

EL MÁS IMPORTANTE

CHULETA RA – POO C# (ESTILO PROFE)

(Usar en examen sin internet. Todo lo que suele pedir está aquí.)

1 Patrón general que SIEMPRE repite

1. Clase base abstracta:

- String con propiedad automática (`Nombre`, `Descripcion`, `Matricula...`).
- Números con propiedad NO automática + validación (si negativo → `0.0`).
- 1 o más propiedades de solo lectura calculadas.
- Método polimórfico `virtual` con cálculo por defecto.
- `ToString()` con info común + `GetType().Name`.

2. Clases derivadas:

- Heredan de la base (y a veces hay clases intermedias abstractas).
- Añaden campos con validación.
- Hacen `override` del método polimórfico con su fórmula.
- Amplían `ToString()` llamando a `base.ToString()`.

3. Programa principal:

- `List<Base>` (lista polimórfica).
- Menú con 4 opciones típicas: crear, ver, total, salir.
- Métodos: `CrearX`, `VerX`, `CalcularTotal`.
- Uso de LINQ: `Any()`, `Sum()`, `Select(...).ToList().ForEach(...)`.

- Método LeerDouble que si falla → 0.0.

Si clavas esto, el ejercicio no se rompe y su estilo lo tienes clonado.

2 Clase base – PLANTILLA

Adáptala al contexto: Empleado, Envío, Vehículo, etc.

```
abstract class Elemento {  
    // Propiedad automática para texto  
    public string Nombre { get; set; }  
  
    // Propiedad NO automática con validación (negativo -> 0.0)  
    double valorBase;  
    public double ValorBase {  
        get => valorBase;  
        set => valorBase = value < 0.0 ? 0.0 : value;  
    }  
  
    // Propiedad solo Lectura calculada (ejemplo, cambia según  
    // enunciado)  
    public double CosteBase => ValorBase * 2.0;  
  
    // Constructor protegido: inicializa comunes  
    protected Elemento(string nombre, double valorBase) {  
        Nombre = nombre ?? string.Empty;  
        ValorBase = valorBase;    // usa la validación del setter  
    }  
  
    // Método polimórfico: se puede usar tal cual o sobreescribir  
    public virtual double CalcularTotal() => CosteBase;  
  
    // ToString común (usa GetType para mostrar tipo real)  
    public override string ToString() =>  
        $"{GetType().Name} | Nombre: {Nombre} | ValorBase:  
        {ValorBase:0.00}";  
}
```

Fíjate: esto es exactamente como:

- Empleado (2.9)

- [Envio](#) (2.10)
 - [Vehiculo](#) (2.11)
-

3 Clases derivadas – PLANTILLAS

Tipo 1: suma algo (EmpleadoFijo / PaqueteEstandar)

```
class TipoA : Elemento {
    double extra;
    public double Extra {
        get => extra;
        set => extra = value < 0.0 ? 0.0 : value;
    }

    public TipoA(string nombre, double valorBase, double extra)
        : base(nombre, valorBase) {
        Extra = extra;
    }

    public override double CalcularTotal() =>
        CosteBase + Extra;

    public override string ToString() =>
        $"{base.ToString()} | Extra: {Extra:0.00} | Total:
{CalcularTotal():0.00}";
}
```

Tipo 2: fórmula distinta (EmpleadoPorHora / PaqueteExpress / Camion / Autobus)

```
class TipoB : Elemento {
    double factor;
    public double Factor {
        get => factor;
        set => factor = value < 0.0 ? 0.0 : value;
    }

    public TipoB(string nombre, double valorBase, double factor)
        : base(nombre, valorBase) {
        Factor = factor;
    }
```

```

public override double CalcularTotal() =>
    CosteBase * Factor; // o la fórmula que diga el enunciado

public override string ToString() =>
    $"{base.ToString()} | Factor: {Factor:0.00} | Total:
{CalcularTotal():0.00}";
}

```

Clases intermedias (como en 2.11)

Cuando te pida jerarquía tipo:

- `Vehiculo` → `TransportePasajeros` → `Autobus`
- `Vehiculo` → `TransporteCarga` → `Camion`

Copia el patrón:

```

abstract class Intermedia : Elemento {
    double dato;
    public double Dato {
        get => dato;
        set => dato = value < 0.0 ? 0.0 : value;
    }

    protected Intermedia(string nombre, double valorBase, double dato)
        : base(nombre, valorBase) {
        Dato = dato;
    }

    public override string ToString() =>
        $"{base.ToString()} | Dato: {Dato:0.00}";
}

```

Y luego la hija final hace su fórmula con `override`.

4] Validaciones (para que NO se rompa)

Tal como hace tu profe:

- Todo número crítico:
 - En setter: `value < 0.0 ? 0.0 : value;`
- Entradas de usuario:
 - Usar `LeerDouble`. Si falla → `0.0`.
 - No hace falta explotar el programa ni lanzar excepciones.

```
static double LeerDouble(string prompt) {
    Console.Write(prompt);
    var raw = Console.ReadLine();
    if (double.TryParse(raw, out double valor))
        return valor;
    Console.WriteLine("Entrada no numérica. Se asignará 0.0.");
    return 0.0;
}
```

Con esto + setters, aunque el usuario meta basura, el programa sigue vivo.

5 Colección + menú – PLANTILLA

```
class Program {
    static void Main() {
        var elementos = new List<Elemento>();

        while (true) {
            Console.WriteLine();
            Console.WriteLine("== Sistema X ==");
            Console.WriteLine("1) Crear elemento");
            Console.WriteLine("2) Ver elementos");
            Console.WriteLine("3) Calcular total");
            Console.WriteLine("4) Salir");
            Console.Write("Opción (1-4): ");
            var op = Console.ReadLine()?.Trim();

            switch (op) {
                case "1": CrearElemento(elementos); break;
                case "2": VerElementos(elementos); break;
                case "3": CalcularTotal(elementos); break;
                case "4": return;
            }
        }
    }
}
```

```
        default: Console.WriteLine("Opción no válida."); break;
    }
}
}

static void CrearElemento(List<Elemento> elementos) {
    Console.WriteLine("Tipo:");
    Console.WriteLine("a) TipoA");
    Console.WriteLine("b) TipoB");
    Console.Write("Elige (a-b): ");
    var tipo = Console.ReadLine()?.Trim().ToLower();

    Console.Write("Nombre: ");
    var nombre = Console.ReadLine() ?? string.Empty;

    var valorBase = LeerDouble("Valor base: ");

    if (tipo == "a") {
        var extra = LeerDouble("Extra: ");
        elementos.Add(new TipoA(nombre, valorBase, extra));
        Console.WriteLine("TipoA creado.");
    } else if (tipo == "b") {
        var factor = LeerDouble("Factor: ");
        elementos.Add(new TipoB(nombre, valorBase, factor));
        Console.WriteLine("TipoB creado.");
    } else {
        Console.WriteLine("Tipo no reconocido. Operación cancelada.");
    }
}

static void VerElementos(List<Elemento> elementos) {
    Console.WriteLine();
    if (!elementos.Any()) {
        Console.WriteLine("No hay elementos.");
        return;
    }

    elementos
        .Select(e => $"{e} | Total: {e.CalcularTotal():0.00}")
        .ToList()
        .ForEach(Console.WriteLine);
}

static void CalcularTotal(List<Elemento> elementos) {
    Console.WriteLine();
```

```

        var total = elementos.Sum(e => e.CalcularTotal());
        Console.WriteLine($"Total ({elementos.Count} elemento/s):
{total:0.00}");
    }

    static double LeerDouble(string prompt) {
        Console.Write(prompt);
        var raw = Console.ReadLine();
        if (double.TryParse(raw, out double v)) return v;
        Console.WriteLine("Entrada no numérica. Se asignará 0.0.");
        return 0.0;
    }
}

```

6 Checklist rápido para el examen

Antes de entregar, revisa que tu código tenga:

- Clase base **abstract**.
- Propiedades numéricas con validación ($< 0 \rightarrow 0.0$).
- Alguna propiedad **solo lectura** calculada.
- Método **virtual** (**Calcular...**) en la base.
- **override** correcto en las derivadas.
- **ToString()** en base + derivadas usando **base.ToString()**.
- **List<Base>** con objetos de distintos tipos.
- Menú con opciones claras y sin romperse en entradas raras.
- Uso de **LINQ (Any, Sum, Select)** como en los ejemplos del profe.
- Comentarios que expliquen diseño (herencia, polimorfismo, validación).

Con esta hoja puedes reconstruir cualquiera de los 2.9, 2.10, 2.11 y el simulacro cambiando nombres y fórmulas. Si quieres, en el siguiente mensaje te hago un “mini mapa mental” específico: *si enunciado habla de X → base es X, hijas son Y/Z, método polimórfico es tal*, para que lo asociés aún más rápido.

SIN HABERLE PASADO TODAVÍA EL 2.9-10-11



CHULETA RA – C# POO (para los ejercicios tipo profe)

1. Esqueleto mental del ejercicio

Casi siempre tendrás que:

1. Crear **una clase base (abstracta)** con:
 - Atributos comunes.
 - Validación.
 - Métodos polimórficos (**abstract** o **virtual**).
2. Crear **clases hijas** que:
 - Heredan de la base.
 - Añaden atributos especiales.
 - Implementan su propia fórmula (**override**).
3. Usar una **colección de la clase base** (**List<Base>**).
4. Hacer un **menú en consola** que:
 - Crea objetos.
 - Muestra info (**ToString()**).
 - Calcula totales usando polimorfismo.

Si sabes hacer esto, apruebas.

2. Clase base: patrón típico

```
// Clase base abstracta
abstract class Elemento
{
    // Propiedades comunes
    public string Nombre { get; set; }

    // Ejemplo numérico con validación
    private double _valor;
    public double Valor
    {
        get => _valor;
        set => _valor = value < 0 ? 0.0 : value; // valida
    }

    // Propiedad solo Lectura (si la piden)
    public double CosteBase => Valor * 2.0; // ejemplo

    // Métodos polimórficos
    public abstract double CalcularTotal();

    public override string ToString()
    {
        return $"Nombre: {Nombre}, Valor: {Valor}";
    }
}
```

Claves que le encantan a tu profe:

- `abstract class` → no se instancia directamente.
- Validación en `set` (si negativo → 0 o 1, según enunciado).
- Propiedades de solo lectura (`get => ...`).
- Métodos polimórficos: `abstract` o `virtual + override` en hijas.
- `ToString()` sobreescrito y usado al mostrar la colección.

3. Herencia en clases hijas

```

class Especial : Elemento
{
    private double _extra;
    public double Extra
    {
        get => _extra;
        set => _extra = value < 0 ? 0.0 : value;
    }

    // Constructor apoyado en base
    public Especial(string nombre, double valor, double extra)
    {
        Nombre = nombre;
        Valor = valor;
        Extra = extra;
    }

    public override double CalcularTotal()
    {
        return CosteBase + Extra; // o la fórmula que toque
    }

    public override string ToString()
    {
        return base.ToString() + $", Extra: {Extra}, Total:
{CalcularTotal()}";
    }
}

```

Recuerda para el examen:

- Siempre que puedas: usa **base** (constructor o **base.ToString()**).
 - Cada hija tiene **su propia fórmula** en **CalcularTotal()** / **CalcularNomina()** / **CalcularCostePorKm()**...
-

4. Encapsulación + validación (muy importante en la rúbrica)

Patrón para atributos numéricos:

```
private double _valor;

public double Valor
{
    get => _valor;
    set => _valor = value < 0 ? 0.0 : value; // o 1.0 según enunciado
}
```

Te pueden pedir:

- No permitir negativos → poner 0.0 o 1.0.
 - Esto va en TODAS las propiedades numéricas “críticas”.
-

5. Polimorfismo con colecciones

Siempre la misma idea:

```
static List<Elemento> elementos = new List<Elemento>();
```

- La lista es del **tipo base**.
- En ella metes objetos de las clases hijas.
- Llamas a métodos polimórficos sin saber el tipo real:

```
// Ver colección
static void VerColeccion()
{
    foreach (var e in elementos)
        Console.WriteLine(e); // usa ToString() polimórfico
}

// Calcular total
static void CalcularTotal()
{
    double total = elementos.Sum(e => e.CalcularTotal());
    Console.WriteLine($"Total = {total}");
}
```

Eso es EXACTAMENTE lo que hacéis con:

- Figuras → `CalcularArea()`, `CalcularPerimetro()`.
ASP-A2.8_ Simulacro de actividad ...
- Envíos → `CalcularCostoTotal()`.
ASP-A2.10_ Ejercicio de repaso ...
- Empleados → `CalcularNomina()`.
ASP-A2.9_ Ejercicio de repaso p...
- Vehículos → `CalcularCostoPorKm()`.
ASP-A2.11_ Ejercicio de repaso ...

Solo cambia la historia.

6. Menú de consola estándar

Puedes casi copiar este patrón:

```
static int LeerOpcion()
{
    int opcion;
    while (!int.TryParse(Console.ReadLine(), out opcion) || opcion < 1
    || opcion > 4)
    {
        Console.WriteLine("Opción no válida. Intente de nuevo.");
    }
    return opcion;
}

static void Menu()
{
    Console.WriteLine("1. Crear elemento");
    Console.WriteLine("2. Ver colección");
    Console.WriteLine("3. Calcular total");
    Console.WriteLine("4. Salir");
}

static void Main()
{
    bool salir = false;
```

```

while (!salir)
{
    Menu();
    int opcion = LeerOpcion();

    switch (opcion)
    {
        case 1:
            CrearElemento(); // pides datos, creas hija concreta, La añades a la lista
            break;
        case 2:
            VerColeccion();
            break;
        case 3:
            CalcularTotal();
            break;
        case 4:
            salir = true;
            break;
    }
}
}

```

CrearElemento() cambia según el enunciado:

- Preguntas tipo (Ej: “1. Estándar / 2. Express”).
 - Según elección → creas **PaqueteEstandar**, **PaqueteExpress**, etc.
 - Validar con **TryParse**, si falla → mensaje y no añades.
-

7. Cosas típicas que revisará tu profe

Ten esto en tu hoja bien visible:

- Clase base **abstract**.
- Uso de **encapsulación** (campos privados + propiedades).
- Validación de negativos.

- Métodos `abstract/virtual` + `override`.
- `ToString()` sobreescrito y usado.
- **Colección del tipo base.**
- Menú con control de errores en la opción.
- Nombres claros y fórmulas correctas del enunciado.

EJERCICIO 2.8

```

abstract class Figura {
    // Requisito técnico de polimorfismo
    public abstract double CalcularArea();
    // Requisito técnico de polimorfismo
    public abstract double CalcularPerimetro();

    protected double _area;
    protected double _perimetro;
}

// Requisito técnico de herencia
class Circulo : Figura {
    // Requisito funcional
    // Propiedad no automática
    private double _radio;
    // Requisito de calidad si valores negativos
    public double Radio { get => _radio; set => _radio = value <= 0 ? 1
        : value; }

    // Requisito técnico de propiedades de sólo Lectura.
    public double Area { get => Math.PI * Math.Pow(Radio, 2); }
    // Requisito técnico de propiedades de sólo Lectura.
    public double Perimetro { get => 2 * Math.PI * Radio; }
    // Requisito técnico de polimorfismo
    public override double CalcularArea() {
        //return Area;
        _area = Area;
        return _area;
}

```

```
}

// Requisito técnico de polimorfismo
public override double CalcularPerimetro() {
    return Perimetro;
}
// Requisito funcional ver colección
public override string ToString() {
    return $"Círculo de Radio {Radio} con área {Area} y perímetro
{Perimetro}";
}
}

// Requisito técnico de herencia
class Rectangulo : Figura {
    // Requisito funcional
    // Propiedad no automática
    private double _base;
    // Requisito de calidad si valores negativos
    public double Base { get => _base; set => _base = value <= 0 ? 1 :
value; }

    // Requisito funcional
    // Propiedad no automática
    private double _altura;
    // Requisito de calidad si valores negativos
    public double Altura { get => _altura; set => _altura = value <= 0 ?
1 : value; }

    // Requisito técnico de propiedades de sólo lectura.
    public double Area { get => Base * Altura; }
    // Requisito técnico de propiedades de sólo lectura.
    public double Perimetro { get => 2 * (Base + Altura); }
    // Requisito técnico de polimorfismo
    public override double CalcularArea() {
        return Area;
    }

    // Requisito técnico de polimorfismo
    public override double CalcularPerimetro() {
        return Perimetro;
    }
    // Requisito funcional ver colección
    public override string ToString() {
        return $"Rectángulo de Base {Base} y Altura {Altura} con área
{Area} y perímetro {Perimetro}";
}
```

```
}

// Requisito técnico de herencia
class Rombo : Figura {
    // Requisito funcional
    // Propiedad no automática
    private double _diagonalMayor;
    // Requisito de calidad si valores negativos
    public double DiagonalMayor { get => _diagonalMayor; set =>
        _diagonalMayor = value <= 0 ? 1 : value; }
    // Requisito funcional
    // Propiedad no automática
    private double _diagonalMenor;
    // Requisito de calidad si valores negativos
    public double DiagonalMenor { get => _diagonalMenor; set =>
        _diagonalMenor = value <= 0 ? 1 : value; }
    // Requisito técnico de propiedades de sólo lectura.
    public double Area { get => DiagonalMayor * DiagonalMenor / 2; }
    // Requisito técnico de propiedades de sólo lectura.
    public double Perimetro { get => 2 *
        Math.Sqrt(Math.Pow(DiagonalMayor,2) * Math.Pow(DiagonalMenor,2)); }

    public override double CalcularArea() {
        return Area;
    }

    // Requisito técnico de polimorfismo
    public override double CalcularPerimetro() {
        return Perimetro;
    }

    // Requisito funcional ver colección
    public override string ToString() {
        return $"Rombo de DiagonalMayor {DiagonalMayor} y DiagonalMenor
{DiagonalMenor} con área {Area} y perímetro {Perimetro}";
    }
}

class Programa {
    static List<Figura> figuras = new List<Figura>();
    static int LeerOpcion() {
        while (true) {
            Console.Write("Introduzca una opción entre 1 y 5; ");
            int opcion = 0;
```

```
        if (int.TryParse(Console.ReadLine(), out opcion)) {
            if (opcion >= 1 && opcion <= 5)
                return opcion;
        }
    }
}

// Requisito funcional
static void CrearFigura() {
    Console.WriteLine("Elija Circulo, Rectangulo o Rombo");
    var figura = Console.ReadLine().ToLower();

    if (figura.Equals("circulo")) {
        Console.Write("Radio: ");
        double radio;
        if (double.TryParse(Console.ReadLine(), out radio)) {
            Circulo circulo = new Circulo();
            circulo.Radio = radio;
            figuras.Add(circulo);
        } else {
            Console.WriteLine("Incorrecto. No se ha creado");
        }
    }
}

// Requisito funcional
static void VerColeccion() {
    figuras.ForEach(f => Console.WriteLine(f));
}

// Requisito funcional
static void CalcularAreaTotal() {
    Console.WriteLine($"Área total = {figuras.Sum(f => f.CalcularArea())}");
}

// Requisito funcional
static void CalcularPerimetroTotal() {
    Console.WriteLine($"Área total = {figuras.Sum(f => f.CalcularPerimetro())}");
}

// Requisito funcional
static void Menu() {
    Console.WriteLine("1.- Crear Figura");
    Console.WriteLine("2.- Ver colección");
    Console.WriteLine("3.- Calcular Área Total");
    Console.WriteLine("4.- Calcular Perímetro Total");
    Console.WriteLine("5.- Terminar");
}
```

```

public static void Main() {
    while (true) {
        Menu();
        int opcion = LeerOpcion();

        switch (opcion) {
            case 1: CrearFigura();break;
            case 2: VerColeccion(); break;
            case 3: CalcularAreaTotal(); break;
            case 4: CalcularPerimetroTotal(); break;
            case 5:break;
        }

        if (5 == opcion) break;
    }
}

```

EJERCICIO 2.9

```

/*
=====
Clase base: Empleado
- Nombre: propiedad automática (requerido)
- SalarioBase: propiedad NO automática con validación en el setter
- CalcularNomina: método polimórfico virtual (comportamiento por
defecto: SalarioBase)
- ToString: virtual para permitir que las clases derivadas añadan
información
Nota: La clase es abstracta para cumplir el requisito de diseño
común, pero
aporta una implementación por defecto de CalcularNomina
(comportamiento "empleado base").
=====
*/
abstract class Empleado {
    // Nombre como propiedad automática (lectura/escritura pública).
    public string Nombre { get; set; }
}

```

```

// Campo privado para el salario base. Usamos propiedad no
automática para añadir validación.
    double salarioBase;
    public double SalarioBase {
        get => salarioBase;
        set {
            // Regla: si se intenta asignar valor negativo, se pone 0.0
            salarioBase = value < 0.0 ? 0.0 : value;
        }
    }

// Constructor que inicializa propiedades comunes.
protected Empleado(string nombre, double salarioBase) {
    Nombre = nombre ?? string.Empty;
    SalarioBase = salarioBase; // el setter hará la validación
}

// Método polimórfico: por defecto, la nómina mensual es
SalarioBase.
// Es virtual para que las subclases lo sobreescriban cuando
proceda.
    public virtual double CalcularNomina() => SalarioBase;

// ToString virtual para imprimir atributos comunes; Las subclases
pueden extenderlo.
    public override string ToString() =>
        $"{GetType().Name} | Nombre: {Nombre} | SalarioBase:
{SalarioBase:0.00}";
}

/* =====
EmpleadoBase
- Representa al "Empleado Base" (no añade campos nuevos)
- Simplemente hereda el comportamiento base (CalcularNomina devuelve
SalarioBase)
- Se incluye como clase concreta para distinguir tipos en la
colección.
===== */
class EmpleadoBase : Empleado {
    public EmpleadoBase(string nombre, double salarioBase)
        : base(nombre, salarioBase) { }

    // No es necesario sobreescribir CalcularNomina; heredará
SalarioBase.
// Pero sobreescribimos ToString para dejar claro que es un

```

```

EmpleadoBase.
    public override string ToString() => base.ToString();
}

/* =====
EmpleadoFijo
- Hereda de Empleado
- BonoAnual: propiedad NO automática con validación (requisito)
- Nómina mensual = SalarioBase + BonoAnual / 12
===== */
class EmpleadoFijo : Empleado {
    double bonoAnual;
    public double BonoAnual {
        get => bonoAnual;
        set => bonoAnual = value < 0.0 ? 0.0 : value;
    }

    public EmpleadoFijo(string nombre, double salarioBase, double bonoAnual)
        : base(nombre, salarioBase) {
        BonoAnual = bonoAnual; // validación en setter
    }

    // Reusar La Lógica base cuando tenga sentido; aquí calculemos sobre
    // SalarioBase.
    public override double CalcularNomina() =>
        // prorratoe del bono anual
        SalarioBase + (BonoAnual / 12.0);

    public override string ToString() =>
        $"{base.ToString()} | BonoAnual: {BonoAnual:0.00}";
}

/* =====
EmpleadoPorHora
- Hereda de Empleado
- TarifaHora, HorasTrabajadasMes: propiedades NO automáticas con
validación
- Nómina mensual = SalarioBase + TarifaHora * HorasTrabajadasMes
===== */
class EmpleadoPorHora : Empleado {
    double tarifaHora;
    public double TarifaHora {
        get => tarifaHora;
        set => tarifaHora = value < 0.0 ? 0.0 : value;
    }
}

```

```
        double horasTrabajadasMes;
        public double HorasTrabajadasMes {
            get => horasTrabajadasMes;
            set => horasTrabajadasMes = value < 0.0 ? 0.0 : value;
        }

        public EmpleadoPorHora(string nombre, double salarioBase, double tarifaHora, double horasTrabajadasMes)
            : base(nombre, salarioBase) {
            TarifaHora = tarifaHora;
            HorasTrabajadasMes = horasTrabajadasMes;
        }

        // Reusar SalarioBase y sumar la parte variable de horas.
        public override double CalcularNomina() =>
            SalarioBase + (TarifaHora * HorasTrabajadasMes);

        public override string ToString() =>
            $"{base.ToString()} | TarifaHora: {TarifaHora:0.00} | HorasMes:
{HorasTrabajadasMes:0.00}";
    }

    /* =====
     * Programa principal (consola)
     * - Colección de Empleado (polimorfismo con LINQ para operaciones)
     * - Menú: Contratar, Ver Nóminas Individuales, Calcular Coste Total,
     * Salir
     ===== */
    class Program {
        static void Main() {
            // Lista polimórfica: guardamos instancias de las distintas
            // subclases.
            var empleados = new List<Empleado>();

            // Bucle principal del menú
            while (true) {
                Console.WriteLine();
                Console.WriteLine("==> TechSolutions - HRSystem (Consola)
==>");
                Console.WriteLine("1) Contratar Empleado");
                Console.WriteLine("2) Ver Nóminas Individuales");
                Console.WriteLine("3) Calcular Coste Total de Nóminas");
                Console.WriteLine("4) Salir");
                Console.Write("Elige una opción (1-4): ");
                var opcion = Console.ReadLine()?.Trim();
            }
        }
    }
}
```

```

        if (opcion == "1") {
            ContratarEmpleado(empleados);
        } else if (opcion == "2") {
            VerNominas(empleados);
        } else if (opcion == "3") {
            CalcularCosteTotal(empleados);
        } else if (opcion == "4") {
            Console.WriteLine("Saliendo... ¡Hasta pronto!");
            break;
        } else {
            Console.WriteLine("Opción no válida. Intenta de
nuevo.");
        }
    }

    // -----
    // ContratarEmpleado: pide tipo y datos, añade a la colección.
    // Uso intensivo de pequeños métodos auxiliares para minimizar
repetición.
    // -----
    static void ContratarEmpleado(List<Empleado> empleados) {
        Console.WriteLine();
        Console.WriteLine("Tipo de empleado a contratar:");
        Console.WriteLine("a) Empleado Base");
        Console.WriteLine("b) Empleado Fijo");
        Console.WriteLine("c) Empleado Por Hora");
        Console.Write("Elige (a-c): ");
        var tipo = Console.ReadLine()?.Trim().ToLower();

        Console.Write("Nombre: ");
        var nombre = Console.ReadLine() ?? string.Empty;

        // SalarioBase: pedimos valor y dejamos que el setter haga la
validación.
        var salarioBase = LeerDouble("Salario base mensual (ej:
1200.50): ");

        switch (tipo) {
            case "a":
                empleados.Add(new EmpleadoBase(nombre, salarioBase));
                Console.WriteLine("Empleado Base contratado
correctamente.");
                break;
        }
    }
}

```

```

        case "b":
            var bono = LeerDouble("Bono anual (se prorrataea): ");
            empleados.Add(new EmpleadoFijo(nombre, salarioBase,
bono));
            Console.WriteLine("Empleado Fijo contratado
correctamente.");
            break;

        case "c":
            var tarifa = LeerDouble("Tarifa por hora: ");
            var horas = LeerDouble("Horas trabajadas en el mes: ");
            empleados.Add(new EmpleadoPorHora(nombre, salarioBase,
tarifa, horas));
            Console.WriteLine("Empleado Por Hora contratado
correctamente.");
            break;

        default:
            Console.WriteLine("Tipo no reconocido. Operación
cancelada.");
            break;
    }
}

// -----
// VerNominas: recorre la colección y muestra ToString + nómina
mensual.

// Usamos LINQ Select para construir las líneas de salida y
ToList().ForEach para imprimir.

// -----
static void VerNominas(List<Empleado> empleados) {
    Console.WriteLine();
    if (!empleados.Any()) {
        Console.WriteLine("No hay empleados contratados.");
        return;
    }

    // Construimos una lista de strings con LINQ para separar la
lógica de presentación.
    var lineas = empleados
        .Select(e => $"{e} | Nómina mensual:
{e.CalcularNomina():0.00}")
        .ToList();

    // Imprimimos cada línea.
    lineas.ForEach(line => Console.WriteLine(line));
}

```

```

    }

    // -----
    // CalcularCosteTotal: suma todas las nóminas mensuales con LINQ
    Sum.

    // -----
    static void CalcularCosteTotal(List<Empleado> empleados) {
        Console.WriteLine();
        var total = empleados.Sum(e => e.CalcularNomina());
        Console.WriteLine($"Coste total mensual de nóminas
        ({empleados.Count} empleado(s)): {total:0.00}");
    }

    // -----
    // LeerDouble: auxiliar que solicita un valor numérico al usuario.
    // Si la entrada no es un número válido, se devuelve 0.0 (seguimos
    La filosofía preventiva).
    // Observa que si el usuario introduce un número negativo, los
    setters de las propiedades
    // convertirán ese valor a 0.0 automáticamente; aquí devolvemos el
    double tal cual.

    // -----
    static double LeerDouble(string prompt) {
        Console.Write(prompt);
        var raw = Console.ReadLine();
        if (double.TryParse(raw, out double valor)) {
            // devolvemos el valor tal cual; la validación final la
            realizará la propiedad.
            return valor;
        }
        Console.WriteLine("Entrada no numérica. Se asignará 0.0 por
        defecto.");
        return 0.0;
    }
}

```

EJERCICIO 2.10

```

-----
using System;
using System.Linq;
using System.Collections.Generic;

/*
    LogiTrack - Sistema de Envíos
    - Código compacto: reutilización, constructores en cadena, máxima
utilización de LINQ
    - Propiedades automáticas cuando es posible, validación que convierte
negativos a 0.0
    - Costo base: 2.0 € por kilogramo; La propiedad CostoBase devuelve el
coste en euros (Peso * 2.0)
*/

```

=====

Clase base: Envio

- Descripcion: propiedad automática
- Peso: propiedad no automática con validación (no negativo)
- CostoBase: propiedad de solo lectura que calcula Peso * 2.0 (euros)
- CalcularCostoTotal: virtual (por defecto, devuelve CostoBase)
- ToString: virtual para impresión de atributos comunes

*===== */*

```

abstract class Envio {
    // Descripción como propiedad automática pública.
    public string Descripcion { get; set; }

    // Campo privado para peso y propiedad con validación.
    double peso;
    public double Peso {
        get => peso;
        set => peso = value < 0.0 ? 0.0 : value; // si negativo -> 0.0
    }

    // Costo base en euros: 2.0 € por kilogramo; propiedad de solo
    // Lectura.
    // Devuelve el coste base absoluto (Peso * 2.0).
    public double CostoBase => Peso * 2.0;

    // Constructor que inicializa descripción y peso (la validación
    // ocurre en el setter).
    protected Envio(string descripcion, double peso) {
        Descripcion = descripcion ?? string.Empty;
        Peso = peso;
    }
}

```

```

// Método polimórfico: por defecto la tarifa total es el costo base.
public virtual double CalcularCostoTotal() => CostoBase;

// Representación textual básica; las subclases la extenderán.
public override string ToString() =>
    $"{GetType().Name} | Descripción: {Descripcion} | Peso(kg): 
{Peso:0.00} | CostoBase: {CostoBase:0.00}€";
}

/* =====
PaqueteEstandar
- Hereda de Envio
- TarifaPlana: propiedad no automática con validación (no negativo)
- CostoTotal = CostoBase + TarifaPlana
===== */
class PaqueteEstandar : Envio {
    double tarifaPlana;
    public double TarifaPlana {
        get => tarifaPlana;
        set => tarifaPlana = value < 0.0 ? 0.0 : value;
    }

    // Constructor: reutiliza constructor base.
    public PaqueteEstandar(string descripcion, double peso, double
tarifaPlana = 10.0)
        : base(descripcion, peso) {
        TarifaPlana = tarifaPlana;
    }

    // Reusar CostoBase y sumar la tarifa plana.
    public override double CalcularCostoTotal() => CostoBase +
TarifaPlana;

    public override string ToString() =>
        $"{base.ToString()} | TarifaPlana: {TarifaPlana:0.00}€";
}

/* =====
PaqueteExpress
- Hereda de Envio
- RecargoUrgencia: propiedad no automática con validación
(porcentaje, p.ej. 0.10 = 10%)
- CostoTotal = CostoBase + RecargoUrgencia * Peso
===== */
class PaqueteExpress : Envio {

```

```

        double recargoUrgencia;
    public double RecargoUrgencia {
        get => recargoUrgencia;
        set => recargoUrgencia = value < 0.0 ? 0.0 : value;
    }

    public PaqueteExpress(string descripcion, double peso, double
recargoUrgencia)
        : base(descripcion, peso) {
        RecargoUrgencia = recargoUrgencia;
    }

    // Calculo: costo base + recargo por urgencia multiplicado por el
peso.
    public override double CalcularCostoTotal() => CostoBase +
(RecargoUrgencia * Peso);

    public override string ToString() =>
        $"{base.ToString()} | RecargoUrgencia:
{RecargoUrgencia:0.00}€/kg";
}

/* =====
Programa principal (consola)
- Colección polimórfica List<Envio>
- Menú: Crear Envío, Ver Costos, Calcular Ingreso Total, Salir
- Uso de LINQ para Listar y sumar
===== */
class Program {
    static void Main() {
        var envios = new List<Envio>();

        while (true) {
            Console.WriteLine();
            Console.WriteLine("== LogiTrack - Sistema de Envíos ==");
            Console.WriteLine("1) Crear Envío");
            Console.WriteLine("2) Ver Costos Individuales");
            Console.WriteLine("3) Calcular Ingreso Total");
            Console.WriteLine("4) Salir");
            Console.Write("Opción (1-4): ");
            var opt = Console.ReadLine()?.Trim();

            switch (opt) {
                case "1":
                    CrearEnvio(envios);
                    break;
            }
        }
    }
}

```

```

        case "2":
            VerCostos(envios);
            break;
        case "3":
            CalcularIngresoTotal(envios);
            break;
        case "4":
            Console.WriteLine("Saliendo. ¡Hasta pronto!");
            return;
        default:
            Console.WriteLine("Opción no válida.");
            break;
    }
}

static void CrearEnvio(List<Envio> envios) {
    Console.WriteLine();
    Console.WriteLine("Tipos de paquete:");
    Console.WriteLine("a) Paquete Estándar");
    Console.WriteLine("b) Paquete Express");
    Console.Write("Elige (a-b): ");
    var tipo = Console.ReadLine()?.Trim().ToLower();

    Console.Write("Descripción: ");
    var desc = Console.ReadLine() ?? string.Empty;

    var peso = LeerDouble("Peso en kg (ej: 5.25): ");

    if (tipo == "a") {
        // Tarifa plana por defecto 10.0€, pero permitimos
personalizar.
        var tarifa = LeerDouble("Tarifa plana (por defecto 10.0): ");
        if (Math.Abs(tarifa) < 1e-9) tarifa = 10.0; // si el usuario
puso 0 tras Leer mal, mantenemos 10 por defecto
        envios.Add(new PaqueteEstandar(desc, peso, tarifa));
        Console.WriteLine("Paquete Estándar creado.");
    } else if (tipo == "b") {
        var recargo = LeerDouble("Recargo por urgencia (€/kg): ");
        envios.Add(new PaqueteExpress(desc, peso, recargo));
        Console.WriteLine("Paquete Express creado.");
    } else {
        Console.WriteLine("Tipo no reconocido. Operación
cancelada.");
    }
}

```

```

}

static void VerCostos(List<Envio> envios) {
    Console.WriteLine();
    if (!envios.Any()) {
        Console.WriteLine("No hay envíos registrados.");
        return;
    }

    // LINQ: construir líneas y mostrarlas
    envios
        .Select(e => $"{e} | CostoTotal:
{e.CalcularCostoTotal():0.00}€")
        .ToList()
        .ForEach(line => Console.WriteLine(line));
}

static void CalcularIngresoTotal(List<Envio> envios) {
    Console.WriteLine();
    var total = envios.Sum(e => e.CalcularCostoTotal());
    Console.WriteLine($"Ingreso total esperado ({envios.Count}
envío(s)): {total:0.00}€");
}

// LeerDouble: intenta parsear, si falla devuelve 0.0 (filosofía
preventiva).
static double LeerDouble(string prompt) {
    Console.Write(prompt);
    var raw = Console.ReadLine();
    if (double.TryParse(raw, out double v)) return v;
    Console.WriteLine("Entrada no numérica. Se asignará 0.0.");
    return 0.0;
}
}

```

EJERCICIO 2.11

```

/*
FleetManager - Sistema de Costes Operacionales
- Minimizar código repetido, maximizar LINQ y propiedades automáticas.
- Validación: cualquier valor numérico crítico negativo se convierte a
0.0.
- Jerarquía:
  Vehiculo (abstract)
    └─ TransportePasajeros (abstract) -> Capacidad
      └─ Autobus
    └─ TransporteCarga (abstract) -> PeajeAnual
      └─ Camion
- Interpretaciones concretas:
  * CostoOperacionalBase = 0.15 € por litro (readonly).
  * Consumo expresado en L/100km.
  * Coste combustible por km = (ConsumoLPor100km / 100) *
CostoOperacionalBase
  * Autobus: Coste/km = coste_combustible_por_km * FactorDesgaste
(1.2)
  * Camion: Coste/km = coste_combustible_por_km + (PeajeAnual /
100000.0)
  * Cálculo total de flota asume 100000.0 km por vehículo cuando se
solicita.
*/

```

```

/* =====
Clase base: Vehiculo
- Matricula: propiedad automática
- ConsumoLPor100km: propiedad con validación
- CostoOperacionalBase: readonly (0.15 €/L)
- CalcularCostoPorKm: virtual (comportamiento por defecto: coste
combustible)
- ToString: virtual
===== */
abstract class Vehiculo {
  // Matrícula: automática, pública
  public string Matricula { get; set; }

  // Campo privado para consumo; setter valida no-negativo.
  double consumoLPor100km;
  public double ConsumoLPor100km {
    get => consumoLPor100km;
    set => consumoLPor100km = value < 0.0 ? 0.0 : value;
  }
}

```

```

// Costo operativo por litro (readonly).
public double CostoOperacionalBase => 0.15;

// Constructor protegido: usado por las subclases.
protected Vehiculo(string matricula, double consumoLPor100km) {
    Matricula = matricula ?? string.Empty;
    ConsumoLPor100km = consumoLPor100km;
}

// Cálculo base: consumo L/100km -> Litros/km = Consumo/100
// coste combustible por km = Litros_por_km * CostoOperacionalBase
public virtual double CalcularCostoPorKm() =>
    (ConsumoLPor100km / 100.0) * CostoOperacionalBase;

public override string ToString() =>
    $"{GetType().Name} | Matrícula: {Matricula} | Consumo(L/100km): {ConsumoLPor100km:0.00}";
}

/* =====
TransportePasajeros (intermedia)
- Añade Capacidad (nº pasajeros) validada
- Permite compartir atributos entre vehículos de pasajeros
===== */
abstract class TransportePasajeros : Vehiculo {
    double capacidad;
    public double Capacidad {
        get => capacidad;
        set => capacidad = value < 0.0 ? 0.0 : value;
    }

    protected TransportePasajeros(string matricula, double consumoLPor100km, double capacidad)
        : base(matricula, consumoLPor100km) {
        Capacidad = capacidad;
    }

    public override string ToString() => $"{base.ToString()} | Capacidad: {Capacidad:0.##}";
}

/* =====
TransporteCarga (intermedia)
- Añade PeajeAnual validado y propiedad de solo Lectura PeajePorKm
- Permite compartir atributos entre vehículos de carga
===== */

```

```

===== */
abstract class TransporteCarga : Vehiculo {
    double peajeAnual;
    public double PeajeAnual {
        get => peajeAnual;
        set => peajeAnual = value < 0.0 ? 0.0 : value;
    }

    protected TransporteCarga(string matricula, double consumoLPor100km,
double peajeAnual)
        : base(matricula, consumoLPor100km) {
        PeajeAnual = peajeAnual;
    }

    // Peaje prorratado por km usando distancia de referencia 100000 km
    public double PeajePorKm => PeajeAnual / 100000.0;

    public override string ToString() => $"{base.ToString()} | 
PeajeAnual: {PeajeAnual:0.00}€";
}

/* =====
Autobus
- FactorDesgaste fijo 1.2
- CostePorKm = coste_combustible_por_km * FactorDesgaste
===== */
class Autobus : TransportePasajeros {
    // Factor de desgaste (readonly)
    public double FactorDesgaste => 1.2;

    public Autobus(string matricula, double consumoLPor100km, double
capacidad)
        : base(matricula, consumoLPor100km, capacidad) { }

    // Reutiliza el cálculo base y lo multiplica por el factor
    public override double CalcularCostoPorKm() =>
        base.CalcularCostoPorKm() * FactorDesgaste;

    public override string ToString() =>
        $"{base.ToString()} | FactorDesgaste: {FactorDesgaste:0.00} | 
CostePorKm: {CalcularCostoPorKm():0.000000}€";
}

/* =====
Camion
- PeajeAnual (heredado); Coste/km = coste_combustible_por_km +

```

```

PeajePorKm
=====
class Camion : TransporteCarga {
    public Camion(string matricula, double consumoLPor100km, double
peajeAnual)
        : base(matricula, consumoLPor100km, peajeAnual) { }

    public override double CalcularCostoPorKm() =>
        base.CalcularCostoPorKm() + PeajePorKm;

    public override string ToString() =>
        $"{base.ToString()} | Peaje/km: {PeajePorKm:0.000000} | 
CostePorKm: {CalcularCostoPorKm():0.000000}€";
}

/*
=====
Programa principal (consola)
- Lista polimórfica List<Vehiculo>
- Menú:
    1) Registrar Vehículo (Autobús / Camión)
    2) Ver Costos Operacionales (por km)
    3) Calcular Costo Total de Flota (asumiendo 100000.0 km/vehículo)
    4) Salir
- Uso de LINQ para listar y sumar
=====
*/
class Program {
    static void Main() {
        var flota = new List<Vehiculo>();
        const double DistanciaReferencia = 100000.0; // km por vehículo
para el cálculo agregado

        while (true) {
            Console.WriteLine();
            Console.WriteLine("==> FleetManager - Costes Operacionales
===");
            Console.WriteLine("1) Registrar Vehículo");
            Console.WriteLine("2) Ver Costos Operacionales (por km)");
            Console.WriteLine("3) Calcular Costo Total de Flota (100000
km / vehículo)");
            Console.WriteLine("4) Salir");
            Console.Write("Elige (1-4): ");
            var opcion = Console.ReadLine()?.Trim();

            switch (opcion) {
                case "1": RegistrarVehiculo(flota); break;
                case "2": VerCostos(flota); break;
            }
        }
    }
}

```

```

        case "3": {
            var total = flota.Sum(v =>
v.CalcularCostoPorKm() * DistanciaReferencia);
            Console.WriteLine($"Costo total estimado para la
flota ({flota.Count} vehículo(s) * {DistanciaReferencia:0.##} km):
{total:0.00}€");
            break;
        }
        case "4": Console.WriteLine("Saliendo. ¡Hasta pronto!");
return;
    default: Console.WriteLine("Opción no válida."); break;
}
}

// Registrar vehículo: selecciona tipo, Lee datos y añade a La
lista.
static void RegistrarVehiculo(List<Vehiculo> flota) {
    Console.WriteLine();
    Console.WriteLine("Tipos:");
    Console.WriteLine("a) Autobús (transporte de pasajeros)");
    Console.WriteLine("b) Camión (transporte de carga)");
    Console.Write("Elige (a-b): ");
    var tipo = Console.ReadLine()?.Trim().ToLower();

    Console.Write("Matrícula: ");
    var matricula = Console.ReadLine() ?? string.Empty;

    var consumo = LeerDouble("Consumo (L/100km) (ej: 25.5): ");

    if (tipo == "a") {
        var capacidad = LeerDouble("Capacidad máxima (nº pasajeros):
");
        flota.Add(new Autobus(matricula, consumo, capacidad));
        Console.WriteLine("Autobús registrado correctamente.");
    } else if (tipo == "b") {
        var peaje = LeerDouble("Peaje anual (euros): ");
        flota.Add(new Camion(matricula, consumo, peaje));
        Console.WriteLine("Camión registrado correctamente.");
    } else {
        Console.WriteLine("Tipo no reconocido. Operación
cancelada.");
    }
}

// VerCostos: muestra ToString() de cada vehículo (que incluye el

```

```
coste por km).
    static void VerCostos(List<Vehiculo> flota) {
        Console.WriteLine();
        if (!flota.Any()) {
            Console.WriteLine("No hay vehículos registrados.");
            return;
        }

// Usamos LINQ para componer las líneas; ToList().ForEach para imprimir.
        flota
            .Select(v => $"{v}") // v.ToString() ya contiene información relevante y el coste por km
            .ToList()
            .ForEach(Console.WriteLine);
    }

// Auxiliar para leer doubles; si la entrada no es válida devuelve 0.0
// (la validación final de negativos ocurre en los setters de propiedades).
    static double LeerDouble(string prompt) {
        Console.Write(prompt);
        var raw = Console.ReadLine();
        if (double.TryParse(raw, out double valor)) return valor;
        Console.WriteLine("Entrada no numérica. Se asignará 0.0.");
        return 0.0;
    }
}
```