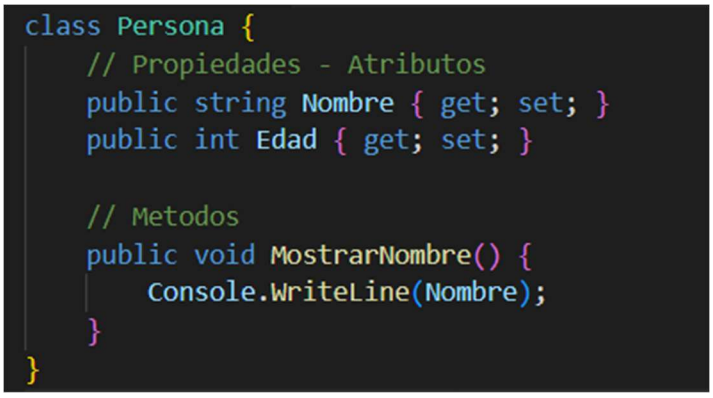
Programacion Orientada a Objetos

Que es una clase?

En la programación orientada a objetos (POO), una clase es un modelo o plantilla para crear objetos. Una clase define las propiedades y métodos que debe tener un objeto de esa clase. En C#, una clase se define mediante la palabra clave "class", seguida del nombre de la clase y un conjunto de llaves.

Por ejemplo, el siguiente código define una clase llamada "Persona":



```
class Persona {
    // Propiedades - Atributos
    public string Nombre { get; set; }
    public int Edad { get; set; }

    // Metodos
    public void MostrarNombre() {
        Console.WriteLine(Nombre);
    }
}
```

En este ejemplo, la clase "Persona" tiene dos propiedades, "Nombre" y "Edad". Estos se definen mediante el modificador de acceso "public", lo que significa que se puede acceder a ellos y modificarlos desde fuera de la clase. La clase también tiene un método llamado "MostrarNombre" que es un método vacío, no devuelve ningún valor y solo imprime el valor de la propiedad Nombre.

Una clase debe contener las siguientes partes:

Propiedades: Definen los datos que debe tener un objeto de la clase. Las propiedades generalmente se declaran como campos y se encapsulan con métodos getter y setter.

Métodos: Estos definen el comportamiento que debe tener un objeto de la clase. Los métodos generalmente se definen como funciones que se pueden llamar en un objeto.
Constructores: Estos son métodos especiales que se llaman cuando se crea un objeto de la clase. Se utilizan para inicializar las propiedades del objeto.
Destructores: estos son métodos especiales que se llaman cuando se destruye un objeto de la clase, se usan para liberar cualquier recurso que tenga el objeto.

También es importante notar que una clase puede tener diferentes modificadores de acceso, como público, privado o protegido. Las clases públicas son accesibles desde cualquier código, las clases privadas solo son accesibles dentro de la clase y las clases protegidas solo son accesibles dentro de la clase o una clase derivada.

Modificadores de acceso:

Los modificadores de acceso en C# son palabras clave que determinan el nivel de acceso a una clase, método o propiedad. Controlan quién puede acceder a los miembros de una clase y qué pueden hacer con ellos.

Los principales modificadores de acceso en C# son:

- **Público (public):** se puede acceder a un miembro público desde cualquier lugar, con cualquier código. Significa que cualquier otra clase puede acceder al miembro, independientemente de dónde se encuentre.
- **Privado (private):** solo se puede acceder a un miembro privado dentro de la misma clase donde está definido. Significa que solo la clase en la que está definido puede acceder al miembro y ninguna otra clase puede acceder a él.
- **Protegido (protected):** solo se puede acceder a un miembro protegido dentro de la misma clase donde está definido, o por cualquier clase derivada. Significa que solo

puede acceder al miembro la clase en la que está definido, o cualquier otra clase que herede de él.

Por ejemplo, si tiene una clase llamada "Persona" y tiene una propiedad privada llamada "Edad", solo puede cambiar el valor de esta propiedad dentro de la clase Persona. Si intenta cambiarlo fuera de la clase, obtendrá un error. Por otro lado, si la propiedad fuera pública, podrías cambiarla de cualquier otra clase.

En general, es una buena práctica hacer que los miembros sean privados a menos que sea necesario acceder a ellos desde otras clases, de esta manera se encapsulan los datos y el comportamiento de la clase y se evitan cambios inesperados o accesos no deseados.

```
class Persona {
    // Atributo Privado
    private int Edad;
    // Atributo Publico
    public string Nombre { get; set; }
    // Atributo Protegido
    protected string Direccion;

    // Metodo Publico
    public void MostrarNombre() {
        Console.WriteLine(Nombre);
    }
    // Metodo Privado
    private void CambiarEdad(int edad) {
        if (edad > 0) {
          this.edad = edad;
        }
    }
    // Metodo Protegido
    protected void CambiarDireccion(string direccion) {
        this.Direccion = direccion;
    }
}

class Estudiante : Persona {
    public void CambiarDireccionEstudiante(string direccion) {
        // Metodo Protegido puede ser accesado
        CambiarDireccion(direccion);
    }
}
```

```
class Program {
    static void Main(string[] args) {
        Persona persona = new Persona();
        // se puede acceder a la propiedad pública
         persona.Nombre = "Juan Pérez";
         // no se puede acceder a la propiedad privada
         // persona.edad = 30; // Error: la edad es inaccesible debido a su nivel de protección
         // no se puede acceder al método privado
         // persona.CambiarEdad(30); // Error: CambiarEdad es inaccesible debido a su nivel de protección
         // no se puede acceder a la propiedad protegida
         // persona.dirección = "En algún lugar"; // Error: la dirección es inaccesible debido a su nivel de protección
         // no se puede acceder al método protegido
         // persona.CambiarDireccion("En algún lugar"); // Error: SetAddress es inaccesible debido a su nivel de protección

         // se puede acceder al método público
        person.MostrarNombre(); // Muestra: Juan Pérez

        Estudiante alumno = new Estudiante();
        // Metodo Protegido puede ser accesado
        alumno.CambiarDireccionEstudiante("la direccion");
    }
}
```

En este ejemplo, la clase "Persona" tiene una propiedad privada llamada "Edad", una propiedad pública llamada "Nombre" y una propiedad protegida llamada "Direccion". También tiene un método público llamado " MostrarNombre", un método privado llamado " CambiarEdad" y un método protegido llamado " CambiarDireccion".

La clase "Estudiante" hereda de la clase "Persona", y tiene un método público llamado " CambiarDireccionEstudiante", que llama al método protegido " CambiarDireccion" de la clase base.

En el método principal, puede ver que se puede acceder a la propiedad pública y al método desde cualquier lugar, pero no se puede acceder a la propiedad privada, el método privado y la propiedad protegida desde fuera de la clase. Además, puede ver que se puede acceder al método protegido desde la clase derivada.

Que es un objeto?

En la programación orientada a objetos (POO), un objeto es una instancia de una clase. Una clase es un modelo o plantilla para crear objetos. En C#, un objeto se crea mediante la palabra

clave 'new', seguida del nombre de la clase y los parámetros de construcción necesarios. Por ejemplo, el siguiente código crea un objeto de la clase "Persona":

```
class Persona {
    public string Nombre { get; set; }
    public int Edad { get; set; }
}

Persona persona = new Persona { Name = "Juan Perez", Age = 30 };
```

En este ejemplo, la clase "Persona" tiene dos propiedades, "Nombre" y "Edad". El objeto "persona" se crea llamando a la palabra clave "new", seguida del nombre de clase "Persona", y estableciendo los valores de las propiedades "Nombre" y "Edad" en "Juan Pérez" y 30, respectivamente.

También se puede crear un objeto de una clase llamando al constructor. Un constructor es un método especial que se llama cuando se crea un objeto.

```
class Persona {
    public string Nombre { get; set; }
    public int Edad { get; set; }
    public Persona(string nombre, int edad) {
        Nombre = nombre;
        Edad = edad;
    }
}

Persona persona = new Persona("Maria Vargas", 30);
```

En este ejemplo, la clase "Persona" tiene un constructor que acepta dos argumentos, "nombre" y "edad". Al crear el objeto "persona", los valores "Maria Vargas" y 30 se pasan al constructor, que los asigna a las propiedades "Nombre" y "Edad" del objeto.

Principios de la Programacion Orientada a Objetos

Los principios de la programación orientada a objetos (POO) son un conjunto de pautas que ayudan a diseñar y organizar el código de una manera que sea fácil de entender, mantener y ampliar. Los principios fundamentales de la programación orientada a objetos son:

- **Encapsulación**: este principio se refiere a la práctica de ocultar los detalles de implementación de una clase y exponer solo la información necesaria al mundo exterior. Esto permite un mayor nivel de control sobre el estado interno de la clase y hace que el código sea más robusto y menos propenso a errores. En C#, la encapsulación se logra mediante el uso de modificadores de acceso (como privado, protegido y público) para controlar la visibilidad de los miembros de la clase.

- **Herencia**: este principio se refiere a la capacidad de una clase para heredar propiedades y métodos de una clase principal. Esto permite la creación de una jerarquía de clases, donde una clase derivada puede heredar las propiedades y métodos de una clase base y también puede agregar nuevas propiedades y métodos propios. En C#, la herencia se logra usando la palabra clave "clase" y la notación ": base".

- **Polimorfismo**: este principio se refiere a la capacidad de un objeto para adoptar múltiples formas. Esto permite que un solo método o propiedad funcione con múltiples tipos de objetos, sin necesidad de una verificación de tipo explícita. En C#, el polimorfismo se logra mediante el uso de interfaces y clases abstractas.

- **Abstracción**: este principio se refiere a la capacidad de una clase para proporcionar una interfaz simplificada a una implementación compleja. Esto permite un mayor nivel de

control sobre el estado interno de la clase y hace que el código sea más robusto y menos propenso a errores. En C#, la abstracción se logra mediante el uso de interfaces y clases abstractas.

Todos estos principios son importantes para organizar y diseñar el código de manera que sea fácil de entender, mantener y ampliar. Al utilizar estos principios, el código se vuelve más robusto, menos propenso a errores, más flexible y más reutilizable.

Herencias en c#

La herencia en C# es una característica de la programación orientada a objetos (POO) que permite que una clase herede propiedades y métodos de una clase principal. Esto permite la creación de una jerarquía de clases, donde una clase derivada puede heredar las propiedades y métodos de una clase base y también puede agregar nuevas propiedades y métodos propios.

Por ejemplo, supongamos que tiene una clase llamada "Vehículo" que tiene propiedades como " NumeroRuedas" y "VelocidadMaxima" y un método llamado "Conducir". Puede crear una clase derivada (heredada) llamada "Carro" que herede de la clase "Vehículo" y agregar nuevas propiedades como "NumeroPuertas" y "TipoMotor" y nuevos métodos como "Pitar".

```
class Vehiculo {
    public int NumeroRuedas {get; set;}
    public int VelocidadMaxima {get; set;}
    public void Conducir() {
        Console.WriteLine("The vehicle is driving");
    }
}

class Carro : Vehiculo {
    public int NumeroPuertas {get; set;}
    public string TipoMotor {get; set;}
    public void Pitar() {
        Console.WriteLine("PEEEP PEEEP!");
    }
}
```

En este ejemplo, la clase "Carro" hereda de la clase "Vehiculo" usando la notación ": base". La clase "Carro" tiene acceso a todas las propiedades y métodos de la clase "Vehículo", y también tiene acceso a sus propias propiedades y métodos.

También puede usar la palabra clave "base" para llamar a los métodos de la clase principal o al constructor, por ejemplo:

```
class Car : Vehicle {
    public Car(int numeroRuedas, int velocidadMaxima, int numeroPuertas, string tipoMotor) {
        base.NumeroRuedas = numeroRuedas;
        base.VelocidadMaxima = velocidadMaxima;
        this.NumeroPuertas = numeroPuertas;
        this.TipoMotor = tipoMotor;
    }
}
```

De esta forma, puede crear una jerarquía de clases, donde cada clase hereda propiedades y métodos de la clase principal y también puede agregar nuevas propiedades y métodos propios.

Esto permite un uso más eficiente del código y mejora la capacidad de mantenimiento del código.

Clases abstractas

Una clase abstracta en C# es una clase de la que no se puede crear una instancia, sirve como clase base para otras clases, no se puede crear una instancia directamente. Las clases abstractas se utilizan para proporcionar una interfaz común y una implementación para las clases derivadas. Las clases abstractas generalmente se usan como clase base para otras clases que comparten un comportamiento común.

Por ejemplo, supongamos que tiene una clase llamada "Forma" que tiene propiedades como "Color" y "Área" y un método llamado "Dibujar". Puede crear una clase abstracta llamada "Forma" que tenga estas propiedades y métodos, y también un método abstracto llamado "CalcularArea" que debe ser implementado por las clases derivadas.

```
abstract class Forma {
    public string Color {get; set;}
    public double Area {get; set;}
    public void Dibujar() {
        Console.WriteLine("Dibujando la figura");
    }
    public abstract double CalcularArea();
}

class Rectangulo : Forma {
    public double Largo {get; set;}
    public double Ancho {get; set;}
    public override double CalcularArea() {
        return Largo * Ancho;
    }
}
```

En este ejemplo, la clase "Forma" se define como una clase abstracta y tiene propiedades como "Color" y "Área" y un método llamado "Dibujar". También tiene un método abstracto llamado " CalcularArea ", que debe ser implementado por cualquier clase que herede de la clase "Forma". La clase "Rectangulo" hereda de la clase "Forma" y está implementando el método CalcularArea.

Es importante notar que no puede crear una instancia de una clase abstracta, pero puede crear una variable del tipo de clase abstracta y asignar una instancia de una clase derivada.

```
Forma rectangulo = new Rectangulo();
rectangulo.Largo = 10;
rectangulo.Ancho = 20;
Console.WriteLine(rectangulo.CalcularArea()); //200
```

Temas Pendientes:

Polimorfismo

Interfaces

Tipos de datos

Ciclos

Switch

Try catch