asignamos un INT a una variable FLOAT).

**Explicitas**: Esto tenemos que generarlo manualmente, y se da justamente cuando el tipo de dato receptor es más chico que aquel que se le esta asignando (Por ejemplo, cuando le asignamos un FLOAT a una variable INT). Para hacer esto, debemos castear el tipo de dato receptor (teniendo en cuenta que se generara una pérdida de información), por lo cual la sintaxis quedaría de la siguiente manera:

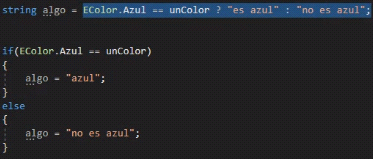
**int entero = (int) 15.2;**

**Tipos de Operadores**

En C# todas las evaluaciones se hacen por “cortocircuito”, por lo cual:

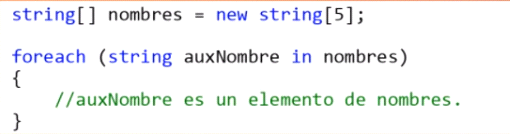
**Sentencias condicionales**

El **if** y **else** actúan de la misma manera que como lo hacía en C. (abajo imagen con ejemplo de escritura diferente)



El **switch** actúa de la misma manera, pero ahora nos obliga a utilizar un **default**.

El **for** actúa de la misma manera.

Ahora aparece el **foreach**, que hace básicamente lo mismo que el **for**, pero no utiliza índices (al menos no declarados por el programador). La sintaxis, por ejemplo, seria de la siguiente manera:

El **while** y el **do** **while**, ahora tienen que tener una condición booleana necesariamente, es decir, ya no se evalúa por 1 y 0, sino por true o false.

**Console**

Es Una clase publica y estática, que nos va a permitir la comunicación con el exterior de nuestra aplicación. Por esto mismo, representa la entrada, salida, y errores de Streams para aplicaciones de consola.

Hay que tener en cuenta que es miembro del Namespace: System.

Dentro de esto, tenemos diferentes métodos, por ejemplo:

Clear(): Que sirve para limpiar la consola.

Read(): Lee el próximo carácter del stream de entrada y devuelve el ASCII.

ReadLine(): Lee la siguiente línea de caracteres completa de la consola, devuelve un string.

Write(): Es como un printf sin salto de línea.

WriteLine(): Es un printf pero con salto de línea incluido.

Y también diferentes propiedades, como:

Title: Para cambiarle el título a la aplicación

BackgroundColor:

**Formato de salida de texto**

Voy a poder darle formato a ese texto que estoy mostrando, por ejemplo:

**Formato completo -> {N [, M][: Formato]}** *(\*)*

**N** será el número del parámetro, empezando por cero.

**M** será el ancho usado para mostrar el parámetro, el cual se rellenará con espacios. Si M es negativo, se justificará a la izquierda, y si es positivo, se justificará a la derecha.

**Formato** será una cadena que indicará un formato extra a usar con ese parámetro.

*(\*) aquellos que están entre [] son opcionales, mientras que la N es obligatoria*

**CLASE 2**

**Programación Orientada a Objetos**

Los conceptos de este paradigma son comunes a todos los diferentes lenguajes que lo utilicen, más la diferencia radica en la sintaxis específica de cada uno de estos lenguajes.

Lo que propone este paradigma, es resolver problemas de la realidad a través de la identificación de objetos (las cosas de la realidad) y de las relaciones de colaboración entre ellos.

El objeto va a ser nuestro elemento principal dentro de este paradigma, y van a estar intercomunicados a través de mensajes (que serían los métodos).

**Pilares de la P.O.O.**

**Abstracción**: lo que propone este concepto es que de un objeto que yo vaya a evaluar para codificarlo, tratar de enfocarme en las características más importantes que tiene ese objeto (descartando las que no lo son). Se podría traducir como una ignorancia selectiva, puesto que voy a ignorar adrede algunos detalles del objeto, y me voy a concentrar en sus características principales. Para reforzar esta abstracción, se utiliza el concepto de encapsulamiento.

**Encapsulamiento**: Básicamente, es tratar de encerrar ciertas funcionalidades de mi clase para que no queden expuestas para fuera de la clase, lo cual haremos a través de funciones o propiedades (este último es particular de C#). Entonces, el exterior de la clase la va a ver como una caja negra, debido a que, a través del encapsulamiento, el interior no queda expuesto. La idea es brindarle al usuario de nuestra clase distintas funciones y métodos para que resuelvan cosas, pero sin que el mismo tenga el acceso al interior de esas funciones.

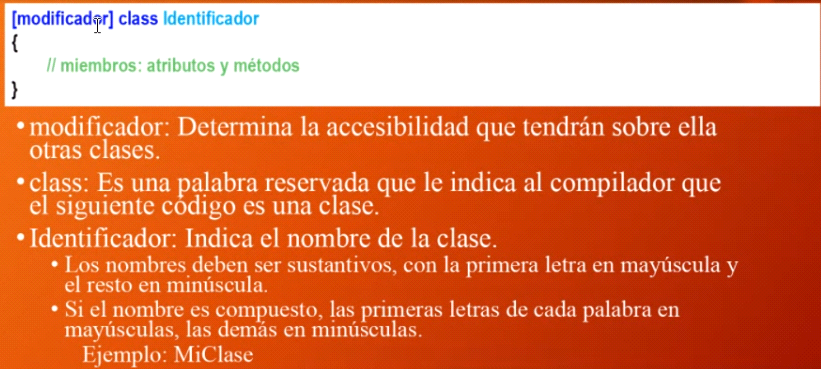
**Herencia**: La idea básica de la misma, es que yo pueda como programador, diseñar una clase con todas sus características, y al mismo tiempo, tener la capacidad de generar nuevo código en base a ese código ya generado. Esto quiere decir, que voy a tener disponible en esta nueva clase que cree, todo el código que cree en la clase padre.

**Polimorfismo**: Es un concepto bastante abstracto, que tiene que ver con las relaciones de herencia entre clases, y la capacidad que van a tener nuestros métodos de poder modificar parcial o totalmente las implementaciones en las clases derivadas. La idea sería que yo pueda definir un método en la clase padre, que por herencia voy a poder utilizar en las clases derivadas, y que además yo pueda desde estas últimas modificar el código que definí en la clase padre. Por ejemplo, en la clase base FIGURA tengo el método Area(), y luego, en las clases derivadas CIRCULO, TRIANGULO y CUADRADO, podre modificar ese mismo método Area() según la necesidad de cada una de estas clases hijas.

**¿Qué es una clase?**

Básicamente es tomar de la vida real cualquier objeto, y tratar de clasificarlo (utilizando el concepto de abstracción) para discriminar los aspectos fuertes de esa clase.

Esta clasificación la vamos a establecer en base a comportamientos (métodos, por ejemplo: Clase auto – Arrancar, frenar) y atributos comunes (cualidades, por ejemplo: Clase auto – Cantidad de ruedas, color), y gracias a esto vamos a poder generar un nuevo tipo de dato, que tendrá estas características.

La sintaxis para crear una clase es muy sencilla:

Variables = Atributos

Funciones = Métodos

**Modificador**: Va a determinar la accesibilidad que tendrán sobre ella otras clases. Dichos modificadores pueden ser

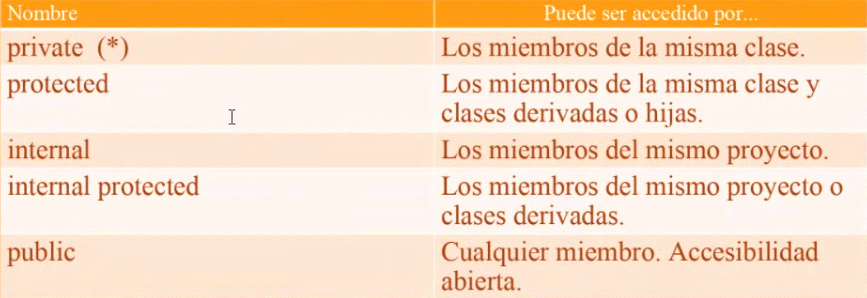
* Sealed: Indica que la clase no podrá heredar.
* Private: Accesor por defecto.
* Public: Accesible desde cualquier proyecto.
* Internal: Accesible en todo el proyecto (Assembly).
* Abstract: Indica que la clase no podrá instanciarse.

**Class**: Es una palabra reservada que le indica al compilador que el siguiente código es una clase.

**Identificador**: Indica el nombre de la clase, el cual debe ser un sustantivo, con la primera letra en mayúscula, y el resto en minúscula. Si el nombre es compuesto, las primeras letras de cada palabra en mayúscula, y las demás en minúscula. Por ejemplo: MiClase.

**Atributos**



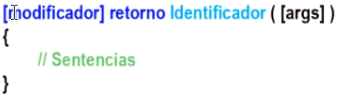
**Modificador**: Determina la accesibilidad que tendrán sobre él, las demás clases. Por defecto son private.

**Tipo**: Representa el tipo de dato. Ejemplo: int, float, etc.

**Identificador**: Indica el nombre del atributo

* Los nombres deben tener todas sus letras en minúscula
* Si el nombre es compuesto, la primera letra de la segunda palabra será mayúscula. Por ej: miNombre.

**Métodos**



**Modificador**: Determina la forma en que los métodos serán usados

**Retorno**: Es el tipo de valor devuelto por el método (solo retorna un único valor)

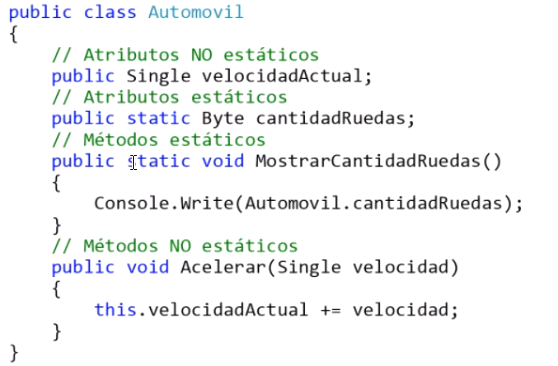
**Identificador**: Indica el nombre del método.

* Los nombres serán siempre verbos
* La primera letra de cada una de las palabras que lo componen debe ser mayúscula

**Args**: Representan una lista de variables cuyos valores son pasados al método para ser usados por este. Son opcionales y se definen de la siguiente manera: (tipo nombreArg, tipo nombreSegArg).  
Además, si hay más de un parámetro, serán separados por una coma entre ellos.  
Si un método no retorna ningún valor, se utilizará la palabra reservada void.



**Ejemplo general:**



**CLASE 3**

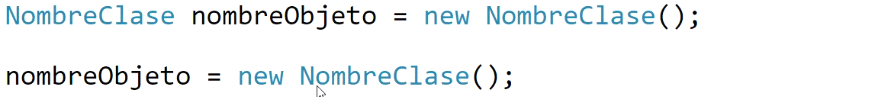
**Objetos**

Los objetos son clases instanciadas por lo cual quiere decir que se crean en tiempo de ejecución. Poseen un comportamiento (ejecutado a través de los métodos) y un estado (formado por sus atributos). Para acceder a los métodos o atributos se utiliza el . (punto) (para esto deben ser estáticos, y tener el modificador de visibilidad adecuado). Para crear un objeto se necesita la palabra reservada NEW.

En la programación orientada a objeto nosotros buscamos clasificar las cosas que intervienen en el problema, y abstraer las más importantes

**Si un método/atributo está definido como static, tengo que invocarlo con NombreClase.metodo/atributo. En cambio, si no está definido como static, entonces tengo que invocarlo con NombreObjeto.metodo/atributo.**

**Declarar e instanciar un objeto**



NombreClase: es el identificador de la clase o del tipo de objeto al que se referirá.

NombreObjeto: es el nombre asignado a la instancia de tipo NombreClase.

NombreClase(): es el constructor del objeto y no el tipo del objeto.

Una vez inicializado el objeto se puede utilizar para manipular sus atributos y métodos.

**Ciclo de vida de un objeto**

Para crear un objeto:

* Se usa el **new** para asignar memoria.
* Se usa un **constructor** para inicializar un objeto en esa memoria.

Para utilizar un objeto:

* Llamadas a **métodos** y **atributos**.

Destrucción del objeto:

* Se pierde la referencia en memoria, ya sea por la finalización del programa, cambio, o eliminación de la variable, etc.
* El **Garbage** **Collector** liberara memoria cuando lo crea necesario.

**Garbage Collector**

En .NET el Garbage Collector será el encargado de liberar memoria.  
Cada vez que creamos un nuevo objeto, el CLR (Common Lenguage Runtime) asigna memoria desde la poción gestionada (Heap)  
Eventualmente, el Garbage Collector liberará memoria de objetos sin referencia.

**La memoria y los Tipos de Datos**

El CLR administra dos segmentos de memoria: Stack (pila) y Heap (montón).  
El Stack es liberado automáticamente y el Heap es administrado por el Garbage Collector.  
Los tipos VALOR se almacenan en el Stack, y los tipos REFERENCIA se almacenan en el Heap.

Todas las definiciones de variables van a almacenarse en el Stack, pero según el tipo de dato, el contenido de esas variables puede ser una posición de memoria dentro del Heap (funciona como los punteros).

**Objetos VS Variables**

El tiempo de vida de una variable local está vinculado al ámbito en el que está declarada

* Tiempo de vida corto (en general).
* Creación y destrucción deterministas: Una variable local se crea en el momento de declararla, y se destruye al final del ámbito/scope donde se definió la misma. Son deterministas porque tienen lugar en momentos conocidos y fijos.

El tiempo de vida de un objeto

* Tiempo de vida más largo
* Destrucción no determinista: Un objeto aparece cuando se crea, pero no se destruye al final del ámbito en el que se crea. La creación de un objeto es determinista, pero su destrucción no. No es posible controlar exactamente cuándo se destruye y libera memoria para un objeto.

**Constructores**

Los constructores son métodos especiales que se utilizan para inicializar los objetos al momento de su creación. En C#, la única forma de crear un objeto es mediante el uso de la palabra reservada **new** para adquirir y asignar memoria.  
Aunque no se escriba ningún constructor, existe uno por defecto que se usa cuando se crea un objeto a partir de un tipo referencia.  
Los constructores llevan el mismo nombre de la clase.  
Lo único que **new** hace es adquirir memoria binaria sin inicializar, mientras que el único propósito de un constructor de instancia es inicializar la memoria y convertirla en un objeto que se pueda utilizar. Aunque new y los constructores de instancia realizan tareas independientes, un programador no puede emplearlos por separado. De esta forma, C# contribuye a garantizar que la memoria está siempre configurada para un valor valido antes de que se lea.

Hay dos tipos de constructores:

* Constructores de instancia: Que inicializan objetos (atributos NO estáticos).
* Constructores estáticos: que son los que inicializan clases (atributos estáticos). No deben llevar modificadores de acceso, utilizan la palabra reservada static, y no pueden recibir parámetros. Se invoca una única vez en toda la aplicación, y será cuando se instancia el primer objeto de mi clase.

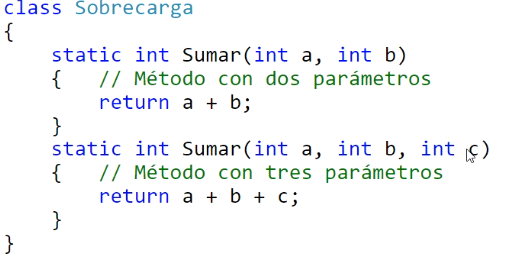
Hay un tipo especial de constructores, que son los constructores por defecto, los cuales:

* Son de acceso publico
* No tienen ningún tipo de retronó (ni siquiera void)
* No reciben ningún argumento
* Inicializan todos los campos a **cero**, **false, o null**.

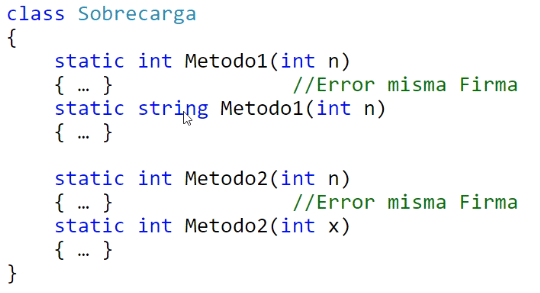
**Clase 4**

**Sobrecarga de Métodos**

Los métodos no pueden tener el mismo nombre que otros elementos en una misma clase (atributos, propiedades, etc.)  
Sin embargo, dos o más métodos en una clase si pueden compartir el mismo nombre, lo cual se conoce con el nombre de **sobrecarga**.  
Los métodos se sobrecargan cambiando el número, el tipo, y el orden de los parámetros (es decir, se cambia la firma del método).  
El compilador de C# distingue métodos sobrecargados comparando las listas de parámetros, por lo cual, según los parámetros que le pasemos a la hora de invocarlo, sabrá que método debe utilizar.



Por esto, las firmas de los métodos deben ser únicas dentro de una clase. No necesariamente con los tipos de datos, pero si con el orden de los mismos en caso de que se repitan.



Forman la definición de la firma de un método:

* Nombre del método
* Tipo de parámetros
* Cantidad de parámetros
* Modificador de parámetro (out o ref)

NO forman la definición de la firma de un método:

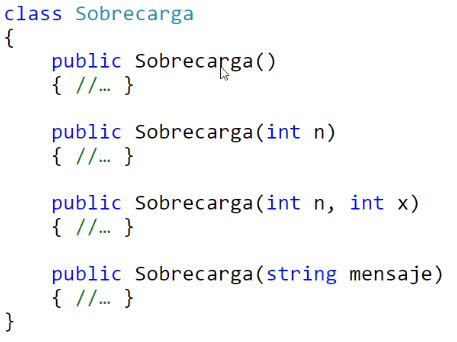
* Nombres de parámetros.
* Tipo de retorno del método.

**Uso de métodos sobrecargados**

Si hay métodos similares que requieren parámetros diferentes o si se quiere añadir funcionalidad al código existente, pero hay que tener en cuenta que son difíciles de depurar y también más difíciles de mantener.

**Sobrecarga de constructores**

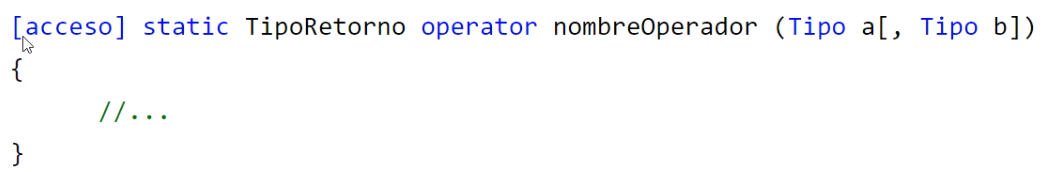
Al igual que los métodos, los constructores también se pueden sobrecargar, y las normas para hacerlo con las mismas.  
Se suele hacer cuando se quiere dar la posibilidad de instanciar objetos de formas diferentes.



Los constructores estáticos, como no reciben parámetros, obviamente no se pueden sobrecargar (debido a que no se puede modificar la firma si no es con el nombre).

**Clase 5**

**Sobrecarga de operador**

Sobrecargar un operador consiste en modificar su comportamiento cuando este se utiliza con una determinada clase, a través del formato

Nota: El modificador de acceso no podrá ser de un ámbito mayor que el de la clase.

Los operadores que vamos a poder sobrecargar son los siguientes:



En el caso de los operadores de comparación, si los sobrecargo, voy a tener que sobrecargar su par. Por ejemplo, si sobrecargo ==, entonces también !=, si sobrecargo <, entonces también >, etc.

Aquellos que no puedo sobrecargar, son los siguientes:



Con algunas excepciones:

* Indexador de array: Puedo definir nuevos indexadores
* Casting: Puedo definir nuevos operadores de conversión
* Asignación: El operador +=, por ejemplo, es evaluado usando el operador +, el cual puede ser sobrecargado

**Clase N6**

**Formularios**

Windows Forms es la plataforma de desarrollo de aplicaciones para Windows, basadas en el marco .NET. Este marco de trabajo proporciona un conjunto de clases que permite desarrollar complejas aplicaciones para Windows.

Un formulario, o plantilla, actúa como interfaz principal para el usuario local de Windows. Este formulario puede ser una ventana estándar, interfaces de múltiples documentos (MDI), cuadros de dialogo, de búsqueda, etc.

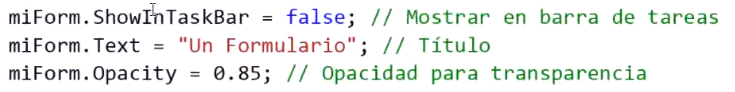
Estos formularios son objetos que exponen propiedades, métodos que definen su comportamiento y **eventos** que definen la interacción con el usuario.

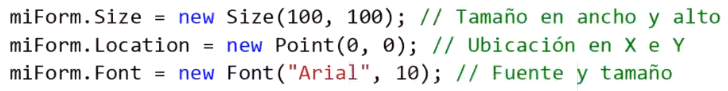
El concepto de Partial Class, que se incorpora en .NET 2.0, permite separar el código de una clase en dos archivos fuentes diferentes. El diseñador de los formularios utiliza esta técnica para escribir en un archivo aparte todo el código que el mismo genera. Gracias a esto, se puede organizar más claramente el código, manteniendo separada la lógica de la aplicación en un archivo diferente.

Una vez dentro del editor de texto, hay dos maneras de diseñar el formulario: Graficamente, a través de la vista de diseño del formulario, o A través del código.

**Objeto Form**

El objeto Form es el principal componente de una aplicación Windows.  
Algunas de sus propiedades admiten valores de alguno de los tipos nativos (por valor) de .NET



Otras propiedades requieren la asignación de objetos

Al mismo tiempo, los formularios tienen muchísimos métodos, y entre los más usados se encuentran

* Show ()
* ShowDialog()
* Close()
* Hide()

**Ciclo de Vida**

Muchos de los eventos a los que responde el objeto Form pertenecen al ciclo de vida del formulario, lo cual nos permite poder ejecutar determinadas acciones en determinado momento del ciclo de vida del mismo.

**New**: Constructor, inicialización de variables, etc.  
**Load**: Este evento se utiliza para setear algunos valores, títulos, cambiar combobox, colores, etc.  
**Paint**: Dibujara los controles que ya están cargados en el formulario  
**Activated**: Cada vez que el formulario recibe foco  
**FormClosing**: Permite cancelar el cierre del mismo (Por ejemplo la ventana de “Tiene cambios sin guardar, seguro que desea cerrar?”.  
**FormClosed**: Acá el formulario empieza a ser invisible nuevamente.  
**Disposed**: Se destruye el usuario, donde se liberan los recursos

Los últimos dos eventos se ejecutan una única vez a lo largo del ciclo de vida de la App, mientras que todos los demás, se pueden ejecutar múltiples veces.

Hay diferentes formas de atrapar un evento.

**MessageBox**

Para mostrar información o pedir intervención del usuario, es posible utilizar la clase MessageBox.  
Esta clase contiene métodos estáticos que permiten mostrar un cuadro de mensaje para interactuar con el usuario de la aplicación  
Los parámetros se especifican a través de enumerados que facilitan la legibilidad del código, por ejemplo: MessageBoxButtons.AbortRetryIgnore  
MessageBox.Show

**Tipos de formularios**

**MDI:** Básicamente es un formulario que permite contener otros formularios. Los formularios que puede contener son de distintos tipos

* Normal (.show): Funciona como una ventana individual, puede perder foco y salir del rango del padre
* Contenido (mdi parent form ppal): No pierde foco, pero no puede salir de adentro del padre
* Propietario (mdi owner form ppal): No pierde foco, pero puede salir de adentro del padre
* Dialogo o Modal: No permite accionar otra ventana mientras este activo

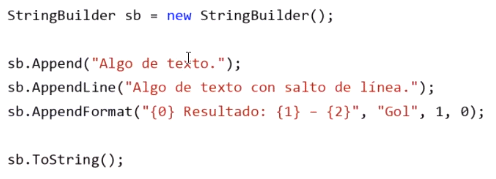
**Arrays y Strings**

Ahora, los **arrays** van a ser objetos que van a derivar de la clase Array. Necesito primero declarar la variable, instanciarlo a partir de su constructor con la palabra reservada new, y también va a tener propiedades y métodos como todo objeto.

Como esto ya es un objeto, y no solo una posición de memoria, vamos a tener que instanciarlo. Esta manera de instanciar es diferente a como lo hacíamos antes. Se sigue manteniendo que el primer elemento va a ser el índice 0, pero cambia la forma de generar esos objetos.



También, los Strings tienen métodos que podemos utilizar, debido a que es otro objeto. Además de todos ellos, encontramos clases diseñadas para la construcción de cadenas, como **StringBuilder**: que me permite construir cadenas de texto de manera más fácil.



**Clase 7**

**Enumerados**



Click derecho en el proyecto, y después agregar, nuevo elemento, archivo de código.

**Clase 8**

**Colecciones**

Se encuentran en otro Namespace, y es una clase publica y estática.

No genéricas: Me permiten pasarles cualquier tipo de valor. Hay distintos tipos de colecciones, como:

* LIFO (Last In, First Out) que se accede con .Stack;
* FIFO (First In, First Out) que se accede con .Queue;
* Listas Dinámicas que se accede desde ArrayList, tiene métodos para ordenar, como .Sort() y .Reverse()
* Hashtable que se accede desde .Hashtable, que se va a manejar siempre por clave y valor, en donde las claves funcionan como los índices de los arrays comunes.

Genericas: Me permiten generar las mismas colecciones que tenemos con las no genéricas (y son las que vienen por default incluidas, porque me garantizan que no sucedan problemas en tiempo de ejecución, sino solo en compilación) y a las cuales les tengo que indicar el tipo de dato que van a recibir

* Diccionario que se accede desde .Dictionary, es el equivalente al Hashtable, pero le tengo que especificar el tipo de clave y el tipo de valor (entre < x, y > al crear el objeto).
* Listas Genericas, que se accede desde .List, es el equivalente al ArrayList, pero le tengo que especificar el tipo de dato que va a contener (entre < x, y > al crear el objeto).
* Lo mismo sucede con .Stack y .Queue

Regla a seguir: Si tenemos que almacenar distintos valores bajo una misma variable, tenemos 3 opciones:

* Almacenarlo en un Array común
* Almacenarlo en un ArrayList (no genérico)
* Almacenarlo en un List (genérico) – (Esta es la que tenemos que tratar de utilizar).

**Clase 9**

**Herencia**

Es una relación **entre clases** en la cual una clase comparte la estructura y comportamiento definido en otra clase. Cada clase que hereda (o subclase) de otra posee:

* Los atributos de la clase base además de los propios
* Soporta todos o algunos de los métodos de la clase base

Existe para organizar mejor las clases que componen a una determinada realidad y poder agruparlas en funcione de sus atributos y comportamientos comunes, mientras que al mismo tiempo cada una se especializa según sus particularidades.

En el caso de .NET, se soporta solamente Herencia Simple (es decir, una clase derivada herede de una clase padre). Existe la Herencia Múltiple, pero no aplica en nuestra materia (C++ es un buen ejemplo tbh).

**Sintaxis**



modificadores: los de visibilidad y/o clase.  
class: Indica que el siguiente código es una clase.  
Operador (:): Le indica al compilador que es una subclase de la clase que precede al operador.  
NombreClaseBase: es el nombre de la clase padre.

**Miembros Heredados**

La clase derivada hereda siempre todo de su clase base, excepto los **constructores**.

Los miembros **públicos** se convierten implícitamente en miembros públicos de la clase derivada.

Los miembros **privados** se reciben como elementos heredados, pero no van a ser accesibles para la clase derivada.

A través de la palabra reservada base, puedo señalar que accederé a algo de la base padre (funciona igual que el **this**, solo que **this** es para lo propio, y **base** para lo de la clase padre).

**Modificador Protected**

El significado del modificador de acceso **protected** depende de la relación entre la clase que tiene el modificador y la clase que intenta acceder a los miembros que usan el modificador.

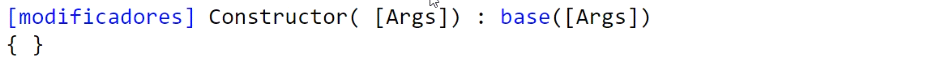
Para una clase derivada, la palabra reservada **protected** es equivalente a la palabra **public**.

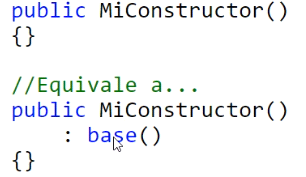
Entre dos clases que no tengan una relación base-derivada, por el contrario, los miembros protegidos de una clase se comportan como miembros **privados** para la otra clase.

Si declaro un miembro protected en la **clase derivada**, el mismo no será accesible por su base padre, pero si por las derivadas de esta derivada.

**Constructores: Base**

Para hacer una llamada a un constructor de la clase base desde un constructor de la clase derivada, se usa la palabra reservada **base**





**Constructores: Implicit**

Si la clase derivada no hace una llamada explicita a un constructor de la clase base, el compilador de C# usara implícitamente un constructor de la forma :base().

**Clases selladas**

Es un tipo de clases que me va a permitir a mi cortar la herencia (y es el único sentido que tiene).

La mayor parte de las clases son autónomas, y no están diseñadas para que otras clases deriven de ellas (aunque si heredan de otras).

Para que el programador pueda comunicar mejor sus intenciones al compilador y a otros programadores, C# permite declarar una clase como **Sealed** (sellada).

La derivación de una clase sellada no está permitida, por lo cual no se puede heredar de ella.



Un ejemplo de clase sellada es **System.String** (La cual esta sellada y, por tanto, no puedo crear una derivada de la misma).

**Puntos clave de la herencia**

La herencia permite crear nuevas clases más especializadas a partir de otras ya existentes más generales.

Las clases derivadas son versiones especializadas de las clases bases (Son del tipo de la clase base).

En .NET solo se admite HERENCIA SIMPLE

La herencia es **transitiva:** Si C hereda de B, y B hereda de A, entonces C también hereda de A (a excepción de los constructores).

Se hereda TODO menos los constructores y finalizadores

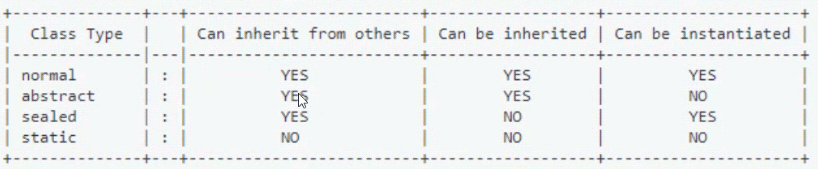
Los miembros **private** no son visibles en las clases derivadas (PERO SI SE HEREDAN).

Los miembros **protected** son accesibles desde todas las clases derivadas directa o indirectamente de la clase base, pero no desde otras clases que no heredan de ella.

Una clase derivada no puede ser más accesible que su clase base.

Si no se realiza una llamada explicita a un constructor de clase base, el compilador de C# proporciona automáticamente una llamada al constructor sin parámetros o predeterminado de la clase base

**Tipos de clases y herencias**



**Clase 10**

**Polimorfismo**

Ver cómo generar que un método o una propiedad definido en una clase, pueda cambiar su forma de manera parcial o total en tiempo de ejecución, según la clase que lo esté invocando.

Es la propiedad que tienen los objetos de permitir invocar genéricamente un comportamiento (método) cuya implementación será delegada al objeto correspondiente recién en tiempo de ejecución.

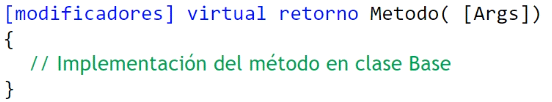
En otras palabras, es la capacidad de tratar objetos diferentes de la misma forma.

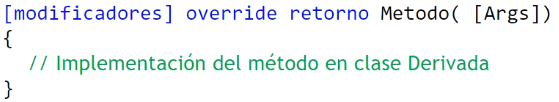
El polimorfismo tiende a existir en relación de herencia.

implica necesariamente la definición en una clase base y sobrescribirlos con nuevas implementaciones en clases derivadas.

**Sintaxis**

El polimorfismo implica la definición de métodos y/o propiedades en una clase base, mediante el uso de la palabra reservada **virtual** y sobrescribirlos con nuevas implementaciones en clases derivadas con la palabra reservada **override**.

La definición del método reside en la clase base (**se utiliza el *virtual***).

La implementación del método reside en la clase derivada (**se utiliza *override***).

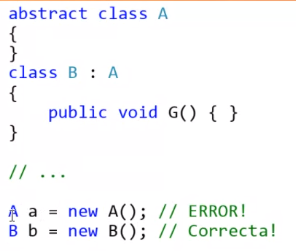
La invocación es resuelta en tiempo de ejecución.

**Clase 11**

**Abstracción**

Basicamente funciona para separar conceptualmente algo de algo. La principal característica de una clase abstracta es que no se puede instanciar (por ejemplo no puedo llamarla desde el main).

No se puede crear una instancia de una clase abstracta directamente, y es un error en tiempo de compilación utilizar el operador new en una clase abstracta.



**Clases abstractas**

*Basicamente se utiliza como base padre de todo un conjunto de derivadas, puesto que es un concepto abstracto que englobara a todos los demás objetos que yo derive de ese concepto en el resto de las clases.  
Por ejemplo: Vehículo seria abstracto (porque puede significar varias cosas) y contendrá todas las características comunes que luego las clases no abstractas Auto, Camión, Moto, etc. instanciaran.*

Aunque es posible tener variables y valores cuyos tipos en tiempo de compilación sean abstractos, tales variables y valores serán null o contendrán referencias a instancias de clases no abstractas derivadas de los tipos abstractos.

Se permite que una clase abstracta contenga miembros abstractos, aunque no es necesario.

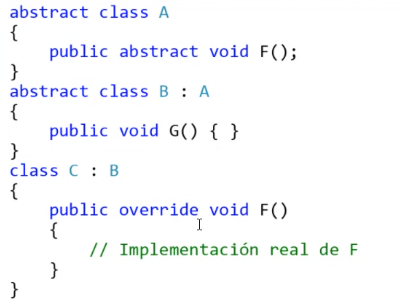
No se puede sellar una clase abstracta.

Si tengo una clase abstracta con métodos sin implementación o propiedades sin implementación, la primera clase derivada que no sea abstracta, está obligada a darles un valor.

Se sitúan siempre en la cima de la jerarquía de clases.

Establecen la estructura y significado del código.

Facilitan la creación de marcos de trabajo, esto es posible ya que las clases abstractas poseen una información y un comportamiento común a todas las clases derivadas de un marco de trabajo.



**Miembros Abstractos**

Cuando una declaración de método de instancia incluye un modificador abstact, se dice que el método es un método abstracto.

Las declaraciones de métodos abstractos solo se permiten en clases abstractas.

Aunque por definición formal, un método abstracto es también implícitamente un método virtual, no puede tener el modificador virtual (se le pone el modificador abstract).

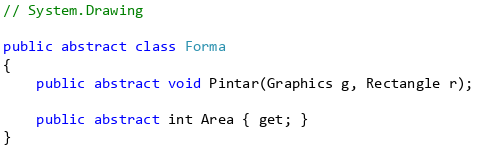
Por ende, podríamos decir:

- Una declaración de método abstracto introduce un nuevo método virtual, pero no proporciona una implementación del método.

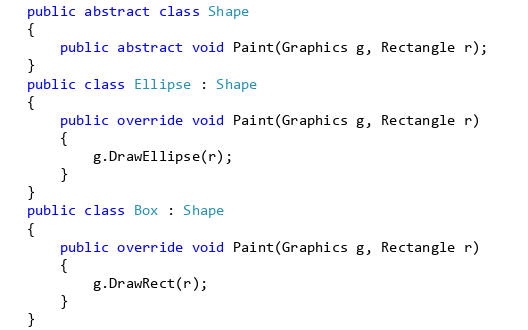
- Por esto es necesario que las clases derivadas no abstractas proporcionen su propia implementación mediante el reemplazo del método

- Debido a que un método abstracto no proporciona una implementación real, el cuerpo del método de un método abstracto consiste simplemente en un punto y coma.

- Son métodos y propiedades que se declaran sin implementación.



o bien:



**Clase 12**

**Indexadores**

En arrays: Aplica la lógica a la asignación de índices. Funciona como una propiedad, de la cual su sintaxis es

Public Objeto this[int índice]

{

Get{}

Set{}

}

**Clase 13**

**Gestión de errores**

Es una técnica que nos permite interceptar errores en **tiempo de ejecución**, esperados y no esperados.

En C# se controla por medio de excepciones.

Cuando se produce un error se ***lanza*** una excepción.

El programa debe construirse usando diferentes técnicas de gestión de errores para ***atrapar*** las excepciones y administrarlas de manera conveniente.

Las excepciones rompen el flujo de información lógico de nuestro programa (por lo cual dejara de funcionar); la idea es que nosotros lo atrapemos y generar en un bloque distinto la acción que nosotros creamos conveniente.

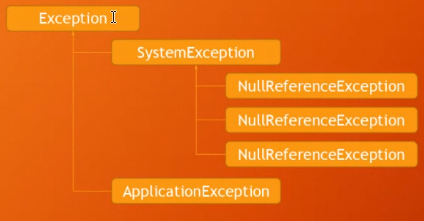
El programador debe habilitar su programa para que resuelva estos problemas.

**Objeto Exception**

Todas las excepciones derivan de la clase **Exception**, que es parte del runtime de lenguaje común (CLR).

**Ventajas**:

* Los mensajes de error no están representados por valores enteros o enumeraciones.
* Los valores enteros de programación como -3, desaparecen y en su lugar se utilizan clases concretas como OutOfMemoryException.
* Cada clase de excepción puede residir dentro de su propio archivo de origen y no está vinculada con las demás clases de excepción.
* Se generan mensajes de error significativos.
* Cada clase de excepción es descriptiva y representa un error concreto de forma clara y evidente.
* Cada clase de excepción contiene también información específica (por ejemplo una clase FIleNotFoundException podría contener el nombre del archivo no encontrado).



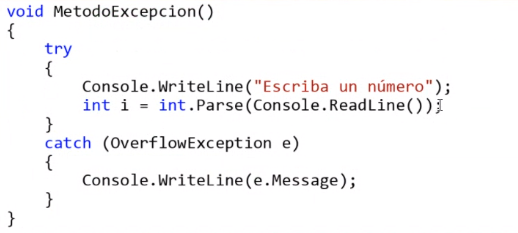
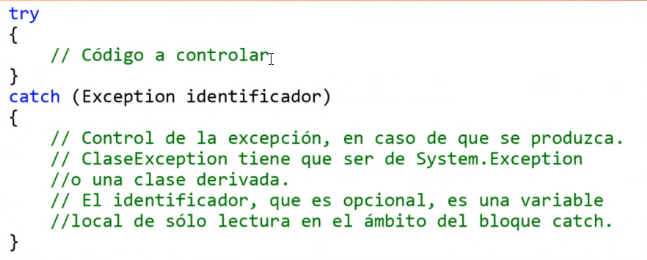
*Estos son los dos grupos grandes de excepciones (System y Application), No son todas null reference exception, es un error de copypaste.*

**Bloque Try-Catch**

Los bloques try catch son la solución que ofrece la orientación a objetos a los problemas de tratamiento de errores

La idea consiste en separar físicamente las instrucciones básicas del programa para el lujo de control normal de las instrucciones para tratamiento de errores.

Así, las partes del código que podrían lanzar excepciones se colocan en un bloque **try**, mientras que el código para el tratamiento de excepciones del bloque try se colocan en el bloque **catch**.



El bloque **try** contiene una expresión, que puede generar una excepción.

En caso de producirse una excepción, el runtime detiene la ejecución normal y empieza a buscar un bloque **catch** que pueda capturar la excepción pendiente (basándose en su tipo).

Si en la función inmediata no se encuentra el bloque **catch** adecuado, el runtime desenreda la pila de llamadas en busca de la función de llamada.

Si tampoco ahí se encuentra un bloque **catch** apropiado, busca la función que llamo a la función de llamada, y así sucesivamente hasta encontrar un bloque **catch** (o hasta llegar el final, en cuyo caso se cerrara el programa).

Con el método StackTrace, puedo ver la pila de llamadas que se ejecutó en búsqueda del catch (que es información para nosotros como programadores, no es representativa para los usuarios).

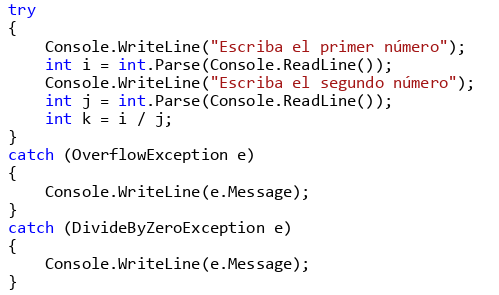
**Múltiples Catch**

Un bloque de código **try** puede contener muchas instrucciones, cada una de las cuales puede producir una o más clases diferentes de excepción.

Al haber muchas clases de excepciones distintas, es posible que haya muchos bloques **catch** y que cada uno de ellos capture un tipo específico de excepción.

La captura de una excepción se basa únicamente en su tipo.

El runtime captura automáticamente objetos excepción de un tipo concreto en un bloque **catch** de ese tipo.



**Catch genérico**

Un bloque **catch** general (de tipo Exception), puede capturar cualquier excepción independientemente de su clase (ya que todas heredan de este objeto), y se utiliza con frecuencia para capturar cualquier posible excepción que se pudiera producir por la falta de un controlador adecuado.

Un bloque try no puede tener más que un bloque catch general

En caso de existir, un bloque catch general debe ser el último bloque catch en el programa.

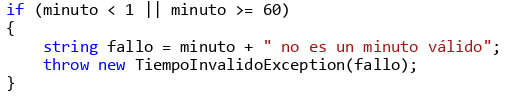
**Throw**

Cuando necesita lanzar una excepción, el runtime ejecuta una instrucción **throw** y lanza una excepción definida por el sistema.

Esto interrumpe inmediatamente la secuencia de ejecución normal del programa, y transfiere el control al primer bloque catch que pueda hacerse cargo de la excepción en función de su clase.

Es posible utilizar la instrucción **throw** para lanzar excepciones propias.

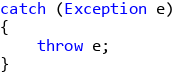
Pueden generar excepciones Common Language Runtime (CLR), .NET Framework, las bibliotecas de otros fabricantes, o el código de aplicación.



En el ejemplo de la imagen se emplea la instrucción **throw** para lanzar una excepción definida por el usuario TiempoInvalidoExcepcion, si el tiempo analizado no es válido.

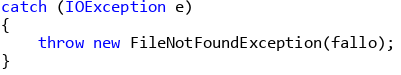
En general, las excepciones esperan como parámetro una cadena con un mensaje significativo que se puede mostrar o quedar registrado cuando se captura la excepción.

También es conveniente lanzar una clase adecuada de excepción.



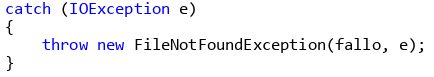
Solo es posible lanzar un objeto si el tipo de ese objeto deriva directa o indirectamente de **System.Exception**.

Se puede utilizar una instrucción **throw** en un bloque **catch** para volver a lanzar el mismo objeto excepción u otro nuevo.



En el ejemplo de la imagen, el objeto **IOException** y toda la información que contiene se pierde cuando la excepción se convierte en un objeto **FIleNotFoundException**.

Es más conveniente ajustar la excepción, añadiendo nueva información pero conservando la que tiene en la propiedad **InnerException**.



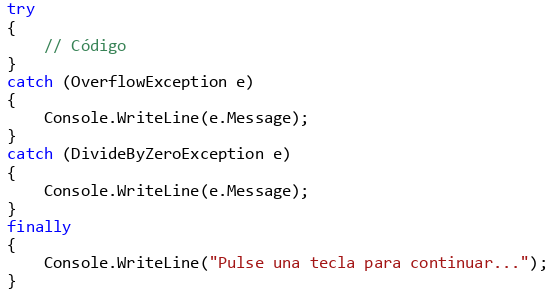
**Finally**

La cláusula **finally** de C# contiene un conjunto de instrucciones que es necesario ejecutar sea cual sea el flujo de control.

Las instrucciones del bloque **finally** se ejecutaran aunque el control abandone un bucle **try** como resultado de la ejecución normal porque el flujo de control llega al final del bloque **try**

Del mismo modo, también se ejecutarán las instrucciones del bloque **finally** si el control abandona un bucle **try** como resultado de una instrucción **throw** o una instrucción de salto como **break** o **continue.**

El bloque **finally** es útil en dos casos: para evitar la repetición de instrucciones y para liberar recursos tras el lanzamiento de una excepción.



*El bloque* ***try*** *se puede asociar a un* ***catch****, a un* ***finally****, o a un* ***catch*** *y a un* ***finally****. Estos últimos dos, nunca pueden estar solos.*

**Clase 14**

**Test Unitarios**

La idea del TU es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto.

Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

Generalmente quien crea los Test es un programador diferente al que genero el código, porque al ser una persona diferente, tendrá en cuenta otras cosas.

**Pruebas Integrales**

Son aquellas que se realizan en el ámbito del desarrollo de software una vez que se han aprobado las pruebas unitarias y lo que prueban es que todos los elementos unitarios que componen el software, funcionan juntos correctamente probándolos en grupo.

**Pruebas Funcionales**

Una prueba funcional es una prueba basada en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software

Las pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.

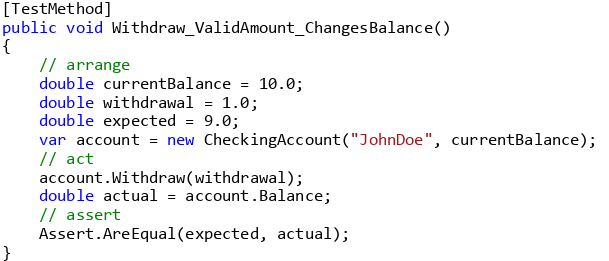
**Como crear un proyecto de TU**

1. Click derecho sobre la solución -> Agregar -> Nuevo
2. Menú lateral izquierdo, seleccionar Test
3. El proyecto será de tipo Test Unitario
4. Darle un nombre y aceptar en el cuadro de dialogo
5. Debo agregar la referencia de la Biblioteca de Clases que utilizare

**Patrón AAA**

El Patrón AAA (Arrange, Act, Assert) es una formula habitual de escribir pruebas unitarias para un método en pruebas

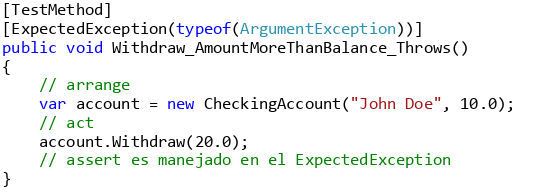
* La sección Arrange de un método de prueba unitaria inicializa objetos y establece el valor de los datos que se pasa al método en pruebas.
* La sección Act invoca al método en pruebas con los parametros organizados
* La sección Assert comprueba si la acción del método en pruebas se comporta de la forma prevista.



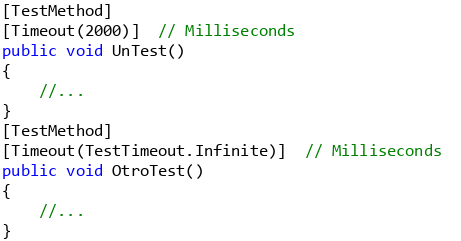
**Clase Assert (estática)**

Explicita para determinar si el método de prueba se supera o no, cumple su tarea a través de métodos estáticos, que analizan una condición True – False.

También puede ser manejado desde los atributos o etiquetas del método:



A través de estos atributos también se puede manejar un tiempo máximo para la prueba:



**Métodos de extensión**

Los métodos de extensión permiten agregar métodos a los tipos existentes sin necesidad de crear un nuevo tipo derivado y volver a compilar, o sin necesidad de modificar el tipo original.

Los métodos de extensión constituyen un tipo especial de método estático, pero se les llama como si se tratasen de métodos de instancia en el tipo extendido. En el caso del código de cliente escrito en C# y Visual, no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos realmente definidos en un tipo.

El primer parámetro de los métodos siempre es la clase a la cual voy a querer agregárselo.

**Clase 15**

**Tipos Genéricos**

Generics es el mecanismo de implementación de clases parametrizadas introducido en la versión 2.0 del lenguaje C#.

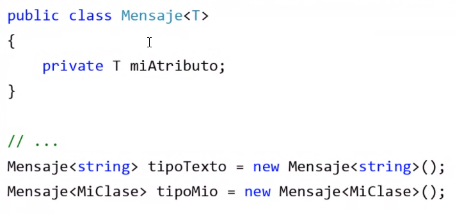
Una clase parametrizada es exactamente igual a una clase de las habituales, salvo por un pequeño detalle: su definición contiene algún elemento que depende de un parámetro que debe ser especificado en el momento de la declaración de un objeto de dicha clase.

Esto puede resultar extremadamente útil a la hora de programar clases genéricas, capaces de implementar un tipado fuerte sin necesidad de conocer a priori los tipos para los que serán utilizadas

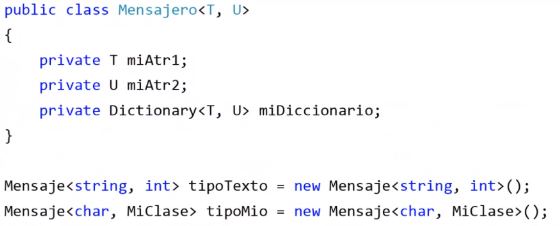
List y Dictionary, por ejemplo, es dos clases parametrizada:



**Uso simple de Generics**

****

**Uso menos simple de Generics**

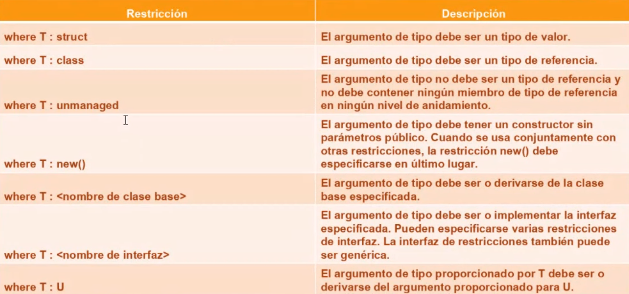
****

**Restricciones**

Una buena regla consiste en aplicar el mayor numero de restricciones posible que siga permitiendo manejar los tipos que se deben utilizar



Las restricciones más comunes son:



Algunas de las restricciones son mutuamente excluyentes.

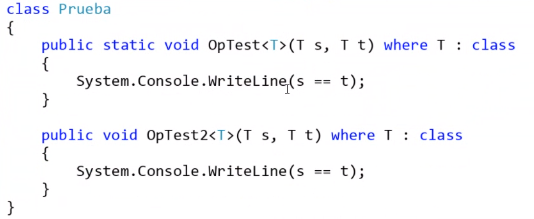
Todos los tipos de valor deben tener un constructor sin parametros accesible

La restricción struct implica la restricción new() y la restricción new() no se puede combinar con la restricción struct.

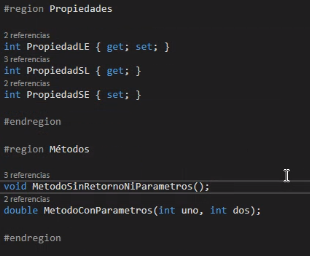
La restricción unmanaged implica la restricción struct.

La restricción unmanaged no se puede combinar con las restricciones struct o new()

**Métodos y Generics**

Puedo tener métodos que sean genéricos incluso en clases que no sean genéricas.

**Clase 16**

**Interfaces**

Es un contrato que establece una clase, en el cual esta clase asegura que implementara un conjunto de métodos.

Son una manera de describir que debería hacer una clase sin especificar el cómo.

Es la descripción de uno o más métodos y propiedades que posteriormente alguna clase debe implementar.

**Generalidades**

C# no permite especificar atributos en las interfaces

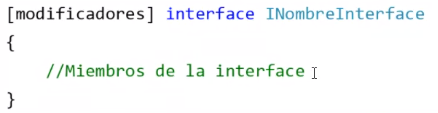
Todos los métodos y propiedades son públicos (no se permite especificarlo)

Todos los métodos y propiedades son como “abstractos” ya que no cuentan con implementación (no se permite especificarlo).

Las clases pueden implementar varias interfaces.

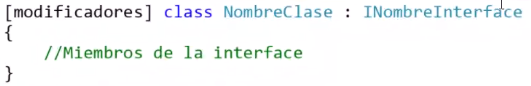
Las interfaces pueden “simular” algo parecido a la herencia múltiple.

**Sintaxis**



Se utiliza la palabra reservada interface

Por convención, los nombres de las interfaces comienzan con la letra I seguida del identificador.



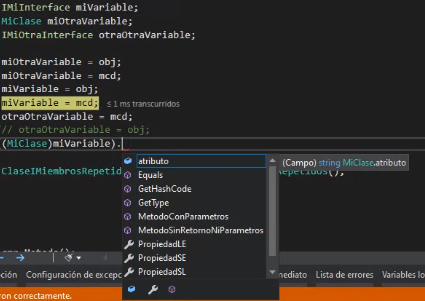
Para que una clase implemente una interface se emplea el operador dos puntos (:)

Para implementar una interface a una clase derivada, primero hay que indicar la clase base, luego la interface separadas por una coma (,).

Para sobrescribir los miembros de las interfaces NO se emplea la palabra override, ya que no fueron declaradas como virtual o abstract en la interface.



En caso de que mi clase, heredara de una clase, y también utilizara interfaces; la clase de la que hereda se pasa como primer parámetro, y luego se divide de las interfaces con una coma igual que antes.

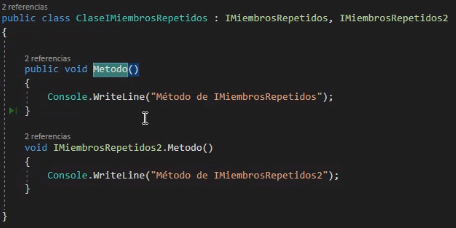
Yo puedo definir variables del tipo **Interfaz,** pero no voy a poder instancias objetos de tipo interfaz (porque no son objetos) pero si puedo asignarles a esas variables cualquier tipo de clase que implemente esa interfaz. Funciona igual que las clases abstractas, yo voy a tener acceso a los métodos y propiedades que sean de la **Interfaz** y que estén instanciados **en la Clase**, pero si quiero acceder a los métodos o propiedades que sean de la Clase particularmente (es decir que no estén definidos en la interfaz), debo generar un **casteo**.

Por ejemplo, en la imagen, como genero el casteo, puedo ver los métodos y propiedades definidos en la interfaz, pero también puedo ver el atributo definido en la clase MiClase, sobre la cual estoy generando el casteo.

Además, no puedo castear una interfaz a una clase que no la implemente, eso me daría un error.

**Implementación Explicita**

Los miembros implementados explícitamente sirven para ocultar la implementación de miembros de interfaces a las clases que lo implementan.



También sirven para evitar la ambigüedad cuando, por ejemplo, una clase implementa dos interfaces **que poseen un miembro con la misma firma.** Si hago esto, no puedo colocarle modificador de visibilidad a aquel método o propiedad al que le esté especificando la implementación explicita

Las clases derivadas de clases derivadas de clases que implementan interfaces de manera explícita, no pueden sobrescribir los métodos definidos explícitamente.

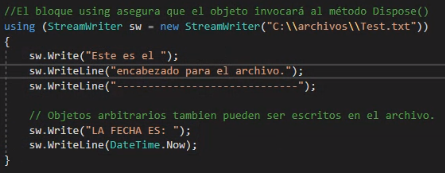
Sintácticamente, la implementación de una interfaz de manera explícita e implícita es igual, lo único que cambia es la firma del miembro en la clase que implementa la interfaz.

Si yo quiero acceder a estos métodos desde el main, dependerá de la firma de cada uno. Aquel que no tiene especificada la interfaz, se accede directamente con el nombre de un objeto, y luego .Metodo(). En cambio, aquel que si tiene especificada la interfaz de la que sale, como no tiene el modificador de visibilidad public (ya que sintácticamente no puedo ponerlo cuando especifico la interfaz), debo generar un casteo al nombre de la interfaz a la que quiero acceder, y de esa manera voy a tener acceso a los miembros definidos de esa interfaz en particular

**Clase 17**

**Archivos**

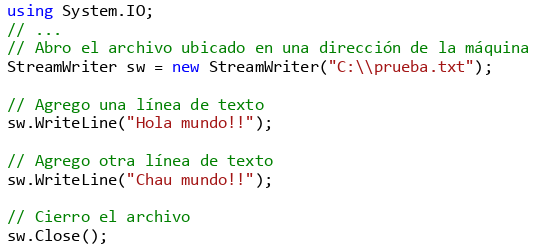
Siempre se maneja dentro de una estructura try catch, ya que es propenso a tirar excepciones. Se utiliza la clase StreamWriter, que escribe caracteres en archivos de texto. La clase StreamReader lee desde un archivo de texto. Ambas clases se encuentran en el Namespace System.IO.  
En lugar de utilizar el método close siempre, puedo manipular los archivos dentro de un bloque using, que se asegurara de liberar espacio al terminar de ejecutarse, y de esa manera se cerrara el archivo al terminar el bloque.



**StreamWriter**

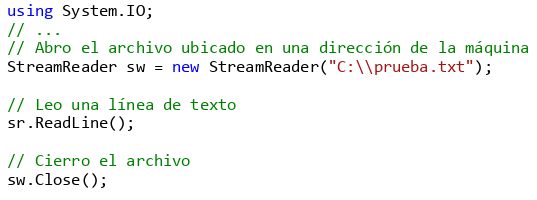
De los métodos más comunes de StreamWriter que voy a utilizar generalmente, tengo:

* StreamWriter(string path) -> Inicializa una nueva instancia de la clase StreamWriter, en un path especifico. Si el archivo existe, se sobrescribirá, sino se creará.
* StreamWriter(string path, bool append) -> Idem anterior, pero si el append es true, se agregarán datos al archivo existente. Caso contrario, se sobrescribirá el archivo.
* Write(string value) -> Escribe una cadena en el archivo sin provocar salto de línea.
* WriteLine(string value) -> Escribe una cadena en un archivo provocando salto de línea.
* Close() -> Cierra el objeto StreamWriter.



**StreamReader**

* StreamReader(string path) -> Inicializa una nueva instancia de la clase StreamReader. El path especifica de donde se leerán los datos.
* StreamReader(string path, Encoding e) -> Ídem anterior, dónde se le especifica el tipo de codificación que se utilizará para leer el archivo.
* Close() -> Cierra el objeto StreamReader.
* Read() -> Lee un carácter del stream y avanza carácter a carácter. Retorna un int (que es el valor en código ASCII).
* ReadLine() -> Lee una línea de caracteres del stream y lo retorna como un string.
* ReadToEnd() -> Lee todo el stream y lo retorna como una cadena de caracteres.



**Excepciones**

La ruta de acceso no es válida porque...  
- Es una cadena de longitud cero, contiene sólo espacios en blanco, o contiene caracteres no válidos. (ArgumentException).  
- La ruta de acceso es Null (ArgumentNullException).

File señala a una ruta de acceso que no existe.  
- FileNotFoundException  
- DirectoryNotFoundException

El archivo está en uso por otro proceso o hay un error de E/S  
- IOException

**Objetos Útiles**

**File.Exists(string path)**

* Es true si tiene los permisos necesarios y path contiene el nombre de un archivo existente; de lo contrario, es false.
* También devuelve false si path es null, una ruta de acceso no válida o una cadena de longitud cero.
* Si no tiene permisos suficientes para leer el archivo especificado, no se produce ninguna excepción y el método devuelve false, independientemente de la existencia del path.

**File.Copy(string path, string path):**

* Copia un archivo existente en un archivo nuevo. No se permite sobrescribir un archivo del mismo nombre.

**File.Delete(string path):**

* Elimina el archivo especificado.

**Directory.Delete(string path)**

* Elimina el directorio especificado, siempre y cuando esté vacío.

**Directory.Delete(string, boolean):**

* Elimina el directorio especificado y, si está indicado, los subdirectorios y archivos que contiene.

**Directory.Exists(string):**

* Determina si la ruta de acceso dada hace referencia a un directorio existente en el disco.

**GetFiles(String):**

* Devuelve los nombres de archivo (con sus rutas de acceso) del directorio especificado.

**SpecialFolder**

Es un enumerado que contiene todos los path especiales de Windows (Mis documentos, Mis imágenes, Mis videos, etc), por lo que nos sirve para poder ejecutar aplicaciones a las que no le pasemos literalmente la dirección, sino que el programa detecte la ruta del archivo. Esto se hace a través del método GetFolderPath de Environment, y luego pasándole como parámetro Environment.SpecialFolder.CarpetaDeseada.



**Serialización**

Es el proceso de convertir un objeto en memoria en una secuencia lineal de bytes. Basicamente es sacarle una foto a un **objeto** (nunca a un método).

Sirve para pasarlo a otro proceso, otra máquina, grabarlo en un disco, o grabarlo en una base de datos.

**Formatters**

Controlan el formato de la serialización.

* Serialización a XML (Es super estándar y estricto, por lo cual la mayor parte de los lenguajes conocerán esta serialización)
  + Por defecto incluye solo las propiedades y atributos públicos.
* Serialización Binaria
  + Por defecto incluyen todos los atributos y propiedades, ya sean públicas o privadas.

Luego de esto se reconstruye el objeto mediante la Deserialización (que es el proceso inverso), que puede ser en el mismo proceso o no, o en la misma maquina o no.

**Serialización XML**

La serialización XML solo serializa los atributos públicos y los valores de propiedad de un objeto en una secuencia XML (o también si tiene una visibilidad inferior al público pero es una colección y tiene una propiedad de solo lectura)

La serialización XML no convierte los métodos, indexadores, atributos privados, ni propiedades de solo lectura (salvo colecciones de solo lectura).

Tiene que estar definido el constructor por default sí o sí.

La clase que yo quiera serializar tiene que ser publica sí o sí.

La clase central de la serialización XML es **XmlSerializer** y sus métodos más importantes son Serialize y Deserialize.

La secuencia XML que genera XmlSerializer cumple con la recomendación 1.0 de W3C ([www.w3.org](http://www.w3.org)) acerca del lenguaje de definición de esquemas XML (XSD).

**Clase 18**

**Manejo de base de datos con SQL**

Para administrar Bases de Datos, las mismas tienen que estar guardadas en el directorio C:\Program Files\Microsoft SQL Server\MSSQL15.SQLEXPRESS\MSSQL\DATA

Varchar = es el que se usa generalmente para las cadenas, esta seguido de (int) que es el máximo de caracteres que permite, y además, se redimensiona para abajo automáticamente en caso de que no los utilice todos.

Edit top rows: me permite editarlos de manera grafica

Select top rows: me permite agregarle código a la bbdd

**Instrucciones comunes:**

--CUALQUIER INSTRUCCION SELECT PUEDE TRAER:

--TODOS, ALGUNOS, UNO, O NINGUN VALOR

--Generado automáticamente con Select top 1000 rows

SELECT TOP (1000) [id]

,[cadena]

,[entero]

,[flotante]

FROM [test].[dbo].[tabla\_uno]

--Esta instrucción es igual a la de arriba

select id, cadena, entero, flotante from tabla\_uno

--También se puede hacer para traer todos los campos

select \* from tabla\_uno

--Puedo generar condiciones con el comando where

--Y si necesito usar operadores lógicos, los escribo literalmente

select \* from tabla\_uno

where id > 1 and flotante < 60

--Las cadenas se comparan totalmente con el =, y siempre entre comillas simples

select \* from tabla\_uno

where cadena = 'otro valor'

--Puedo establecer condiciones en las cadenas con like y con %

--El like sirve para indicar que contenga lo que le paso entre las ''

--El % indica que no importa lo que tenga del lado que lo pongo (por ejemplo,

--si lo pongo al principio, no importa lo que tenga detrás)

select \* from tabla\_uno

where cadena like '%valor%'

--Le puedo dar un ordenamiento según el criterio que le pase

--desc(descendente) | asc(ascendente)

select \* from tabla\_uno

order by id desc

--Puedo poner más de un criterio de ordenamiento, siempre se ejecutará primero

--el que esta más cerca del order by, pero si hay un empate en esa comparación,

--se aplicará el ordenamiento del segundo criterio (pueden ser independientes los

--criterios, es decir, por ejemplo, uno asc y el otro desc)

select \* from tabla\_uno

order by cadena asc, entero desc, flotante asc

--También puedo combinar las restricciones y criterios

--Por ejemplo acá, todas las cadenas que contengan una 'o', ordenadas por el float

select \* from tabla\_uno

where cadena like '%o%'

order by flotante desc

/\*OTRA INSTRUCCION MUY USADA, ES LA INSTRUCCION INSERT

QUE ME PERMITE AGREGAR REGISTROS A LA BBDD

\*/

insert into tabla\_uno

(cadena, entero, flotante) --acá van las columnas a las que le agrego valores (en este caso todas porque no aceptan NULL)

values

('nuevo valor', 888, 99) --acá van los valores que les asigno a cada columna (obviamente no puedo poner más valores que columnas)

/\*OTRA INSTRUCCION MUY USADA, ES LA INSTRUCCION UPDATE

QUE ME VA A PERMITIR MODIFICAR LAS COLUMNAS DE UNO O VARIOS REGISTROS

\*/

--LO UNICO QUE NO PUEDO MODIFICAR, SON LOS CAMPOS MARCADOS COMO PRIMARY KEY (COMO EL ID)

update tabla\_uno

set cadena = 'valor modificado', entero = 0

where id = 4 --le agrego esta restricción porque si no lo aplicara a TODOS los registros

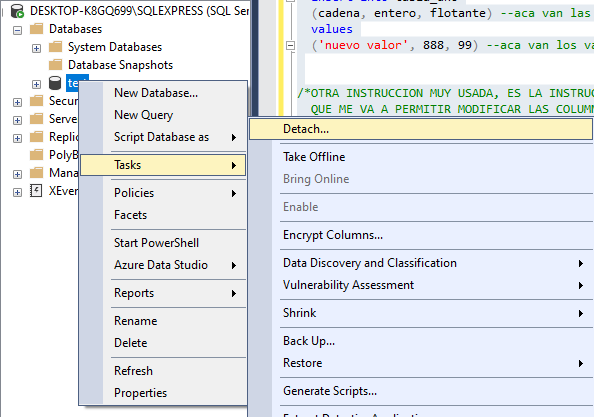
/\*OTRA INSTRUCCION MUY USADA, ES LA INSTRUCCION DELETE

QUE ME PERMITE ELIMINAR REGISTROS\*/

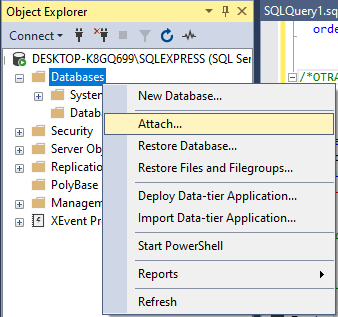
delete from tabla\_uno

where id = 5

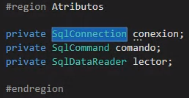
para poder pasar la BBDD hacia otra persona (y desadjuntarla de nuestro motor), tengo que utilizar el Detach:



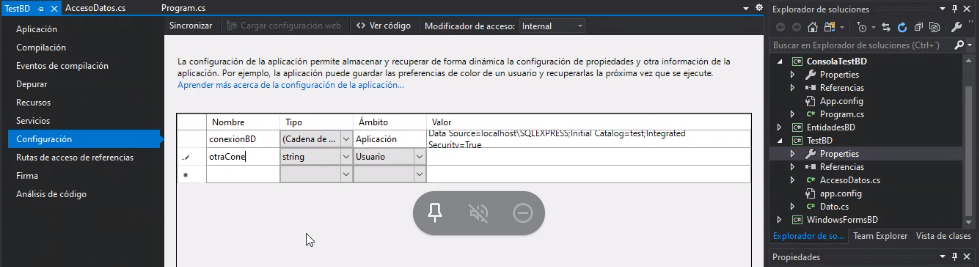
Luego, para adjuntarla nuevamente, tengo que utilizar el comando Attach:

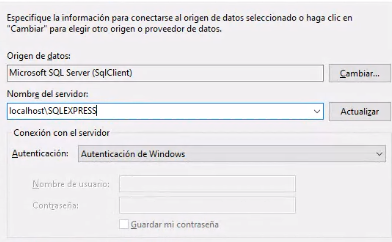


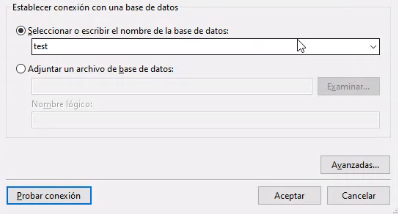
**Vinculación de BBDD a proyecto de C#**

Primero tengo que establecer una conexión con la BBDD. Para esto necesito objetos de tipo Sql (que se encuentran en las librerias System.Data y System.Data.SqlClient)

Para configurar la cadena de conexión:







**Delegados**

Lo que hace es acumular referencias/direcciones de memoria de funciones. Funciona básicamente como los punteros a funciones de C.

El delegado nunca ejecuta ninguna función, simplemente es invocado y va llamando a las funciones que comparten su firma (esta entre quien invoca, y el método que comparte la firma).

Para definirlo uso la palabra reservada **delegate** y debe ser accesible para las partes implicadas.



**Hilos**