Apuntes Completos del Seminario .NET

Clase 1: Introducción a .NET y C#

Tema: Herramientas Necesarias

- **.NET 8.0**: Se trabajará con la versión .NET 8.0, que tiene soporte a largo plazo (LTS) hasta el 10/11/2026. No se usará .NET 9.0 por ser de soporte corto y no traer mejoras relevantes para el curso.
- **Visual Studio Code**: Se utilizará como editor de código.

Tema: Generalidades de .NET

- **Historia y Evolución**:
    - **.NET Framework**: Lanzado en 2002 para Windows, no es open source. Su última versión es la 4.8.
    - **.NET Core**: Versión multiplataforma y open-source iniciada en 2016. Su última versión fue la 3.1.
    - **.NET (5.0 y superior)**: A partir de noviembre de 2020, .NET Core evolucionó a .NET 5.0, unificando la plataforma. Las versiones posteriores son .NET 6.0, .NET 7.0 y .NET 8.0.
- **Plataforma .NET**:
    - Es una plataforma de desarrollo gratuita y de código abierto.
    - Soporta múltiples lenguajes como C#, Visual Basic y F#.
    - Permite construir aplicaciones Web, Mobile, Desktop, Microservicios, Cloud, Machine Learning, Juegos e Internet of Things.
- **Componentes Clave**:
    - **Common Language Runtime (CLR)**: Es el motor de ejecución de .NET. Provee un entorno virtual donde se ejecutan las aplicaciones.
    - **Microsoft Intermediate Language (MSIL/CIL)**: Al compilar código C# (u otro lenguaje .NET), se genera un código intermedio llamado MSIL.
    - **Just In Time (JIT) Compiler**: En tiempo de ejecución, el compilador JIT traduce el código MSIL al código nativo del procesador.
    - **Base Class Library (BCL)**: Es la biblioteca de clases base de .NET, organizada en namespaces que agrupan funcionalidades. Ejemplos:

        System, System.Collections, System.IO.

Tema: Introducción a C#

- **Primera Aplicación (Hola Mundo)**:
    - Se crea un nuevo proyecto de consola con el comando

      dotnet new console -o <nombre>.
    - El código fuente principal se encuentra en el archivo

      Program.cs.
    - Se compila y ejecuta con

      dotnet run.
- **Instrucciones de Nivel Superior**: Desde .NET 6, la plantilla de consola utiliza esta característica que permite escribir el cuerpo del método Main directamente en Program.cs, simplificando el código.
- **Directiva using**: Permite usar tipos de un namespace sin escribir el nombre completo. Con <ImplicitUsings>enable</ImplicitUsings> en el archivo .csproj, el compilador agrega un conjunto de directivas using comunes automáticamente.

Tema: Tipos de Datos y Operadores

- **Tipos de Datos Integrados**:
    - C# ofrece tipos para enteros (int, long, byte), punto flotante (float, double, decimal), booleanos (bool), caracteres (char) y texto (string).
    - Estos son alias de tipos de .NET (ej. int es System.Int32).
- **Literales**:
    - Los literales numéricos con punto decimal son double por defecto. Para float se usa el sufijo f y para decimal el sufijo m.
- **Operadores**:
    - **Aritméticos**: +, -, *, /, %. La división de enteros trunca el resultado.
    - **Relacionales**: ==, !=, <, >, <=, >=.
    - **Lógicos**: !, &, |, ^ y sus versiones de cortocircuito && y ||.
    - **Asignación**: =, +=, -=, ++, --.

Tema: Estructuras de Control

- **Bloque**: Conjunto de sentencias entre llaves {}. Una variable declarada dentro de un bloque tiene ese alcance.
- **Condicionales**:
    - if-else: Ejecuta bloques de código basados en una condición booleana.
    - switch: Compara una expresión con diferentes case.
    - **Operador Ternario** ?:: Una forma compacta de if-else para asignar un valor.

        condición ? valor_si_verdadero : valor_si_falso.
- **Bucles**:
    - while: Ejecuta un bloque mientras una condición sea verdadera.
    - do-while: Similar a while, pero garantiza la ejecución del bloque al menos una vez.
    - for: Permite ejecutar un bloque un número determinado de veces, con inicialización, condición e iterador.
    - **Sentencias de interrupción**: break para salir del bucle y continue para saltar a la siguiente iteración.

Clase 2: Sistema de Tipos

Tema: Sistema de Tipos Común (CTS)

- **Definición**: El CTS define un conjunto común de tipos para que los lenguajes .NET puedan interoperar.
- **Categorías de Tipos**:
    - **Tipos de Valor (Value Types)**: Almacenan su valor directamente en la pila de ejecución (stack). Incluyen struct y enum.
    - **Tipos de Referencia (Reference Types)**: Almacenan en la pila una dirección a la memoria heap, donde se encuentra el valor real. Incluyen class, delegate e interface.

Tema: Conversión de Tipos

- **Conversiones Implícitas**: Se realizan automáticamente cuando no hay riesgo de pérdida de datos (ej. de int a double).
- **Conversiones Explícitas (Casting)**: Requieren un operador de conversión (tipo) y se usan cuando puede haber pérdida de información (ej. de double a int).
- **Clases Auxiliares**:
    - La clase System.Convert permite conversiones entre tipos no compatibles (ej. bool a int).
    - Los métodos .Parse() (ej. int.Parse("123")) convierten un string a un tipo numérico.
    - El método .ToString() convierte un objeto a su representación en string.

Tema: Sistema Unificado de Tipos

- **System.Object**: Todos los tipos en .NET derivan directa o indirectamente de la clase System.Object, que es un tipo de referencia.
- **Boxing y Unboxing**:
    - **Boxing**: Es el proceso de convertir un tipo de valor a un tipo de referencia (como object). Esto implica crear un objeto "caja" en el heap para almacenar el valor.
    - **Unboxing**: Es el proceso inverso, de extraer el valor de la "caja" en el heap y pasarlo a una variable de tipo de valor.
    - Estas operaciones tienen un costo de rendimiento.
- **Consecuencias**: Como todos los tipos son object, pueden invocar métodos definidos en object, como GetType() y ToString().

Tema: Tipos de Datos Adicionales

- **Arreglos (Arrays)**:
    - Son tipos de referencia.
    - Pueden ser de una dimensión (int[]), multidimensionales (int[,]) o irregulares (arreglos de arreglos, int[][]).
    - Se pueden inicializar con la sintaxis de **expresión de colección** ([1, 2, 3]) a partir de C# 12.

- **Inferencia de Tipos (var)**:
    - La palabra clave var permite al compilador inferir el tipo de una variable local a partir de su inicialización.
    - La variable sigue siendo fuertemente tipada; su tipo no puede cambiar después de la declaración.
- **Tipos Anónimos**:
    - Permiten crear objetos simples con propiedades de solo lectura sin definir explícitamente una clase. Se crean con

      new { Propiedad = valor, ... } y se asignan a una variable var.
- **Tipo dynamic**:
    - Permite omitir la comprobación de tipos en tiempo de compilación. Las operaciones se verifican en tiempo de ejecución, lo que puede causar excepciones si una operación no es válida.
- **Tipos Nullable**:
    - Para que un tipo de valor pueda aceptar null, se declara con un ? (ej. int?).
    - **Operador ?? (Null-Coalescing)**: Provee un valor por defecto si una variable es null.

variable ?? valor_defecto es equivalente a variable != null ? variable : valor_defecto.

Tema: Formato de Strings

- **Cadenas de Formato Compuesto**: Utilizan marcadores de posición indizados (ej. "{0}") que se reemplazan con los valores de una lista de argumentos.

  string.Format("Es un {0} año {1}", "Ford", 2000).
- **Cadenas Interpoladas**: A partir de C# 6.0, usan el prefijo $ y permiten insertar expresiones directamente entre llaves.

  $"Es un {marca} año {modelo}"

  Son más legibles y es la forma recomendada.

Clase 3: Más sobre Tipos y Manejo de Excepciones

Tema: Arreglos Multidimensionales e Irregulares

- **Arreglos de 2 Dimensiones (Matrices)**: Se declaran con una coma dentro de los corchetes (ej. int[,]). Todos los elementos de la fila tienen la misma longitud.
- **Arreglo de Arreglos (Jagged Arrays)**: Son arreglos cuyos elementos son, a su vez, arreglos. Cada "fila" puede tener una longitud diferente. Se declaran con corchetes sucesivos (ej. int[][]).

Tema: Manejo de Excepciones

- **Excepciones**: Son errores que ocurren en tiempo de ejecución , como

  NullReferenceException, IndexOutOfRangeException o DivideByZeroException.

- **Bloque try-catch-finally**:
  - **try**: Encierra el código que podría lanzar una excepción.
  - **catch**: Contiene el código que maneja la excepción. Se pueden tener múltiples bloques catch para distintos tipos de excepciones. El catch más general (sin tipo o con Exception) se coloca al final.
  - **finally**: Este bloque se ejecuta siempre, haya o no una excepción. Se usa para liberar recursos.
- **Lanzar Excepciones (throw)**:
  - Se pueden lanzar excepciones personalizadas o existentes usando la palabra clave throw.
  - throw new Exception("mensaje"); crea y lanza una nueva excepción.
  - Dentro de un bloque catch, throw; relanza la excepción original capturada, conservando la información de la pila.

Clase 4: Programación Orientada a Objetos (POO)

Tema: Conceptos Fundamentales

- **POO**: Es un paradigma de programación que modela la realidad a través de objetos que colaboran entre sí. Sus elementos fundamentales son el **objeto** y el **mensaje**.
- **Clase**: Es una plantilla que describe los **atributos** (campos) y el **comportamiento** (métodos) de un conjunto de objetos.
- **Objeto**: Es una instancia de una clase. Se crea con la palabra clave new.

Tema: Miembros de una Clase

- **Miembros de Instancia**: Pertenecen a un objeto específico. Cada instancia tiene su propia copia de estos miembros.
    - **Campos**: Son variables que almacenan el estado de un objeto. Por defecto, son private.
    - **Métodos**: Definen el comportamiento del objeto. Pueden acceder a todos los campos del objeto, incluso a los privados.
- **Encapsulamiento**: Es la práctica de ocultar la representación interna de un objeto, exponiendo solo una interfaz pública. Es una buena práctica mantener los campos como private y controlarlos a través de métodos o propiedades.
- **Referencia this**: Dentro de un método de instancia, this se refiere al propio objeto que está ejecutando el código.

Tema: Constructores

- **Definición**: Es un método especial que se ejecuta al crear una nueva instancia de una clase (new). Se usa para inicializar el estado del objeto.
- **Sintaxis**: Tiene el mismo nombre que la clase y no tiene tipo de retorno.
- **Constructor por Defecto**: Si no se define ningún constructor, el compilador proporciona uno público, sin parámetros y con cuerpo vacío. Si se define cualquier constructor explícitamente, el constructor por defecto ya no se genera automáticamente.
- **Sobrecarga de Constructores**: Una clase puede tener múltiples constructores, siempre que sus firmas (cantidad, tipo y orden de los parámetros) sean diferentes.
- **Encadenamiento de Constructores (: this(...))**: Un constructor puede invocar a otro de la misma clase para reutilizar la lógica de inicialización.

- **Constructores Primarios (C# 12)**: Permiten declarar parámetros directamente en el encabezado de la clase, simplificando la inicialización.

Clase 5: Miembros Estáticos, Propiedades e Indizadores

Tema: Miembros Estáticos

- **Definición**: Son miembros que pertenecen a la clase en sí, no a una instancia particular. Se declaran con el modificador static.
- **Acceso**: Se accede a ellos usando el nombre de la clase, no una instancia (NombreClase.MiembroEstatico).
- **Campos Estáticos**: Es una única variable compartida por todas las instancias de la clase. Sirve para mantener datos globales para esa clase (ej. un contador de instancias).
- **Métodos Estáticos**: Solo pueden acceder a otros miembros estáticos de la misma clase. No pueden acceder a miembros de instancia (como _monto) porque no están asociados a ningún objeto.
- **Constructores Estáticos**:
    - Se ejecutan una sola vez, cuando el runtime de .NET carga la clase.
    - No pueden tener parámetros ni modificadores de acceso.
    - Se usan para inicializar campos estáticos.

Tema: Clases Estáticas

- Una clase declarada como static solo puede contener miembros estáticos.
- No se pueden crear instancias de una clase estática.
- Son útiles para agrupar funciones de utilidad (utility classes), como System.Math o System.Console.

Tema: Constantes y Campos de Solo Lectura

- **const (Campos Constantes)**:
    - Su valor es inmutable y se conoce en tiempo de compilación.
    - Deben inicializarse en su declaración.
    - Son implícitamente static.
- **readonly (Campos de Solo Lectura)**:

    - Su valor se asigna en tiempo de ejecución, ya sea en la declaración o dentro de un constructor.
    - A diferencia de const, pueden ser de cualquier tipo.
    - Pueden ser de instancia o estáticos.

Tema: Encapsulamiento y Propiedades

- **Rol del Encapsulamiento**: Es un pilar de la POO que consiste en ocultar los detalles de implementación, protegiendo el estado del objeto (campos) declarándolos como private.
- **Getters y Setters**: Un patrón tradicional para acceder a campos privados, pero en .NET se prefieren las propiedades.
- **Propiedades (get/set)**:
    - Combinan la simplicidad de acceso de un campo con la lógica de un método.
    - El bloque get se ejecuta al leer la propiedad.
    - El bloque set se ejecuta al asignar un valor a la propiedad, usando el parámetro implícito value.
    - Permiten añadir lógica como validación.
- **Propiedades de Solo Lectura/Escritura**:
    - Una propiedad con solo get es de solo lectura.
    - Una propiedad con solo set es de solo escritura (uso no recomendado).
- **Propiedades Autoimplementadas**:
    - Sintaxis simplificada public string Nombre { get; set; } cuando no se necesita lógica personalizada.
    - El compilador crea un campo de respaldo (backing field) oculto automáticamente.
    - Pueden ser inicializadas en su declaración.
- **Propiedades Estáticas**: También pueden ser declaradas como static para trabajar con campos estáticos.

Tema: Indizadores (Indexers)

- **Definición**: Permiten que los objetos de una clase sean indexados como si fueran un arreglo (usando []).

- **Sintaxis**: Se declaran usando la palabra clave this como nombre del miembro.

  public T this[int i] { get { ... } set { ... } }.

- **Flexibilidad**: A diferencia de los arreglos, el índice no está limitado a enteros; puede ser de cualquier tipo (ej. string), y se pueden sobrecargar múltiples indizadores.

Clase 6: Herencia y Polimorfismo

Tema: Herencia

- **Definición**: Mecanismo que permite a una clase (clase derivada) reutilizar, extender y modificar el comportamiento de otra (clase base).
- **Relación "Es Un" (Is-A)**: La herencia establece una relación "es un". Si Auto hereda de Automotor, entonces un Auto **es un** Automotor.
- **Sintaxis**: class ClaseDerivada : ClaseBase { ... }.
- **Herencia de Miembros**: Una clase derivada hereda todos los miembros de su clase base, excepto los constructores y finalizadores. La visibilidad depende de los modificadores de acceso (public, protected, private).
- **System.Object**: Todas las clases en .NET derivan implícita o explícitamente de System.Object.

Tema: Polimorfismo y Métodos Virtuales

- **Polimorfismo**: Es la capacidad de un objeto de ser tratado como una instancia de su tipo base. Permite que una variable de tipo Automotor pueda contener una instancia de Auto o de Colectivo.
- **virtual y override**:
  - Para que un método de una clase base pueda ser reimplementado en una clase derivada, debe marcarse como virtual.
  - La clase derivada usa la palabra clave override para proporcionar su propia implementación de ese método.
- **Enlace Dinámico**: Cuando se invoca un método virtual sobre una referencia de tipo base, el CLR determina en tiempo de ejecución qué implementación específica (override) ejecutar, basándose en el tipo real del objeto. Esto es la base del comportamiento polimórfico.

- **Acceso a la Clase Base (base)**: Desde una clase derivada, se puede usar la palabra clave base para acceder a miembros de la clase base, como invocar su constructor (: base(parametros)) o un método sobrescrito (base.Imprimir()).

Tema: Clases y Miembros Abstractos

- **Clases Abstractas (abstract class)**:
    - Son clases base que no pueden ser instanciadas.
    - Su propósito es definir un comportamiento común que las clases derivadas deben implementar.
- **Miembros Abstractos (abstract void Metodo();)**:
    - Son miembros declarados en una clase abstracta que no tienen implementación.
    - Obligan a las clases derivadas no abstractas a proporcionar una implementación (override).

Tema: Extensión de Métodos (Extension Methods)

- **Definición**: Permiten "agregar" métodos a tipos existentes (incluyendo clases selladas o estructuras) sin modificar su código fuente original.
- **Implementación**:
    - Se definen como métodos static dentro de una clase static.
    - El primer parámetro del método especifica el tipo que se está extendiendo y se precede con la palabra clave this.

Clase 7: Interfaces y Arquitectura de Software

Tema: Interfaces

- **Definición**: Una interfaz es un tipo de referencia que define un **contrato**. Especifica un conjunto de miembros (métodos, propiedades, eventos, indizadores) que una clase o struct que la implementa debe proporcionar.
- **Propósito**: Permiten que clases de diferentes jerarquías compartan una interfaz polimórfica común, desacoplando el código.
- **Implementación**: Una clase puede implementar múltiples interfaces.

class MiClase : ClaseBase, IInterfaz1, IInterfaz2 { ... }.

- **Implementación Explícita**: Cuando una clase implementa dos interfaces que tienen un miembro con el mismo nombre y firma, puede implementar cada uno explícitamente para evitar ambigüedad: void IInterfaz1.Metodo() { ... }. Estos miembros solo son accesibles a través de una referencia del tipo de la interfaz.

Tema: Principio de Inversión de Dependencias (DIP)

- **Enunciado**: Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben depender de **abstracciones** (interfaces).
- **Objetivo**: Lograr un **acoplamiento débil** entre los componentes del software, lo que facilita el mantenimiento, la prueba y la extensibilidad.
- **Patrón de Inyección de Dependencias (DI)**: Es la técnica principal para aplicar el DIP. Consiste en "inyectar" las dependencias (objetos que una clase necesita) desde el exterior, en lugar de que la clase las cree internamente. La forma más común es la **inyección por constructor**.

Tema: Arquitectura Limpia (Clean Architecture)

- **Concepto**: Es un diseño de software que organiza el proyecto en capas concéntricas (anillos) para lograr la separación de responsabilidades y un bajo acoplamiento.
- **Regla de Dependencia**: Las capas externas dependen de las internas, pero las internas no saben nada de las externas.
- **Capas Propuestas en el Curso**:
    1. **Aplicación (Núcleo)**: Contiene la lógica de negocio, las entidades y las interfaces de los repositorios. No depende de ninguna otra capa.
    2. **Repositorios (Infraestructura)**: Implementa las interfaces de la capa de aplicación para la persistencia de datos (ej. acceso a base de datos, archivos de texto). Depende de la capa de Aplicación.
    3. **UI (Interfaz de Usuario)**: Es la capa de presentación (ej. aplicación de consola, Blazor). Depende de las otras dos capas.

Clase 8: Interfaces Avanzadas y Delegados

Tema: Interfaces para Comparación

- **System.IComparable**: Permite que los objetos de una clase definan un orden natural para sí mismos. Requiere implementar el método int CompareTo(object obj), que devuelve un valor negativo, cero o positivo si la instancia actual precede, es igual o sigue al objeto obj. Array.Sort() la utiliza para ordenar.
- **System.IComparer**: Permite definir criterios de ordenamiento externos y personalizados. Requiere implementar el método int Compare(object x, object y). Se pasa una instancia de la clase comparadora como argumento a Array.Sort().

Tema: Interfaces para Enumeración

- **IEnumerable**: Expone un enumerador, que permite la iteración sobre una colección. Una clase que implementa esta interfaz puede ser usada en un bucle foreach. Requiere el método GetEnumerator().
- **IEnumerator**: Proporciona la lógica para iterar a través de una colección. Define la propiedad Current y los métodos MoveNext() y Reset().
- **Iteradores (yield)**: Simplifican enormemente la creación de enumeradores.
    - yield return valor; devuelve el siguiente elemento de la secuencia.
    - yield break; finaliza la iteración.

Tema: Delegados

- **Definición**: Un delegado es un tipo de referencia que puede encapsular un método (o múltiples métodos). Permite tratar a los métodos como si fueran variables: asignarlos, pasarlos como parámetros, etc.
- **Sintaxis**: delegate int MiDelegado(string param);. La firma del delegado debe coincidir con la de los métodos que encapsulará.
- **Métodos Anónimos**: Permiten crear un bloque de código "en línea" para asignarlo a un delegado, sin necesidad de definir un método con nombre separado. Se usa la palabra clave delegate.
- **Expresiones Lambda (=>)**: Son una sintaxis más concisa y preferida para crear métodos anónimos. Ej: (n) => n * n;.

Clase 9: Genéricos

Tema: ¿Por qué usar Genéricos?

- Los genéricos permiten escribir clases, interfaces y métodos que trabajan con cualquier tipo de dato, manteniendo la **seguridad de tipos** en tiempo de compilación y evitando conversiones (casting) y boxing/unboxing ineficientes.

Tema: Métodos Genéricos

- **Definición**: Un método genérico se declara con parámetros de tipo entre corchetes angulares, después del nombre del método: void Metodo<T>(T parametro).
- **Inferencia de Tipos**: En muchos casos, el compilador puede inferir los argumentos de tipo a partir de los parámetros del método, por lo que no es necesario especificarlos explícitamente en la llamada (Metodo(123) en lugar de Metodo<int>(123)).
- **Restricciones (where)**: Se puede restringir los tipos que se pueden usar como argumentos de tipo mediante la cláusula where. Ejemplos:
    - where T : IComparable: T debe implementar la interfaz IComparable.
    - where T : class: T debe ser un tipo de referencia.
    - where T : struct: T debe ser un tipo de valor.
    - where T : new(): T debe tener un constructor público sin parámetros.

Tema: Tipos Genéricos

- **Clases, Interfaces y Delegados Genéricos**: Se declaran con parámetros de tipo después del nombre del tipo.
    - **Clases Genéricas**: class MiClase<T> { ... }. Un ejemplo común es List<T>.
    - **Interfaces Genéricas**: interface IInterfaz<T> { ... }. Ejemplos de la BCL son IEnumerable<T>, IComparer<T>.
    - **Delegados Genéricos**: delegate TResult MiDelegado<T, TResult>(T param);. La BCL provee los delegados genéricos Action<> (para métodos void) y Func<> (para métodos que devuelven un valor).
- **Varianza (Covarianza y Contravarianza)**:
    - **Covarianza (out)**: Permite usar un tipo más derivado. Aplica cuando el tipo genérico T solo se usa en posiciones de **salida** (ej. valor de retorno).

IEnumerable<out T> es covariante.

- **Contravarianza (in)**: Permite usar un tipo menos derivado. Aplica cuando T solo se usa en posiciones de **entrada** (ej. parámetros de método).

  IComparer<in T> y Action<in T> son contravariantes.

- **Invarianza**: Es el comportamiento por defecto. No permite sustitución.

  List<T> es invariante.

Clase 10: LINQ y Persistencia con EF Core

Tema: LINQ (Language-Integrated Query)

- **Concepto**: LINQ es un conjunto de métodos de extensión, principalmente para la interfaz IEnumerable<T>, que permiten realizar consultas y transformaciones de datos de forma declarativa y fluida.
- **Operadores Principales**:
  - Select: Proyecta cada elemento de una secuencia a una nueva forma.
  - Where: Filtra una secuencia basándose en un predicado (una función que devuelve bool).
  - OrderBy / OrderByDescending: Ordena los elementos de una secuencia.
  - GroupBy: Agrupa los elementos de una secuencia según una clave.
  - Join: Combina dos secuencias basándose en claves coincidentes (similar al INNER JOIN de SQL).
  - **Operadores de Agregación**: Sum, Average, Count, Max, Min.
  - **Operadores de Conjunto**: Concat, Union, Intersect.
- **Ejecución Diferida (Deferred Execution)**: Las consultas LINQ no se ejecutan hasta que el resultado es realmente necesario (ej. cuando se itera con un foreach o se invoca un método como ToList() o Sum()).

Tema: Persistencia de Datos con SQLite y Entity Framework Core (EF Core)

- **ORM (Object-Relational Mapper)**: EF Core es un ORM que permite a los desarrolladores trabajar con una base de datos usando objetos .NET, eliminando la necesidad de escribir la mayor parte del código de acceso a datos (SQL) manualmente.
- **SQLite**: Es un motor de base de datos SQL autónomo y sin servidor, ideal para aplicaciones locales o de pequeña escala.

- **Modelo de EF Core**:
    - **Clases de Entidad**: Son clases POCO (Plain Old C# Object) que representan las tablas de la base de datos. Sus propiedades se mapean a las columnas de la tabla.
    - **Contexto de Base de Datos (DbContext)**: Representa una sesión con la base de datos. Se hereda de DbContext y se definen propiedades DbSet<T> para cada entidad, que representan las tablas.
    - **Configuración**: El método OnConfiguring del DbContext se usa para especificar el proveedor de base de datos y la cadena de conexión.
- **Estrategia "Code First"**: Permite crear la base de datos y su esquema a partir del código C# (clases de entidad y DbContext). El método Database.EnsureCreated() crea la base de datos si no existe.
- **Operaciones CRUD**:
    - **Crear (Create)**: context.Add(nuevoObjeto);
    - **Leer (Read)**: Usando consultas LINQ sobre los DbSet (ej. context.Alumnos.Where(...)).
    - **Actualizar (Update)**: Se modifica un objeto traído del contexto y luego se guarda.
    - **Borrar (Delete)**: context.Remove(objetoAEliminar);
    - **Guardar Cambios**: context.SaveChanges() aplica todas las operaciones pendientes a la base de datos.

Clase 11: Inyección de Dependencias (DI)

Tema: Contenedor de Inyección de Dependencias (DI Container)

- **Propósito**: Un DI Container es una herramienta que automatiza la creación y gestión de las dependencias de una aplicación. En lugar de crear objetos manualmente (new), se le pide al contenedor que proporcione una instancia.
- **Contenedor de .NET**:
    - Se basa en el paquete Microsoft.Extensions.DependencyInjection (incluido en Microsoft.Extensions.Hosting).
    - **IServiceCollection**: Es donde se registran los servicios (dependencias).
    - **IServiceProvider**: Se construye a partir de la ServiceCollection y se usa para obtener instancias de los servicios registrados (proveedor.GetService<IMiServicio>()).
- **Registro de Servicios**:
    - AddTransient<IInterfaz, Clase>(): Se crea una nueva instancia cada vez que se solicita el servicio.
    - AddScoped<IInterfaz, Clase>(): Se crea una única instancia por "ámbito". En Blazor Server, un ámbito corresponde a la conexión de un usuario.
    - AddSingleton<IInterfaz, Clase>(): Se crea una única instancia que es compartida por toda la aplicación durante su ciclo de vida.

Tema: Integración con Blazor

- **Blazor y DI**: Las aplicaciones Blazor vienen con un contenedor de DI integrado de fábrica.
- **Registro**: Los servicios se registran en Program.cs usando builder.Services (que es un IServiceCollection).
- **Inyección en Componentes**: Se usa la directiva @inject en un archivo .razor para inyectar un servicio en una propiedad del componente.

```
@inject IServicioX MiServicio.
```

Clase 12: Arquitectura Limpia en Blazor

Tema: Repaso de Arquitectura Limpia

- Se revisan los conceptos de separación en capas (UI, Repositorios, Aplicación) y la regla de dependencias (capas externas dependen de las internas).

Tema: Implementación Práctica en Blazor

- **Estructura de la Solución**: Se crea una solución con tres proyectos:
    1. AL.Aplicacion (Biblioteca de Clases)
    2. AL.Repositorios (Biblioteca de Clases)
    3. AL.UI (Aplicación Blazor Server).
- **Referencias entre Proyectos**:
    - AL.UI referencia a AL.Aplicacion y AL.Repositorios.
    - AL.Repositorios referencia a AL.Aplicacion.
- **Desarrollo de la UI con Blazor**:
    - **Componentes**: Se crean componentes .razor para las diferentes vistas de la aplicación (ej. ListadoClientes.razor, EditarCliente.razor).
    - **Enrutamiento**: Se usa la directiva @page "/ruta" para hacer un componente navegable. Se pueden incluir parámetros en la ruta, como

      @page "/cliente/{Id:int?}".
    - **Inyección de Servicios**: Se utiliza @inject para obtener instancias de los casos de uso y otros servicios registrados en el DI container.
    - **Manejo de Eventos**: Se usa la directiva @onclick para asociar un método del componente al evento click de un botón.
    - **EventCallback**: Se usa para que un componente hijo pueda notificar un evento a su componente padre (ej. un diálogo de confirmación).
    - **Aislamiento de CSS**: Se pueden crear estilos específicos para un componente creando un archivo NombreComponente.razor.css en la misma carpeta.

Clase 13: Programación Asincrónica

Tema: Patrón Asincrónico Basado en Tareas (TAP)

- **Concepto**: Es el enfoque recomendado para la programación asincrónica en .NET. Se basa en los tipos

Task y Task<TResult>.

- **Task**: Representa una operación asincrónica que no devuelve un valor.
- **Task<TResult>**: Representa una operación asincrónica que devuelve un valor de tipo TResult.
- **async y await**:
    - **async**: Es un modificador que se aplica a la declaración de un método para indicar que es asincrónico. Permite el uso del operador await dentro de él.
    - **await**: Es un operador que suspende la ejecución del método async hasta que la tarea (Task) que se está esperando se complete. Durante la espera, el hilo de ejecución no se bloquea, sino que se libera para realizar otro trabajo.

Tema: Creación y Ejecución de Tareas

- **new Task(...) y Start()**: Se puede crear una instancia de Task pasando un Action (un método void) y luego iniciarla con .Start().
- **Task.Run()**: Es una forma más simple de crear e iniciar una tarea en una sola operación.
- **Esperando Tareas**:
    - t.Wait(): Bloquea el hilo actual y espera de forma **sincrónica** a que la tarea t finalice. Debe evitarse en métodos asincrónicos ya que bloquea el hilo.
    - await t: Espera de forma **asincrónica** a que la tarea t finalice, liberando el hilo actual.
    - Task.WaitAll(...): Espera a que todas las tareas en un arreglo finalicen.

Tema: Métodos Asincrónicos

- **Convención de Nomenclatura**: Por convención, los métodos asincrónicos deben terminar con el sufijo Async (ej. PrintAsync).
- **Tipos de Retorno**:
    - async Task: Para una operación asincrónica que no devuelve un valor.

    - async Task<T>: Para una operación asincrónica que devuelve un valor de tipo T. El valor se devuelve con return valor; (no return Task.FromResult(valor)).
    - async void: **Debe evitarse**, excepto para controladores de eventos. No se puede esperar una tarea void ni capturar sus excepciones.
- **Obtención de Resultados**: El resultado de una Task<T> se puede obtener con la propiedad .Result (que bloquea) o de forma asincrónica con await (var resultado = await miTarea;).