3) ¿Por qué no funciona el siguiente código? ¿Cómo se puede solucionar fácilmente?

```
class Auto
{
    double velocidad;
    public virtual void Acelerar()
        => Console.WriteLine("Velocidad = {0}", velocidad += 10);
}

class Taxi : Auto
{
    public override void Acelerar()
        => Console.WriteLine("Velocidad = {0}", velocidad += 5);
}
```

El código en C# tiene un error de sintaxis en las expresiones de cuerpo de las funciones (=>) utilizadas en los métodos Acelerar() de las clases Auto y Taxi. En C#, cuando se utiliza la sintaxis de expresiones de cuerpo (=>) en un método, se requiere que la expresión de cuerpo esté dentro de llaves ({}) si la expresión es más compleja que una sola línea.

Para solucionar el error, puedes ajustar el código de la siguiente manera:

```
class Auto
{
    double velocidad;
    public virtual void Acelerar()
    {
        Console.WriteLine("Velocidad = {0}", velocidad += 10);
    }
}

class Taxi : Auto
{
    public override void Acelerar()
    {
        Console.WriteLine("Velocidad = {0}", velocidad += 5);
    }
}
```

4) Contestar sobre el siguiente programa:

```
Taxi t = new Taxi(3);
Console.WriteLine($"Un {t.Marca} con {t.Pasajeros} pasajeros");

class Auto
{
   public string Marca { get; private set; } = "Ford";
   public Auto(string marca) => this.Marca = marca;
   public Auto() { }
}

class Taxi : Auto
{
   public int Pasajeros { get; private set; }
   public Taxi(int pasajeros) => this.Pasajeros = pasajeros;
}
```

¿Por qué no es necesario agregar **`:base`** en el constructor de **`Taxi`**? Eliminar el segundo constructor de la clase **`Auto`** y modificar la clase **`Taxi`** para el programa siga funcionando

El motivo por el cual no es necesario agregar :base en el constructor de Taxi se debe a que el constructor de Taxi está utilizando el constructor sin parámetros de la clase base Auto de forma implícita. Cuando no se especifica un constructor base utilizando :base en la definición de un constructor en una clase derivada, el compilador de C# automáticamente asume que se está llamando al constructor sin parámetros de la clase base.

En este caso, la clase Auto tiene dos constructores: uno con parámetros Auto(string marca) y otro sin parámetros Auto(). Si no se especifica ningún constructor base en la definición del constructor Taxi(int pasajeros), el compilador de C# asume que se está llamando al constructor sin parámetros de Auto.

Por lo tanto, el código funciona correctamente sin necesidad de agregar :base en el constructor de Taxi.

Para eliminar el segundo constructor de la clase Auto y hacer que el programa siga funcionando, puedes modificar la clase Taxi de la siguiente manera:

```
class Taxi : Auto
{
   public Taxi(int pasajeros) => this.Pasajeros = pasajeros;
   public int Pasajeros { get; private set; }
}
```

En este caso, el constructor Taxi(int pasajeros) se encarga de inicializar la propiedad Pasajeros de la clase Taxi, y no es necesario llamar explícitamente a ningún constructor de

la clase base Auto, ya que el compilador asume implícitamente que se está llamando al constructor sin parámetros de Auto.

5) ¿Qué líneas del siguiente código provocan error de compilación y por qué?

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

public class Auto
{
    private Persona _dueño1, _dueño2;
    public Persona GetPrimerDueño() => _dueño1;
    protected Persona SegundoDueño
    {
        set => _dueño2 = value;
    }
}
```

7) Ofrecer una implementación polimórfica para mejorar el siguiente programa:

```
Imprimidor.Imprimir(new A(), new B(), new C(), new D());

class A {
    public void ImprimirA() => Console.WriteLine("Soy una instancia A");
}

class B {
    public void ImprimirB() => Console.WriteLine("Soy una instancia B");
}

class C {
    public void ImprimirC() => Console.WriteLine("Soy una instancia C");
}

class D {
    public void ImprimirD() => Console.WriteLine("Soy una instancia D");
}

static class Imprimidor {
    public static void Imprimir(params object[] vector) {
        foreach (object o in vector) {
            if (o is A) { (o as A)?.ImprimirA(); }
            else if (o is B) { (o as B)?.ImprimirB(); }
            else if (o is C) { (o as C)?.ImprimirC(); }
            else if (o is D) { (o as D)?.ImprimirD(); }
        }
    }
}
```

```
interface IImprimible
{
    void Imprimir();
}

class A : IImprimible
{
    public void Imprimir() => Console.WriteLine("Soy una instancia A");
}

class B : IImprimible
{
    public void Imprimir() => Console.WriteLine("Soy una instancia B");
}

class C : IImprimible
{
    public void Imprimir() => Console.WriteLine("Soy una instancia C");
}

class D : IImprimible
{
    public void Imprimir() => Console.WriteLine("Soy una instancia D");
}

static class Imprimidor
{
    public static void Imprimir(params IImprimible[] vector)
    {
        foreach (IImprimible item in vector)
        {
            item.Imprimir();
        }
    }
}
```

En esta implementación, se utiliza una interfaz IImprimible que define el método Imprimir(). Luego, cada una de las clases A, B, C, D implementa la interfaz IImprimible y proporciona su propia implementación del método Imprimir(). La clase Imprimidor ahora acepta un arreglo de objetos que implementan IImprimible en lugar de usar object, lo que permite el uso del polimorfismo para imprimir cada objeto sin necesidad de realizar comprobaciones de tipo. Esto hace que el código sea más escalable y extensible, ya que se puede agregar fácilmente más clases que implementen IImprimible sin tener que modificar el código del Imprimidor.

8) Crear un programa para gestionar empleados en una empresa. Los empleados deben tener las propiedades públicas de sólo lectura **`Nombre`**, **`DNI`**, **`FechaDeIngreso`**, **`SalarioBase`** y **`Salario`**. Los valores de estas propiedades (a excepción de **`Salario`** que es una propiedad calculada) deben establecerse por medio de un constructor adecuado.
Existen dos tipos de empleados: **`Administrativo`** y **`Vendedor`**. No se podrán crear objetos de la clase padre **`Empleado`**, pero sí de sus clases hijas (**`Administrativo`** y **`Vendedor`**). Aparte de las propiedades de solo lectura mencionadas, el administrativo tiene otra propiedad pública de lectura/escritura llamada **`Premio`** y el vendedor tiene otra propiedad pública de lectura/escritura llamada **`Comision`**.
La propiedad de solo lectura **`Salario`**, se calcula como el salario base más la comisión o el premio según corresponda.
Las clases tendrán además un método público llamado **`AumentarSalario()`** que tendrá una implementación distinta en cada clase. En el caso del administrativo se incrementará el salario base en un 1% por cada año de antigüedad que posea en la empresa, en el caso del vendedor se incrementará el salario base en un 5% si su antigüedad es inferior a 10 años o en un 10% en caso contrario.
El siguiente código (ejecutado el día 9/4/2022) debería mostrar en la consola el resultado indicado:

```
Empleado[] empleados = new Empleado[] {
  new Administrativo("Ana", 20000000, DateTime.Parse("26/4/2018"), 10000) {Premio=1000},
  new Vendedor("Diego", 30000000, DateTime.Parse("2/4/2010"), 10000) {Comision=2000},
  new Vendedor("Luis", 33333333, DateTime.Parse("30/12/2011"), 10000) {Comision=2000}
};
foreach (Empleado e in empleados)
{
   Console.WriteLine(e);
   e.AumentarSalario();
   Console.WriteLine(e);
}
```

Salida por consola

```
Administrativo Nombre: Ana, DNI: 20000000 Antigüedad: 3
Salario base: 10000, Salario: 11000
------------
Administrativo Nombre: Ana, DNI: 20000000 Antigüedad: 3
Salario base: 10300, Salario: 11300
------------
Vendedor Nombre: Diego, DNI: 30000000 Antigüedad: 12
Salario base: 10000, Salario: 12000
------------
Vendedor Nombre: Diego, DNI: 30000000 Antigüedad: 12
Salario base: 11000, Salario: 13000
------------
Vendedor Nombre: Luis, DNI: 33333333 Antigüedad: 10
Salario base: 10000, Salario: 12000
------------
Vendedor Nombre: Luis, DNI: 33333333 Antigüedad: 10
Salario base: 11000, Salario: 13000
------------
```

Recomendaciones: Observar que el método **`AumentarSalario()`** y la propiedad de solo lectura **`Salario`** en la clase **`Empleado`** pueden declararse como abstractos. Intentar no utilizar campos sino propiedades auto-implementadas todas las veces que sea posible. Además sería deseable que la propiedad **`SalarioBase`** definida en **`Empleado`** sea pública para la lectura y protegida para la escritura, para que pueda establecerse desde las subclases **`Administrativo`** y **`Vendedor`**.

```
using System;

abstract class Empleado
{
        public string Nombre { get; }
        public int DNI { get; }
        public DateTime FechaDeIngreso { get; }
        public decimal SalarioBase { get; }
        public abstract decimal Salario { get; }

        protected Empleado(string nombre, int dni, DateTime fechaDeIngreso, decimal
    salarioBase)
        {
            Nombre = nombre;
            DNI = dni;
            FechaDeIngreso = fechaDeIngreso;
            SalarioBase = salarioBase;
        }

        public abstract void AumentarSalario();

        public override string ToString()
        {
            return $"Nombre: {Nombre}, DNI: {DNI}, Fecha de Ingreso: {FechaDeIngreso}, Salario
    Base: {SalarioBase}, Salario: {Salario}";
        }
}

class Administrativo : Empleado
{
        public decimal Premio { get; set; }

        public Administrativo(string nombre, int dni, DateTime fechaDeIngreso, decimal
    salarioBase)
            : base(nombre, dni, fechaDeIngreso, salarioBase)
        {
            Premio = 0;
        }

        public override decimal Salario => SalarioBase + Premio;

        public override void AumentarSalario()
        {
            int antiguedad = DateTime.Now.Year - FechaDeIngreso.Year;
            SalarioBase += SalarioBase * (antiguedad * 0.01m);
        }
}

class Vendedor : Empleado
{
        public decimal Comision { get; set; }
```

```
        public Vendedor(string nombre, int dni, DateTime fechaDeIngreso, decimal salarioBase)
            : base(nombre, dni, fechaDeIngreso, salarioBase)
        {
            Comision = 0;
        }

        public override decimal Salario => SalarioBase + Comision;

        public override void AumentarSalario()
        {
            int antiguedad = DateTime.Now.Year - FechaDeIngreso.Year;
            if (antiguedad < 10)
            {
                SalarioBase += SalarioBase * 0.05m;
            }
            else
            {
                SalarioBase += SalarioBase * 0.10m;
            }
        }
}

class Program
{
    static void Main()
    {
        Empleado[] empleados = new Empleado[] {
            new Administrativo("Ana", 20000000, DateTime.Parse("26/4/2018"), 10000) {Premio=1000},
            new Vendedor("Diego", 30000000, DateTime.Parse("2/4/2010"), 10000) {Comision=2000},
            new Vendedor("Luis", 33333333, DateTime.Parse("30/12/2011"), 10000) {Comision=2000}
        };

        foreach (Empleado e in empleados)
        {
            Console.WriteLine(e);
            e.AumentarSalario();
            Console.WriteLine(e);
        }
    }
}
```

En esta implementación, se utiliza una clase abstracta Empleado como clase padre para las clases hijas Administrativo y Vendedor. La clase Empleado tiene propiedades públicas de solo lectura Nombre, DNI, FechaDeIngreso, SalarioBase y una propiedad abstracta Salario que debe ser implementada en las clases hijas.