DESARROLLO DE SOFTWARE

IDS340-02

Resumen Nivel 1 / Lección 3 y 4

Daniel Germán ID: 1128127

28 de Noviembre, 2024

La programación orientada a objetos (OOP) es un paradigma de programación que organiza el código en "objetos", que son instancias de clases, cada uno con sus propios atributos y métodos. Este enfoque permite la reutilización de código, la modularidad y la abstracción, facilitando el mantenimiento y la escalabilidad de los sistemas.

Los modificadores son palabras clave que agregan o quitan funciones a los tipos de referencia, como clases, o a los miembros definidos en ellas, como propiedades y métodos. Algunas de las más útiles son public y private, que definen cuándo se pueden acceder a las propiedades definidas dentro de una clase, otros como readonly o static que modifican a las propiedades en sí. En C#, las clases se crean utilizando la palabra clave class, seguida del nombre de la clase y un bloque de código que define sus atributos y métodos. Los atributos son variables que almacenan el estado de la clase, mientras que los métodos son funciones que definen el comportamiento de los objetos creados a partir de la clase.

Los métodos se implementan dentro de una clase, y se definen con un tipo de retorno, un nombre y una lista de parámetros opcionales. El tipo de retorno puede ser cualquier tipo de dato, como void si no se espera ningún valor de retorno, y los parámetros son variables que los métodos pueden aceptar como entrada. Los métodos pueden tener modificadores de acceso como public, private y protected, que determinan su visibilidad. Por ejemplo, un método private sólo puede ser accedido dentro de la misma clase.

Los objetos se crean a partir de una clase y se utilizan para acceder a los atributos y métodos definidos dentro de esa clase. Un objeto es una instancia de una clase y se puede considerar como una "copia" de la clase que contiene datos y comportamientos.

Para crear y usar un objeto:

1. Definir una clase: Definir una clase que actúa como plantilla para crear objetos.
2. Crear un objeto: Usar el operador **new** para crear una instancia de la clase.
3. Acceder a los miembros del objeto: Usar el operador **.** para acceder a los atributos y métodos del objeto.

Un campo es una variable que se define dentro de una clase para almacenar datos. Los campos representan el estado o las características del objeto. Las propiedades son una forma de exponer los campos de una clase a través de métodos de acceso (getters y setters) de una

manera más controlada. En lugar de acceder a los campos directamente, los usuarios de la clase interactúan con las propiedades, lo que permite agregar lógica adicional como validaciones si es necesario. Un constructor es un método especial dentro de una clase que se llama automáticamente cuando se crea un objeto de esa clase. Su propósito principal es inicializar el objeto, es decir, establecer valores iniciales para los campos.

Una clase estática es una clase que no puede ser instanciada (no se pueden crear objetos de ella) y cuya funcionalidad está asociada a métodos y propiedades que son accesibles sin necesidad de crear una instancia de la clase. Las clases estáticas se utilizan principalmente para agrupar métodos y propiedades que no dependen de una instancia específica de la clase.

- La clase solo puede contener miembros estáticos, por lo que no se pueden crear objetos dentro de una.
- Todos los campos, métodos, propiedades y eventos dentro de una clase estática deben ser estáticos. No se pueden usar miembros de instancia dentro de una clase estática.
- Los miembros estáticos de una clase estática se acceden directamente a través del nombre de la clase, sin necesidad de crear una instancia de la clase.

Clase Base (Superclase): Es la clase de la cual otras clases pueden heredar. Contiene atributos y métodos que se pueden compartir o modificar por las clases derivadas.

Clase Derivada (Subclase): Es la clase que se hereda de la clase base. Puede acceder a los miembros públicos y protegidos de la clase base y puede agregar o modificar sus propios miembros.

El polimorfismo permite que diferentes tipos de objetos puedan ser tratados de manera uniforme a través de una interfaz común, pero que, al mismo tiempo, puedan comportarse de manera diferente según su tipo específico. El mismo mensaje o la misma acción puede generar diferentes comportamientos dependiendo del objeto que la reciba. Esto permite que una sola función, método o clase pueda actuar de diferentes maneras según el contexto.

```
using System;


public abstract class GatoDaniel

{


    public string Color { get; set; }

    public string TipoPelo { get; set; }

    public string ColorOjos { get; set; }


    public static int ContadorGatos = 0;


    public Gato(string color, string tipoPelo, string colorOjos)

    {

        Color = color;

        TipoPelo = tipoPelo;

        ColorOjos = colorOjos;

        ContadorGatos++;

        Console.WriteLine($"Contador de gatos: {ContadorGatos}");

    }


    public abstract void Comer();

    public abstract void Maullar();

    public abstract void Dormir();

}
```

```
public class Siames : Gato

{

    public Siames(string color, string tipoPelo, string colorOjos) :
base(color, tipoPelo, colorOjos) { }



    public override void Comer()

    {

        Console.WriteLine("El Siames está comiendo.");

    }



    public override void Maullar()

    {

        Console.WriteLine("El Siames maúlla suavemente.");

    }



    public override void Dormir()

    {

        Console.WriteLine("El Siames duerme en su cama favorita.");

    }

}



public class MaineCoon : Gato

{

    public MaineCoon(string color, string tipoPelo, string colorOjos) :
base(color, tipoPelo, colorOjos) { }
```

```
    public override void Comer()
    {
        Console.WriteLine("El Maine Coon está comiendo.");
    }

    public override void Maullar()
    {
        Console.WriteLine("El Maine Coon maúlla fuertemente.");
    }

    public override void Dormir()
    {
        Console.WriteLine("El Maine Coon duerme en su sofá.");
    }
}

public class Persa : Gato
{
    public Persa(string color, string tipoPelo, string colorOjos) :
base(color, tipoPelo, colorOjos) { }

    public override void Comer()
    {
        Console.WriteLine("El Persa está comiendo.");
    }
```

```
    public override void Maullar()
    {
        Console.WriteLine("El Persa maúlla delicadamente.");
    }

    public override void Dormir()
    {
        Console.WriteLine("El Persa duerme plácidamente en su cama de lujo.");
    }
}

public class Tarea39484
{
    public static void Main()
    {
        Siames siames = new Siames("Blanco", "Corto", "Azul");
        Console.WriteLine($"Características del Siames: Color - {siames.Color}, Tipo de pelo - {siames.TipoPelo}, Color de ojos - {siames.ColorOjos}");
        siames.Comer();
        siames.Maullar();
        siames.Dormir();

        Console.WriteLine();
```

```
        MaineCoon maineCoon = new MaineCoon("Marrón", "Largo",
"Verde");

        Console.WriteLine($"Características del Maine Coon: Color -
{maineCoon.Color}, Tipo de pelo - {maineCoon.TipoPelo}, Color de ojos -
{maineCoon.ColorOjos}");

        maineCoon.Comer();

        maineCoon.Maullar();

        maineCoon.Dormir();



        Console.WriteLine();



        Persa persa = new Persa("Crema", "Largo", "Naranja");

        Console.WriteLine($"Características del Persa: Color -
{persa.Color}, Tipo de pelo - {persa.TipoPelo}, Color de ojos -
{persa.ColorOjos}");

        persa.Comer();

        persa.Maullar();

        persa.Dormir();

    }

}
```

```
PS C:\Users\zerol\Downloads\Software> ^C
PS C:\Users\zerol\Downloads\Software>
PS C:\Users\zerol\Downloads\Software>  & 'c:\Users\zerol\.vscode\extensions\ms-dotnettools.csharp-2.55.29-win32-x
terpreter=vscode' '--connection=9680ff4739be4a7aaaaf4082d6eae9ad'
Contador de gatos: 1
Características del Siames: Color - Blanco, Tipo de pelo - Corto, Color de ojos - Azul
El Siames está comiendo.
El Siames maúlla suavemente.
El Siames duerme en su cama favorita.

Contador de gatos: 2
Características del Maine Coon: Color - Marrón, Tipo de pelo - Largo, Color de ojos - Verde
El Maine Coon está comiendo.
El Maine Coon maúlla fuertemente.
El Maine Coon duerme en su sofá.

Contador de gatos: 3
Características del Persa: Color - Crema, Tipo de pelo - Largo, Color de ojos - Naranja
El Persa está comiendo.
El Persa maúlla delicadamente.
El Persa duerme plácidamente en su cama de lujo.
PS C:\Users\zerol\Downloads\Software>
```

Una interfaz solo declara la firma de los métodos y propiedades (es decir, el nombre, los parámetros y el tipo de retorno), pero no proporciona el código que realiza la acción. Las clases que implementan la interfaz deben proporcionar la implementación de estos métodos.

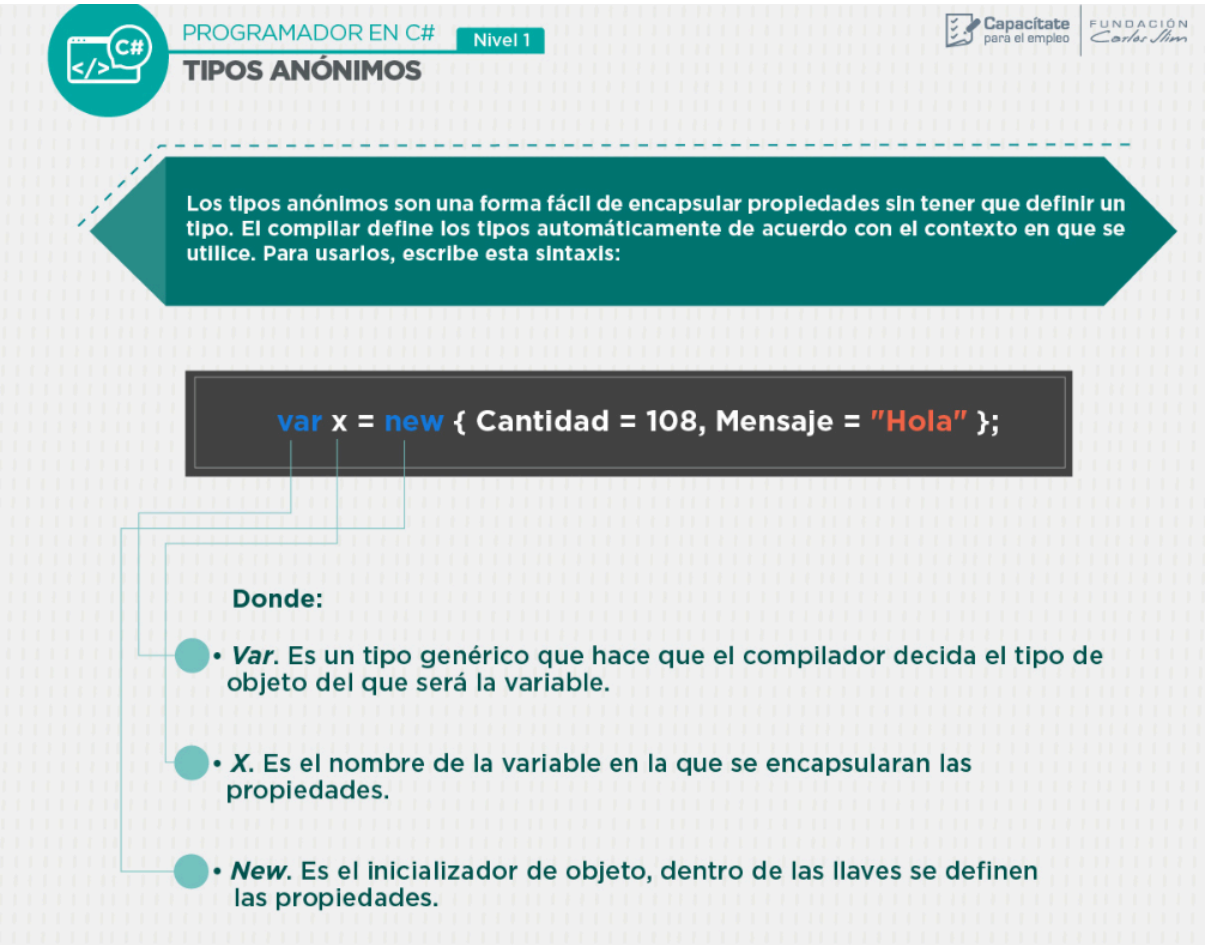
- Las interfaces no contienen lógica o implementación de los métodos. Solo especifican qué métodos deben ser implementados.
- Una clase en C# puede implementar varias interfaces, lo que le permite adoptar múltiples comportamientos o funcionalidades. Esto resuelve la limitación de la herencia simple, en la que una clase sólo puede heredar de una única clase base.
- Las interfaces se definen con la palabra clave interface, seguida de un nombre y los métodos que se deben implementar.

Una colección es un tipo de estructura de datos que agrupa múltiples elementos, generalmente de manera organizada y gestionada. Las colecciones permiten almacenar, acceder, manipular y gestionar datos de forma eficiente. Hay listas, diccionarios, conjuntos, colas, pilas y listas enlazadas.

Un delegado puede invocar un método, lo que permite un enfoque flexible para llamar a métodos sin necesidad de que la clase que los define conozca detalles de cómo se implementan los métodos. Los delegados permiten asociar un método con un objeto en tiempo de ejecución y, por lo tanto, encapsular un comportamiento específico de una clase.

Un evento es un mensaje que un objeto emite para informar a otros objetos que algo ha sucedido. Los eventos están basados en delegados y representan un tipo seguro para notificar la ocurrencia de acciones. Tiene un emisor y un suscriptor también. Son utilizados principalmente para las interfaces gráficas del usuario o GUI.

Los valores de referencia almacenan una dirección de memoria en el heap donde reside el valor real. Los cambios realizados en un objeto a través de una referencia afectan a todas las referencias que apuntan al mismo objeto. Por lo general, se usan para trabajar con datos grandes y complejos de manera eficiente, compartiendo el mismo valor entre diferentes partes del programa sin copiarlo.



```
using System;


namespace MatematicasDaniel
{
    public interface IOperaciones
    {
        double[] Suma(double[] v1, double[] v2);
        double[] Resta(double[] v1, double[] v2);
        double[] Multiplicacion(double[] v1, double[] v2);
        double[] Division(double[] v1, double[] v2);
    }

    public class Operaciones : IOperaciones
    {
        public double[] Suma(double[] v1, double[] v2)
        {
            return AplicarOperacion(v1, v2, (x, y) => x + y);
        }

        public double[] Resta(double[] v1, double[] v2)
        {
            return AplicarOperacion(v1, v2, (x, y) => x - y);
        }

        public double[] Multiplicacion(double[] v1, double[] v2)
```

```
            {
                return AplicarOperacion(v1, v2, (x, y) => x * y);
            }

            public double[] Division(double[] v1, double[] v2)
            {
                return AplicarOperacion(v1, v2, (x, y) =>
                {
                    if (y == 0)
                        throw new DivideByZeroException("ERROR,
DIVISION ENTRE 0");
                    return x / y;
                });
            }

            private double[] AplicarOperacion(double[] v1, double[]
v2, Func<double, double, double> operacion)
            {
                if (v1.Length != v2.Length)
                    throw new ArgumentException("Los vectores deben
tener la misma longitud");

                double[] resultado = new double[v1.Length];
                for (int i = 0; i < v1.Length; i++)
                {
                    resultado[i] = operacion(v1[i], v2[i]);
```

```
                }

                return resultado;

            }

        }


        public class Program

        {

            public delegate double[] OperacionDelegate(double[] v1,
double[] v2);


            public static double[] Apply(OperacionDelegate operacion,
double[] v1, double[] v2)

            {

                return operacion(v1, v2);

            }


            public static void Main()

            {


                Operaciones operaciones = new Operaciones();


                double[] vector1 = { 2.0, 4.0, 6.0, 8.0, 10.0 };

                double[] vector2 = { 1.0, 2.0, 3.0, 4.0, 5.0 };


                OperacionDelegate suma = operaciones.Suma;

                OperacionDelegate resta = operaciones.Resta;
```

```
                OperacionDelegate multiplicacion =
operaciones.Multiplicacion;

                OperacionDelegate division = operaciones.Division;



                Console.WriteLine("Suma: " + string.Join(", ",
Apply(suma, vector1, vector2)));

                Console.WriteLine("Resta: " + string.Join(", ",
Apply(resta, vector1, vector2)));

                Console.WriteLine("Multiplicación: " + string.Join(",
", Apply(multiplicacion, vector1, vector2)));



                try

                {

                    Console.WriteLine("División: " + string.Join(",
", Apply(division, vector1, vector2)));

                }

                catch (DivideByZeroException ex)

                {

                    Console.WriteLine($"Error en la división:
{ex.Message}");

                }

            }

        }

    }
```

```
PS C:\Users\zerol\Downloads\Software> ^C
PS C:\Users\zerol\Downloads\Software>
PS C:\Users\zerol\Downloads\Software>  & 'c:\Users\zerol\.vscode\extensions\ms-dotnettools.csharp-2.55.29-win32-x64\.debugger\x86_64\vsdbg.exe' '--interpreter=vscode' '--connection=8c0ce1dafae54940b00cd30f92e2e47c'
Suma: 3, 6, 9, 12, 15
Resta: 1, 2, 3, 4, 5
Multiplicación: 2, 8, 18, 32, 50
División: 2, 2, 2, 2, 2
PS C:\Users\zerol\Downloads\Software>
```