CLASE 1: INTRODUCCION A C#.

Common Type System (CTS):
Define un conjunto común de "tipos" de datos orientados a objetos.
Todo lenguaje de programación .NET debe implementar los tipos definidos por el CTS.
Todo tipo hereda directa o indirectamente del tipo System.Object.
El CTS define tipos de VALOR y de REFERENCIA

Tipos de Datos

Las variables **escalares** son **constantes** o variable que contiene un **dato atómico y unidimensional**. Las variables **no escalares** son **array** (vector)**, lista** y **objeto**, que pueden tener almacenado en su estructura más de un valor.

Enteros: 0 (cero) // **Punto flotante**: 0 (cero) // **Lógicos**: False // **Referencias**: Null

Conversiones Básicas

Implícitas: No interviene el programador
```
float flotante = 15;
```

Explícitas: interviene el programador, ya que puede haber perdida de datos.
```
int entero = (int)15.2;
```

Entry Point

El punto de entrada para los programas en C# es la función **Main**

static: Es un modificador que permite ejecutar un método sin tener que instanciar a una variable (sin crear un objeto). El método Main() debe ser estático.

void: Indica el tipo de valor de retorno del método Main(). No necesariamente tiene que ser void.

string [] args: Es un Array de tipo string que puede recibir el método Main() como parámetro. Este parámetro es opcional.

Console

Es una clase pública y estática.
Representa la entrada, salida y errores de Streams para aplicaciones de consola.
Es miembro del NameSpace **System**.

Metodos

Clear() Limpia el buffer de la consola. Equivalente a **clrscr()** de C.
Read() Lee el próximo carácter del stream de entrada. Devuelve un entero.
ReadKey(bool) Obtiene el carácter presionado por el usuario. La tecla presionada puede mostrarse en la consola.
ReadLine() Lee la siguiente línea de caracteres de la consola. Devuelve un string.
Write() Escribe el string que se le pasa como parámetro a la salida estándar.
WriteLine() Ídem método Write, pero introduce un salto de línea al final de la cadena.
BackGroundColor: Obtiene o establece el color de fondo de la consola.
ForeGroundColor: Obtiene o establece el color del texto de la consola.
Title: Obtiene o establece el título de la consola.

Formato de salida de Texto: Con los marcadores ("{}"), además de indicar el número de parámetro que se usará, podemos indicar la forma en que se mostrará.
{ N [, M] [: Formato] } (*) **N** será el número del parámetro, empezando por cero **M** será el ancho usado para mostrar el parámetro. **Formato** será una cadena que indicará un formato extra a usar con ese parámetro.

CLASE 2: PROGRAMACION ORIENTADA A OBJETOS.

P.O.O. Es una manera de construir Software basada en un **nuevo paradigma**. Propone resolver problemas de la realidad a través de **identificar objetos** y relaciones de **colaboración** entre ellos. El ***Objeto*** y el ***Mensaje*** son sus elementos fundamentales.

PILARES: **ABSTRACCION / ENCAPSULAMIENTO / ENCAPSULAMIENTO / POLIMORFISMO**

ABSTRACCION: Ignorancia selectiva. Decide que es importante y que no. Se enfoca en lo que es importante. Ignora lo que no es importante. Utiliza el encapsulamiento para reforzarse.

ENCAPSULAMIENTO: Denota la capacidad del objeto de responder a peticiones mediante el uso de métodos o propiedades, sin exponer los medios utilizados para llegar a estos resultados. Desde fuera se ve como una caja negra.

HERENCIA: Es la relación entre clases. Va desde la generalización a la especialización. Clase padre, o base. Clase hija, o derivada.

POLIMORFISMO: La definición del método reside en la clase padre o base. La implementación, reside en la hija o derivada. La invocación se resuelve al momento de ejecución.

CLASES [modificador] class Identificador

Una clase es una clasificación. Se clasifican a base de **comportamientos** y **atributos** comunes. A partir de la clasificación, se crea un **vocabulario**. Es la **abstracción** de un objeto.

modificador: Determina la accesibilidad que tendrán sobre ella otras clases.
class: Es una palabra reservada que le indica al compilador que el siguiente código es una clase.
Identificador: Indica el nombre de la clase.

- Los nombres deben ser sustantivos, con la primera letra en mayúscula y el resto en minúscula.
- Si el nombre es compuesto, las primeras letras de cada palabra en mayúsculas, las demás en minúsculas.

Modificadores: **Abstract** (No podrá instanciarse) – **Internal** (Accesible en todo el proyecto) – **Public** (Accesible desde cualquier proyecto) – **Private** (Accesor por defecto) – **Sealed** (Indica que no se podrá heredar).

ATRIBUTOS [modificador] tipo identificador

modificador: Determina la accesibilidad que tendrán sobre él las demás clases. Por defecto son **private**.
tipo: Representa al tipo de dato. Ejemplo: int, float, etc.
Identificador: Indica el nombre del atributo.

- Los nombres deben tener todas sus letras en minúsculas.
- Si el nombre es compuesto, la primera letra de la segunda palabra estará en mayúsculas, las demás en minúsculas.

Modificadores: Private (Los miembros de la misma clase) – Protected (Solo por misma clase, y derivadas) – Internal (Solo mismo proyecto) – Internal proyected (Mismo proyecto y derivadas) – Public (Cualquier miembro, accesibilidad abierta).

METODOS [modificador] retorno Identificador ([args])

modificador: Determina la forma en que los métodos serán usados.
retorno: Es el tipo de valor devuelto por el método (sólo retornan un único valor).
Identificador: Indica el nombre del método.

- Los nombres deben ser verbos, con la primera letra en mayúscula y el resto en minúscula.
- Si el nombre es compuesto, las primeras letras de cada palabra en mayúsculas, las demás en minúsculas.

args: Representan una lista de variables cuyos valores son pasados al método para ser usados por este. Los corchetes indican que los parámetros son opcionales.

Los parámetros se definen como -> tipo identificador

- Si hay más de un parámetro, serán separados por una coma (,).
- Si un método no retorna ningún valor se usará la palabra reservada **void**.
- Para retornar algún valor del método se utilizará la palabra reservada **return**.

Modificadores: **Abstract** (Solo la firma del método, sin implementar) – **Extern** (Firma del método para métodos externos) – **Internal** (Accesible desde el mismo proyecto) – **Override** (Reemplaza la implementación del mismo método declarado como virtual) – **Public** (Accesible desde cualquier proyecto) – **Private** (Accesible SOLO desde la clase) – **Protected** (Accesible desde clase o derivadas) – **Static** (Indica que es un metodo de clase) – **Virtual** (Permite definir métodos sobre escritos en clases derivadas)

NAMESPACE

Es una **agrupación lógica de clases y otros elementos**. Toda clase está dentro de un NameSpace. Proporcionan un **marco** de **trabajo jerárquico** sobre el cuál se construye y organiza todo el código. Su función principal es la **organización del código** para reducir los conflictos entre nombres. Esto hace posible **utilizar en un mismo programa componentes de distinta procedencia**.

System.Console.WriteLine() **System** es el NameSpace de la BCL (Base Class Library) **Console** es una clase dentro del NameSpace System **WriteLine** es uno de los métodos de la clase Console

DIRECTIVAS

Son elementos que permiten a un programa **identificar** los **NameSpaces** que se usarán en el mismo. Permiten el uso de los miembros de un **NameSpace** sin tener que especificar un nombre completamente cualificado. C# posee dos directivas de NameSpace: **Using** **Alias**

USING:

- Permite la especificación de una llamada a un método sin el uso obligatorio de un nombre completamente cualificado.
- **using** System; //Directiva USING

ALIAS:

- Permite utilizar un nombre distinto para un Namespace.
- Generalmente se utiliza para abreviar nombres largos.
- **using** SC = System.Console; //Directiva ALIAS

CLASE 3: OBJETOS.

Los objetos son **clases instanciadas**. Se crean en tiempo de ejecución. Poseen Comportamiento (**métodos**) y Estado (**atributos**).
Para acceder a los métodos o atributos se utiliza el **. (punto).**
Para crear un objeto se necesita la palabra reservada **NEW**.

Sintaxis -> **NombreClase** nombreObjeto;

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 nombreObjeto = new **NombreClase**();
nombreObjeto = new **NombreClase**();

NombreClase(): Es el *constructor* del objeto y no el *tipo* de objeto.
Una vez inicializado el objeto se puede utilizar para manipular sus atributos y llamar a sus métodos.

CICLO DE VIDA:

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

Utilización del objeto
Llamadas a métodos y atributos.

Destrucción del objeto
Se pierde la referencia en memoria, ya sea por finalización del programa, cambio o eliminación de la variable, etc.
El **Garbage Collector** liberará memoria cuando lo crea necesario. Es el encargado de liberar memoria de objetos sin referencia.
Cada vez que creamos un objeto, el **CLR (Common Lenguage Runtime)** asigna memoria desde la porción gestionada (Heap).

LA MEMORIA Y LOS TIPOS DE DATOS

El **CLR** administra dos segmentos de memoria: **Stack** (Pila) y **Heap** (Montón).
Los tipos **VALOR** se almacenan en el **Stack**. Los tipos **REFERENCIA** se almacenan en el **Heap**.
El **Stack** es liberado automáticamente y el **Heap** es administrado por el **GC** (**Garbage Collector**).

VARIABLES:

Creación y destrucción deterministas: Se crea en el momento de declararla y se destruye al final del ámbito en el que está declarada. El punto inicial y el punto final de la vida del valor son **deterministas**, tienen lugar en momentos conocidos y fijos.

Tiempos de vida muy cortos por lo general: Un valor se declara en alguna parte de un método y no puede existir más allá de una llamada al método. Cuando un método devuelve un valor, lo que se devuelve es una copia del valor.

OBJETOS:

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

Tiempos de vida más largos: Puesto que el tiempo de vida de un objeto no está vinculado al método que lo crea, un objeto puede existir mucho más allá de una llamada al método.

CONSTANTES:

Una constante es otro tipo de campo. Contiene un valor que se asigna cuando se compila el programa y nunca cambia. Las constantes se declaran con la palabra clave **const**; son útiles para que el código sea más legible.

```
const int velocidadLimite = 90;
```

CONSTRUCTORES:

Los constructores son **métodos especiales** que se utilizan para **inicializar 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. Los constructores llevan el mismo nombre de la clase.

Lo único que **new** hace es adquirir memoria binaria sin inicializar, mientras que el solo propósito de un constructor de instancia es inicializar la memoria y convertirla en un objeto que se pueda utilizar. En particular, **new** no participa de ninguna manera en la inicialización y los constructores de instancia no realizan ninguna función en la adquisición de memoria.

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 válido antes de que se lea (a esto se le llama ***asignación definida***).

TIPOS DE CONSTRUCTORES:

Constructores de **instancia**: Que inicializan objetos (**atributos NO estáticos**).
Constructores **estáticos**: Que son los que inicializan clases (**atributos estáticos**).

CONSTRUCTORES POR DEFECTO:

Acceso público.
No tiene tipo de retorno (ni siquiera **void**).
No recibe ningún argumento.
Inicializa todos los campos a **cero**, **false** o **null.**

Si el constructor por defecto generado por el compilador no es adecuado, lo mejor es que escribamos nuestro propio constructor.

Podemos escribir un constructor que contenga únicamente el código necesario para inicializar campos con valores distintos de cero. Todos los campos que no estén inicializados en este constructor conservarán su inicialización predeterminada a **cero**, **false** o **null**.

Los constructores pueden recibir valores pasados como argumentos. El número de argumentos en principio no tiene límites.

CONSTRUCTORES ESTATICOS:

Son los encargados de inicializar clases.
Sólo inicializará los atributos estáticos.
No debe llevar modificadores de acceso.
Utilizan la palabra reservada **static**.
No pueden recibir parámetros.

CLASE 4: SOBRECARGA DE METODOS.

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 sí pueden compartir el mismo nombre. A esto se le da el nombre de **sobrecarga**.

Los métodos se sobrecargan cambiando el número, el tipo y el orden de los parámetros **(se cambia la firma del método).**
El compilador de C# distingue métodos sobrecargados **comparando las listas de parámetros**.

EJEMPLO: static int **Sumar**(**int** a, **int** b) static int **Sumar**(**float** a, **float** b)

Firmas de Métodos

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 afectan 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.
- Si se quiere añadir funcionalidad al código existente.
- Son difíciles de depurar.
- Son difíciles de mantener.

Sobrecarga de Constructores

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

Sobrecarga de Operadores

Sobrecargar un operador consiste en modificar su comportamiento cuando este se utiliza con una determinada clase.

[acceso] static TipoRetorno **operator** nombreOperador (**Tipo** a [, **Tipo** b])

(*) Nota: *Los operadores de Comparación, si son sobrecargados, se deben sobrecargar en pares; es decir, si se sobrecarga el operador ==, se deberá sobrecargar el operador !=.*

Operadores de conversión:

- Las conversiones definidas permiten hacer compatibles tipos que antes no lo eran.
- Los operadores de conversión pueden ser implícitos o explícitos.
- Los operadores de conversión explícitos son muy usados cuando se quiere que los usuarios estén conscientes que una conversión se llevará a cabo.

IMPLICITOS: **[acceso] static implicit operator** nombreTipo(**Tipo** a)
EXPLICITOS: **[acceso] static explicit operator** nombreTipo(**Tipo** a)

CLASE 6: GUI: Formularios

Windows Forms

Windows Forms es la plataforma de **desarrollo de aplicaciones** para Windows basadas en el marco .NET. Proporciona un conjunto de clases que permite desarrollar complejas aplicaciones para Windows.

Las clases se exponen en el NameSpace **System.Windows.Forms** y **NameSpaces** asociados. Es posible crear clases propias que hereden de algunas de las anteriores.

Formularios

Un formulario **Windows Forms** actúa como **interfaz** del usuario local de Windows. Pueden ser ventanas estándar, interfaces de múltiples documentos (MDI), cuadros de diálogo, etc. 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 formularios utiliza esta técnica para escribir en un archivo aparte todo el código que él mismo genera. Esto permite organizar más claramente el código, manteniendo separada la lógica de la aplicación en un archivo diferente.

Objeto Form -> es el principal componente de una aplicación Windows.

Show()	Visualiza el formulario. Puede especificarse su formulario Owner (dueño o propietario).
ShowDialog()	Puede utilizar este método para mostrar un cuadro de diálogo modal en la aplicación.
Close()	Cierra el formulario.
Hide()	Oculta el formulario del usuario.

Ciclo de Vida -> Muchos de los eventos a los que responde el objeto *Form* pertenecen al ciclo de vida del formulario.

New:	Se crea la instancia del formulario
Load	El formulario está en memoria, pero invisible
Paint	Se dibuja el formulario y sus controles
Activated	El formulario recibe foco
FormClosing	Permite cancelar el cierre
FormClosed	El formulario ya es invisible
Disposed	El objeto está siendo destruido

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**.

Propiedades

Name	Indica el nombre utilizado en el código para identificar el objeto.
BackColor	Indica el color de fondo del componente.
Cursor	Cursor que aparece al pasar el puntero por el control.
Enabled	Indica si el control esta habilitado.
Font	Fuente utilizada para mostrar el texto en el control.
ForeColor	Color utilizado para mostrar texto.
Locked	Determina si se puede mover o cambiar de tamaño el control.
Modifiers	Indica el nivel de visibilidad del objeto.
TabIndex	Determina el índice del orden de tabulación que ocupará este control.

Text Texto asociado al control.
Visible Determina si el control esta visible u oculto.

CONTROLES:

CheckBox Este control muestra una casilla de verificación, que podemos marcar para establecer un estado.

Generalmente el estado de un **CheckBox** es marcado (**verdadero**) o desmarcado (**falso**), sin embargo, podemos configurar el control para que sea detectado un tercer estado, que se denomina indeterminado, en el cual, el control se muestra con la marca en la casilla, pero en un color de tono gris.

RadioButton Los controles RadioButton nos permiten definir conjuntos de opciones auto excluyentes, de modo que situando varios controles de este tipo en un formulario, sólo podremos tener seleccionado uno en cada ocasión.

GroupBox Permite agrupar controles en su interior, tanto RadioButton como de otro tipo, se trata de un control contenedor.

ListBox Contiene una lista de valores, de los cuales, el usuario puede seleccionar uno o varios simultáneamente.

Items. Contiene la lista de valores que visualiza el control. Se trata de un tipo ListBox.ObjectCollection, de manera que el contenido de la lista puede ser tanto tipo cadena, numéricos u objetos de distintas clases.

SelectionMode. Establece el modo en el que se pueden seleccionar los elementos de la lista.
* **None**, no se realizará selección.
* **One**, permite seleccionar los valores uno a uno.
* **MultiSimple** permite seleccionar múltiples valores de la lista pero debemos seleccionarlos independientemente.
* **MultiExtended** nos posibilita la selección múltiple, con la ventaja de que podemos hacer clic en un valor, y arrastrar, seleccionando en la misma operación varios elementos de la lista.

ComboBox Control basado en la combinación (de ahí su nombre) de dos controles: **TextBox** y **ListBox**.

Un control **ComboBox** dispone de una zona de edición de texto y una lista de valores, que se pueden desplegar desde el cuadro de edición. El estilo de visualización por defecto de este control, muestra el cuadro de texto y la lista oculta, aunque mediante la propiedad **DropDownStyle** se puede cambiar dicho estilo.

La propiedad **DropDownStyle** también influye en una diferencia importante de comportamiento entre el estilo **DropDownList** y los demás, dado que cuando se crea un ComboBox con el mencionado estilo, el cuadro de texto sólo podrá mostrar información, no permitiendo que esta sea modificada.

En el caso de que la lista desplegable sea muy grande, mediante la propiedad **MaxDropDownItems**, se puede asignar el número de elementos máximo que mostrará la lista del control.

CLASE 6: ARRAYS Y STRINGS.

ARRAYS:

Un Array puede ser **unidimensional**, **multidimensional** o **anidado** (jagged). El valor por defecto de Array de elementos numéricos (value types) se establece en **cero**, mientras que los objetos y arrays anidados (reference types) se establece en **null**.

Los Arrays en C# son **base-cero**: un Array con 'n' elementos está indexado de **0 a n-1**. Los elementos de un Array pueden ser de **cualquier** tipo, incluso de tipo **Array (*Clase*).** Los Arrays son **reference type**, heredan de la clase *abstracta* System.Array. Implementan la *interfaz* **IEnumerable** por lo tanto se pueden iterar usando **foreach**.

ARRAYS UNIDIMENSIONALES:

[acceso] tipo**[]** identificador;
Identificador = new tipo[TAMAÑO];

ARRAYS MULTIDIMENSIONALES:

[acceso] tipo**[]** identificador;
Identificador = new tipo[FILAS, COLUMNAS];

En la declaración de un Array en C# los corchetes se colocan detrás del tipo y no detrás de la variable.

ARRAYS – METODOS Y PROPIEDADES:

CopyTo: Copia los elementos del Array unidimensional al Array unidimensional especificado, desde el índice indicado.
Resize: Redimenciona un Array.
GetLength: Obtiene el número de elementos para una dimensión determinada del Array.
GetValue/SetValue: Obtiene o establece el valor de uno o varios elementos del Array.
Initialize: Inicializa todos los elementos del Array, llamando al constructor por defecto.
Length: Obtiene la cantidad total de elementos de todas las dimensiones del Array.
Rank: Obtiene el número de dimensiones del Array.

STRINGS – METODOS:

Compare (de clase): Compara entre dos cadenas y devuelve tres valores posibles:
Concat (de clase): Concatena una o más cadenas.
Copy (de clase): Copia una cadena dentro de otra.
CompareTo (de instancia): Ídem método estático Compare().
Contains (de instancia): Indica si una cadena está o no contenida en el objeto actual.
CopyTo (de instancia): Ídem método estático Copy().
EndsWith/StartsWith (de instancia): Determina si el final/principio de la cadena contenida por el objeto y la pasada coinciden
IndexOf/LastIndexOf (de instancia): Devuelve el índice de la primera/última ocurrencia de la cadena especificada.
Insert (de instancia): Inserta una cadena a partir de una posición determinada.
Remove (de instancia): Borra todos los caracteres a partir del índice especificado.
Replace (de instancia): Reemplaza las ocurrencias de una cadena especificada en la instancia por otra cadena.
Substring (de instancia): Retorna una sub-cadena, a partir de un índice especificado con una determinada longitud.
ToLower/ToUpper (de instancia): Devuelve una copia de la cadena convertida a minúscula/mayúscula.
TrimEnd/TrimStart (de instancia): Remueve las ocurrencias de un conjunto de caracteres especificado en un Array
Trim (de instancia): Remueve todos los espacios en blanco del principio y del final de la instancia.

STRINGS – PROPIEDADES:

Chars: Obtiene el carácter situado en una posición de carácter especificada en el objeto System.String actual.
Length: Retorna la cantidad de caracteres que posee la instancia.

STRINGBUILDER: Representa una cadena de caracteres mutable. Esta clase no puede heredarse. Provee métodos y propiedades para operar con Strings. Mediante la propiedad **Capacity** podemos obtener o establecer el número máximo de

caracteres que puede contener la memoria asignada para una instancia en particular. Una vez superada esa capacidad, el especio utilizado en memoria se asignará dinámicamente (proceso más costoso).

CLASE 7: COLECCIONES

COLECCIONES GENERICAS:

Existen 2 métodos para agrupar colecciones:

Creación de matrices de objetos -> Muy útiles para crear / trabajar un numero fijo de objetos tipados.
Creación de colecciones de objetos -> Métodos más flexibles para trabajar. El grupo de objetos se puede agrandar y reducir dinámicamente dependiendo las necesidades.

Una **colección** es una **clase**. Antes de agregar elementos a esta colección, hay que **declararla**.

En una **colección genérica**, **ningún** otro **tipo de datos** se puede agregar en ella. Solo permite un tipo de dato deseado, y todos los elementos de la colección tendrán el mismo tipo de dato. Cuando se recupera un elemento no hay que determinar su tipo de dato.

```
List<string> palabras;
palabras = new List<string>();
palabras.add("Hola");
palabras.remove("Hola");
```

Se puede crear una colección genérica utilizando una de las clases en el espacio de nombres **System.Collections.Generic**.

Dictionary: - > Representa una colección de pares de clave y valor que se organizan por claves.

List -> Representa una lista de objetos que pueden ser obtenidos mediante un índice. Proporciona métodos para buscar, ordenar y modificar listas.

SortedList -> Representa una colección de pares de clave y valor que se ordenan por claves según la implementación de la interfaz **IComparer<T>** asociada.

Queue -> Representa una colección de objetos con el orden **primero** en entrar, **primero** en salir (FIFO).

Stack: -> Representa una colección de objetos con el orden **último** en entrar, **primero** en salir (LIFO).

COLECCIONES NO GENERICAS:

Son las incluidas en el espacio de nombres **System.Collections**. **No almacenan** los elementos como **objetos de un tipo específico**, sino como objetos de tipo **Object**. Siempre que sea posible, se deberían utilizar **las colecciones genéricas**.

ArrayList -> Representa una matriz de objetos cuyo tamaño aumenta dinámicamente según sea necesario.

Hashtable -> Representa una colección de pares de clave y valor que se organizan por código hash de la clave.

Queue -> Representa una colección de objetos con el orden **primero** en entrar, **primero** en salir (FIFO).

Stack -> Representa una colección de objetos con el orden **último** en entrar, **primero** en salir (LIFO).

Son las colecciones en el espacio de nombres **System.Collections.Concurrent**. Proporcionan operaciones eficaces y seguras para subprocesos con el fin de obtener acceso a los elementos de **colección** desde **varios subprocesos** (hilos).

Deben utilizarse en lugar de sus equivalentes en los espacios de nombres **System.Collections.Generic** y **System.Collections** cuando varios subprocesos tienen acceso a la colección simultáneamente.

NO TODAS LAS COLECCIONES TIENEN LAS MISMAS PROPIEDADES.

CLASE 8: ENCAPSULAMIENTO

Se denomina **encapsulamiento** al **ocultamiento** del estado, es decir, de los datos miembro de un objeto de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. Se mantiene ocultos los procesos internos, dándole al programador acceso sólo a lo que necesita. Cada objeto está **aislado del exterior**, **protegiendo** los datos asociados de un objeto contra su **modificación**.

Niveles de Encapsulamiento POO

Público: Todos pueden acceder a los datos o métodos de una clase que se definen con este nivel, este es el nivel más bajo.

Protegido: Podemos decir que estás no son de acceso público, solamente son accesibles dentro de su clase y por subclases.

Privado: En este nivel se puede declarar miembros accesibles sólo para la propia clase.

Propiedades

Una propiedad es un miembro que proporciona un mecanismo **flexible** para **leer**, **escribir** o **calcular** el valor de un campo. Se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados ***descriptores de acceso***. Permiten que una clase **exponga una manera pública** de obtener y establecer valores, a la vez que se **oculta el código** de implementación o validación.

Para devolver el valor de la propiedad se usa un descriptor de acceso de propiedad **get**
Para asignar un nuevo valor se emplea un descriptor de acceso de propiedad **set**.
La palabra clave **value** se usa para definir el valor que va a asignar el descriptor de acceso **set**.
Las propiedades pueden ser de lectura y escritura (**get** y **set**), de solo lectura (**get**) o de solo escritura (**set**).

```
private int totalGoles;
private int totalPartidos;

public int PromedioGol
{
    get
    {
        int prom = totalGoles / totalPartidos;
        return prom;
    }
}
```

```
class EjemploIndexadores
{
    // Declaro un array
    private string[] palabras = new string[100];

    // Defino el indexador
    public string this[int i]
    {
        get { return palabras[i]; }
        set { palabras[i] = value; }
    }
}
//...
EjemploIndexadores ejemplo = new EjemploIndexadores();
ejemplo[0] = "Hola";
ejemplo[1] = "Chau";
Console.WriteLine(ejemplo[0]);
```

Indexadores:

Los indexadores permiten a la instancia de una clase ser indexada tal cómo un array. No es necesario indexarlos sólo con un entero. La declaración de un indexador luce cómo una propiedad, sólo que este recibe parámetros y utiliza la palabra clave **this** para su definición

Enumerados

Son un conjunto propio de constantes con nombre. Estos tipos de datos permiten declarar un conjunto de nombres que definen todos los valores posibles que se pueden asignar a una variable. Por dentro, estas constantes están asociadas con el tipo de dato int. Normalmente es mejor definir un **enum** directamente dentro de un espacio de nombres para que todas las clases del espacio

de nombres puedan acceder a él **con igual comodidad**. Sin embargo, un **enum** también **se puede anidar dentro de una clase o struct**.

CLASE 9: HERENCIA

La herencia es una relación entre clases en la cual una clase comparte la **estructura y comportamiento** definido en otra clase.
Cada clase que hereda de otra posee:
* Los atributos de la clase base además de los propios.
* Soporta todos o algunos de los métodos de la clase base.
Una subclase hereda de una clase base.

El propósito principal de la herencia es el de **organizar mejor las clases** que componen una determinada realidad, y poder agruparlas en función de atributos y comportamientos comunes. A la vez que cada una se especializa según sus particularidades.
La herencia **permite crear nuevas clases a partir de otras ya existentes** (en lugar de crearlas partiendo de cero).
La clase en la que está basada la nueva clase se la conoce como ***clase base o padre***, la clase hija se conoce como ***clase derivada***.

Tipos de herencia:

Herencia Simple: Una **clase derivada** puede **heredar sólo de una clase base** (los lenguajes .NET soportan este tipo de herencia)
Herencia Múltiple: Una **clase derivada** puede **heredar de una o más clases** base (C++ soporta este tipo de herencia).

Una clase derivada hereda todo de su clase base, **excepto los constructores**.
Los miembros públicos de la clase base -> miembros públicos de la clase derivada.
Miembros privados -> Solo los miembros de la clase base pueden acceder a ellos.
Una clase derivada no puede ser más accesible que su clase base.

Palabra **protected**: Si son clases derivadas, es lo mismo que **public**, si no lo son, es lo mismo que **privated**. Cuando una clase derivada hereda un miembro **protected**, ese miembro también es implícitamente un miembro protegido.

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

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

* Una clase sin clases base explícitas extiende implícitamente la clase **System.Object**, que contiene un constructor público sin parámetros (por defecto).
* Si una **clase no contiene** ningún **constructor**, el compilador utilizará inmediatamente el constructor por "**defecto**".
* El compilador **no creará un constructor por defecto** si una clase tiene su **propio constructor** explícito.
* El compilador generará un **mensaje de error** si el constructor indicado **n** con ningún constructor de la clase base.

C# permite declarar una clase como ***sealed*** (sellada). La derivación de una clase sellada no está permitida. Por ejemplo, la clase **System.String**. Esta clase está sellada y, por tanto, ninguna otra clase puede derivar de ella.

RESUMEN Y PUNTOS CLAVE

- 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 base. (Son del tipo de la clase base).
- En **.NET** sólo se admite HERENCIA SIMPLE (Sólo se puede heredar de una clase).
- La herencia es **transitiva**: Si C hereda de B, y B hereda de A, entonces C también hereda de A.
- Se hereda **TODO** menos los **constructores** y **finalizadores**.
- Los miembros **private** no son visibles en las clases derivadas (PERO SÍ SE HEREDAN).
- Los miembros **protected** SOLO son accesibles desde todas las clases derivadas o indirectamente de la clase base.
- Una clase derivada no puede ser más accesible que su clase base.

- Si no se realiza una llamada explícita a un constructor de clase base [:base()], el compilador de C# proporciona automáticamente una llamada al constructor sin parámetros o predeterminado de la clase base.

CLASE 10: ABSTRACT Y VIRTUAL

Definición Formal: La abstracción es el efecto de separar conceptualmente algo de algo. El modificador abstract se utiliza para indicar que una clase está incompleta y que sólo se va a utilizar como una clase base.

* **No se puede crear** una instancia de una clase abstracta **directamente**.
* Da **error** en tiempo de compilación al utilizar el **operador new** en una clase abstracta.
* 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.

* Cuando una clase **no abstracta** se deriva de una **clase abstracta**, la **clase no abstracta** debe incluir **implementaciones reales** de todos los miembros abstractos heredados.

* Las clases abstractas se sitúan en la cima de la jerarquía de clases y establecen la estructura y significado del código.

* Facilitan la creación de **marcos de trabajo**.

```
abstract class A
{
}
class B : A
{
    public void G() { }
}

// ...

A a = new A(); // ERROR!
B b = new B(); // Correcta!
```

Miembros abstractos

Cuando una declaración de método de instancia incluye un modificador **abstract**, se dice que el método es un método abstracto. Las declaraciones de métodos abstractos sólo se permiten en clases abstractas. Por definición formal, un método abstracto es también implícitamente un método virtual.

Dada la definición formal, podríamos decir:

Una declaración de **método abstracto** introduce un nuevo **método virtual**, pero **no proporciona una implementación** del método. Es necesario que las **clases derivadas no abstractas proporcionen su propia implementación mediante el reemplazo del método**. Son métodos y propiedades que se declaran sin implementación.

Miembros Virtuales

El modificador *virtual* sirve para métodos, propiedades, indexadores o evento declarado y permitir que este sea anulado en una clase derivada. Cuando una declaración de método de instancia incluye un modificador *virtual*, se dice que el este es un método virtual. Si no existe un modificador virtual, se dice que el método es un método no virtual.

El modificador **override** es necesario para ampliar o modificar la implementación abstracta o virtual de un método, propiedad, indixador o evento heredado. Un método **override** proporciona una nueva implementación de un miembro que se hereda de una clase base. El método base reemplazado debe tener la misma firma que el método override.

No se puede reemplazar un método **estático** o no **virtual**. El método base reemplazado debe ser **virtual**, **abstract** u **override**. El método **override** y el método virtual deben tener el mismo modificador de nivel de acceso. **No se pueden usar los modificadores new, static o virtual para modificar un método override.**

CLASE 11: POLIMORFISMO

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 las relaciones de herencia, esto implica la definición de métodos en una clase base y sobrescribirlos con nuevas implementaciones en clases derivadas.

La **definición del método** reside en la clase base -> Se utiliza **virtual**.
La **implementación del método** reside en la clase derivada -> Se utiliza **override**.
La invocación es resuelta en tiempo de ejecución.

Virtual y Override

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**.

```
[modificadores] virtual retorno Metodo( [Args])
{
   // Implementación del método en clase Base
}

[modificadores] override retorno Metodo( [Args])
{
   // Implementación del método en clase Derivada
}
```

OTRO EJEMPLO:

```
class ClaseBase
{
    public virtual void NombreMétodo()
    {
        // Implementación del método
    }
}
class ClaseDerivada : ClaseBase
{
     public override void NombreMétodo()
     {
          base.NombreMétodo(); // Llamada al método virtual
          // Implementación específica
     }
}
```