

Ejercicios Polimorfismo y Propiedades

[Descargar estos ejercicios](#)

Índice

- ☒ Ejercicio 1
- [Ejercicio 2](#)
- ☒ Ejercicio 3
- [Ejercicio 4](#)
- ☒ Ejercicio 5

✓ Ejercicio 1

Para practicar el concepto de **polimorfismo funcional o sobrecarga**, vamos a suponer que tenemos un método de utilidad que nos permite calcular el coste de la carrera de un taxi.

Hay cuatro conceptos básicos que conforman el precio de la carrera. Dos de ellos fijos, que son la bajada de bandera y la carrera mínima, y dos conceptos variables en cada trayecto: los kilómetros recorridos y el tiempo de espera.

Además de los cuatro conceptos básicos, existen algunos recargos o suplementos que deben pagarse en determinadas circunstancias: por día festivo o domingo, por horario nocturno, por mascotas u otros conceptos de ocupación extra.

A partir del método `CosteCarrera` con **parámetros opcionales** que se muestra a continuación, refactoriza el código para quitar los parámetros opcionales de métodos públicos sobrecargando `CosteCarrera` y que se ofrezcan sobrecargas con el menor número de parámetros posibles.

🔴 **Nota:** Modificaremos el `Main` para evitar llamadas donde se pasen los valores a `0` o a `false` en las llamadas a `CosteCarrera`, al realizar estos cambios deberás modificar el tipo de `ocupacionExtra` a `uint`, para que no se produzca ambigüedad en la llamada.

```
public static class Taxi
{
    const float BAJADA_BANDERA = 1.82F;
    const float CARRERA_MINIMA = 3.63F;
    const float COSTE_KM = 0.9F;
    const float ESPERA_POR_HORA = 18.77F;
    const short PORCENTAJE_NOCTURNO = 30;

    public static double CosteCarrera(
        float kilometrosRecorridos, float minutosEspera,
        bool nocturno = false, int porcentajeFestivo = 0, int ocupacionExtra = 0)
    {
        float costeCarrera = BAJADA_BANDERA + kilometrosRecorridos * COSTE_KM
            + minutosEspera * (ESPERA_POR_HORA / 60);
        costeCarrera = costeCarrera < CARRERA_MINIMA ? CARRERA_MINIMA : costeCarrera;
        float incrementoNocturno = nocturno ? costeCarrera / PORCENTAJE_NOCTURNO : 0;
        float incrementoFestivo = porcentajeFestivo != 0 ? costeCarrera * porcentajeFestivo / 100f : 0;
        costeCarrera += incrementoFestivo >= incrementoNocturno ? incrementoFestivo : incrementoNocturno;
        costeCarrera += ocupacionExtra;
        return costeCarrera;
    }
}
```

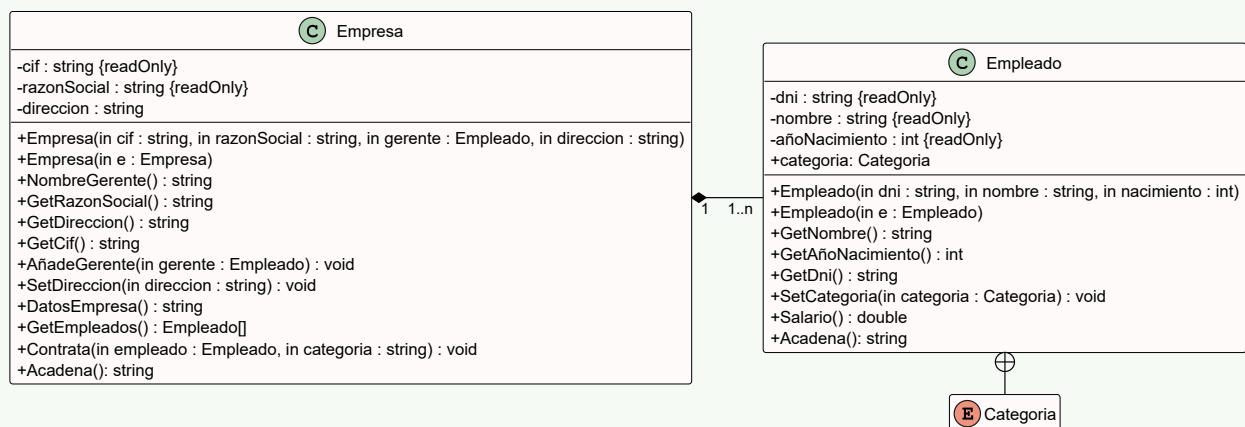
```

class Program
{
    static void Main()
    {
        Console.WriteLine($"Coste carrera lunes mañana -> {Taxi.CosteCarrera(20, 5):f2}");
        Console.WriteLine($"Coste carrera lunes noche -> {Taxi.CosteCarrera(20, 5, true):f2}");
        Console.WriteLine($"Coste carrera lunes con mi mascota Dogo -> "
            + $"{Taxi.CosteCarrera(20, 5, false, 0, 1):f2}");
        Console.WriteLine($"Coste carrera Domingo de Ramos -> "
            + $"{Taxi.CosteCarrera(20, 5, false, 40):f2}");
        Console.WriteLine($"Coste carrera Domingo noche -> "
            + $"{Taxi.CosteCarrera(20, 5, true, 20):f2}");
        Console.WriteLine($"Coste carrera Domingo de Ramos noche con Dogo y Minina -> "
            + $"{Taxi.CosteCarrera(20, 5, true, 40, 2):f2}");
    }
}

```

Ejercicio 2

A partir del **ejercicio 6 de objetos básicos (Bloque 4)**, vas a sustituir los métodos accesoros y mutadores por las propiedades propias de C#. Adjuntamos el UML del ejercicio anterior para que puedas guiarte.



✓ Ejercicio 3

Vamos a ampliar el ejercicio de la cuenta corriente que hicimos en el bloque anterior de excepciones, modificando ligeramente su definición inicial y añadiendo los TAD necesarios para modelar **los distintos tipos de cuenta de un banco algo usurero**. e tal manera que:

- La **clase Cuenta**, importada del bloque anterior, va a simbolizar un tipo de cuenta genérico del que sólo podremos ingresar, retirar fondos y tener unos datos de usuario.
- Además, añadiremos una **propiedad autodefinida Saldo** donde el get será público y el set protegido.
- Por otro lado se modificará el método **Reintegro** para que devuelva la cantidad reintegrada, que en ocasiones no será la misma que el usuario desea reintegrar.

👉 **Importante:** Para poder construir correctamente el código y no perderos durante la lectura del texto, os aconsejamos concienzudamente, que miréis las llamadas a los métodos del programa principal.

Las distintas especificaciones para cuenta serán:

Cuenta de ahorro

- Se especializa respecto a la genérica mediante la aplicación de intereses al saldo actual. or ejemplo, si una cuenta tiene un saldo de **1000€** y la tasa de interés es del **2%**, después del pago de intereses el saldo será de **1020€**.
- En este tipo de cuenta los intereses se sumarán **al finalizar el mes** sobre el Saldo en ese momento.

```
saldo = saldo + (saldo * interes_tpu);
```

- Redefiniremos el método ToString para que indique el tipo de cuenta antes de llamar al de la clase base para que muestre sus datos básicos. La salida correspondiente al ToString es la siguiente:

```
....en Cuenta Ahorro
Numero de cuenta: 2085-0103-92-0300731702
Titular: Nicolas
Saldo: 0,00 €.
....
```

Cuenta de depósito

- También aplica intereses al saldo actual, por lo cual hereda de cuenta de ahorro. Pero **si el titular realiza un reintegro de parte del capital antes del vencimiento del plazo fijo**, el banco deducirá un porcentaje sobre el reintegro.
Por ejemplo, si el titular retira **1000€** antes del vencimiento del plazo y hay un recargo del **5% sobre el reintegro**, el saldo disminuirá en **1000€** pero el titular sólo recibirá **950€**. Por tanto el Reintegro devolverá la cantidad reducida.

```

this.Saldo = this.Saldo - cantidad;
// Si el depósito aún no venció.
cantidad = cantidad - (cantidad * recargo_tpu);

```

- Si el plazo de la cuenta venció, no se cobrará recargo por el reintegro.
- Este tipo de cuenta no permite crédito y los intereses se abonarán al finalizar el mes sobre el saldo en ese momento, igual que en la clase padre.
- Anularemos el método **ToString** para que indique el tipo de cuenta antes de llamar al de la clase base para que muestre sus datos básicos como en la salida mostrada más abajo.

👉 **Importante:** Para poder probar la clase, el vencimiento se controlará desde fuera de la clase y en esta quedará reflejado mediante un valor booleano.

Cuenta corriente

- No aplica intereses al saldo, pero permite al titular girar cheques y realizar reintegros a través del cajero automático. No obstante, el banco restringe el número de transacciones mensuales a una determinada cantidad y si el titular excede las transacciones; el banco le cobrará un recargo por transacción. Se consideran como transacciones los reintegros y los ingresos.
Por ejemplo si tenemos **5 transacciones** y realizamos **8** y el recargo por transacción adicional es de **1€**, el banco nos cobrará **3€**.
- Este recargo por pasar del número de transacciones mensuales establecido, se realizara al finalizar el mes. Momento en el cual se aplicará el recargo sobre el Saldo y además se reiniciará el número de transacciones a cero.

```

Saldo = Saldo
    - (numeroTrasaccionesMesActual - maximoTrasaccionesGratisPorMes)
    * recargoXTransaccionAdicional_Euros;

```

- Redefiniremos el método ToString para que indique el tipo de cuenta antes de llamar al de la clase base para que muestre sus datos básicos.

Cuenta crédito

- Permite al titular retirar dinero adicional al que indica su saldo, hasta un límite de crédito. Pero esto no es gratuito; al finalizar el mes, el banco aplicará una tasa de interés al saldo negativo o descubierto en ese momento.
Por ejemplo, si nuestro saldo es **-1000€ al 20%**, pagaremos un recargo de **200€**, de tal manera que el nuevo saldo será de **-1200€**.

```

double cargo = (Saldo < 0) ? Math.Abs(Saldo) * interes_tpu : 0d;
Saldo = Saldo - cargo;

```

- A diferencia de la cuenta corriente, la cuenta de crédito no tiene límite para la cantidad de transacciones que se pueden realizar sobre ella. Pues al banco le interesa que nos quedemos sin dinero para poder *"chuparnos la sangre"*.
- En todos los casos el control de tiempo se realizará de forma externa a las clases y los intereses, recargos, etc... se gestionarán a través de operaciones sobre los diferentes objetos de cuenta.

- Definiremos una excepción personalizada `CreditoMaximoExcedidoException`, como clase anidada pública. **Esta excepción se generará si el usuario al intentar realizar un reintegro que supere el límite de crédito establecido para la cuenta, teniendo en cuenta el Saldo**, se le pasarán las tres cantidades al constructor y se construirá la cadena indicando el problema en la propia llamada a base.
- Anularemos el método `ToString` para que indique el tipo de cuenta antes de llamar al de la clase base para que muestre sus datos básicos.

Programa principal

Deberemos diseñar la jerarquía de clases que creamos más conveniente y probar el siguiente código...

```
class Ejercicio3
{
    static CuentaAhorro ca = new CuentaAhorro("2085 0103 92 0300731702", "Nicolas", .02d);
    static CuentaDeposito cd = new CuentaDeposito("2100 1162 43 0200084482", "Juan", .06d, .05d);
    static CuentaCorriente cc = new CuentaCorriente("2100 0721 09 0200601249", "Jhon", 2, 3d);
    static CuentaCredito cr = new CuentaCredito("0049 0345 31 2710611698", "Jose", .18d, 2000);

    static void SaldoActual(Cuenta cuenta)
    {
        Console.WriteLine($"Saldo tras operación {cuenta.Saldo:C}\n");
    }

    static void Ingresa(Cuenta cuenta, double[] cantidades)
    {
        foreach (double cantidad in cantidades)
        {
            Console.WriteLine($"Ingresando {cantidad:C} en {cuenta}.");
            cuenta.Ingreso(cantidad);
            SaldoActual(cuenta);
        }
    }
}
```

```

static void Retira(Cuenta cuenta, double[] cantidades)
{
    foreach (double cantidad in cantidades)
    {
        try
        {
            Console.WriteLine($"Retirando {cantidad:C} en {cuenta}.");
            Console.WriteLine($"La cantidad recibida es: {cuenta.Reintegro(cantidad)}");
        }
        catch (Cuenta.SaldoInsuficException e)
        {
            Console.WriteLine(e.Message);
        }
        catch (CuentaCredito.CreditoMaximoExcedidoException e)
        {
            Console.WriteLine(e.Message);
        }
        SaldoActual(cuenta);
    }
}

```

```

static void FinalizaMes()
{
    Console.WriteLine("Finalizando mes actual ...");
    Console.WriteLine($"Aplicando intereses en {ca}...");
    ca.SumaInteres();
    SaldoActual(ca);
    Console.WriteLine($"Aplicando intereses en {cd}...");
    cd.SumaInteres();
    SaldoActual(cd);
    Console.WriteLine($"Aplicando recargos en {cc}...");
    cc.AplicaRecargosMes();
    SaldoActual(cc);
    Console.WriteLine($"Revisando cargo sobre saldo negativo en {cr}...");
    double cargo = cr.CargaInteresesMes();
    if (cargo > 0d)
        Console.WriteLine($"Has tenido un cargo de {cargo:C} por saldo negativo.");
    SaldoActual(cr);
}

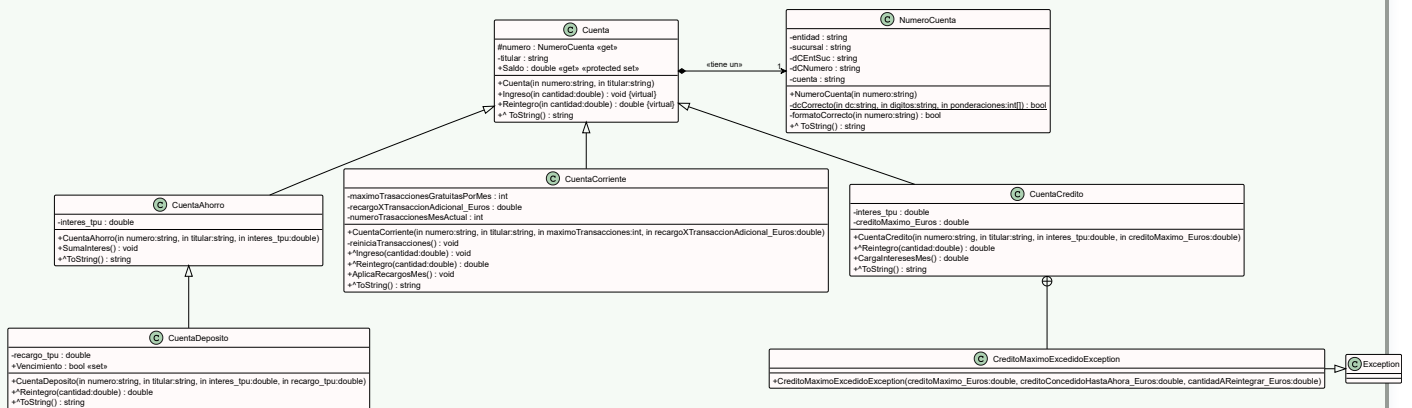
```

```

static void Main()
{
    Ingresa(ca, new double[] { 1000, 1000 });
    Ingresa(cd, new double[] { 10000 });
    Retira(ca, new double[] { 10000000, 500 });
    Ingresa(cc, new double[] { 1000, 2000 });
    Ingresa(cc, new double[] { 2000 });
    Retira(cc, new double[] { 500, 10000000, 500 });
    Ingresa(cr, new double[] { 1000 });
    Retira(cr, new double[] { 500 });
    Retira(cd, new double[] { 10000000, 1000 });
    FinalizaMes();
    cd.Vencimiento = true;
    Ingresa(ca, new double[] { 1000, 1000 });
    Retira(ca, new double[] { 500 });
    Ingresa(cc, new double[] { 1000 });
    Retira(cc, new double[] { 100, 200, 100, 100, 100 });
    Retira(cr, new double[] { 1000 });
    Retira(cd, new double[] { 1000 });
    FinalizaMes();
    Retira(cr, new double[] { 11000 });
}
}

```

🚩 **Nota:** La siguiente imagen es un **UML orientativo** para las clases y métodos propuestos en el ejercicio. Solo úsalo si tienes dudas con el enunciado pues posiblemente en un examen no dispongas de él y debes saber si tu interpretación del enunciado te da un resultado parecido.



El resultado por consola de ejecutar el programa será parecido al siguiente...

```

Ingresando 1000 en Cuenta Ahorro
Numero de cuenta: 2085-0103-92-0300731702
Titular: Nicolas
Saldo: 2530 euros.
Saldo tras operación 3530

```


Retirando 500 en Cuenta Ahorro
Numero de cuenta: 2085-0103-92-0300731702
Titular: Nicolas
Saldo: 3530 euros.
La cantidad recibida es: 500
Saldo tras operación 3030

Ingresando 1000 en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 3988 euros.
Saldo tras operación 4988

Retirando 100 en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 4988 euros.
La cantidad recibida es: 100
Saldo tras operación 4888

Retirando 200 en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 4888 euros.
La cantidad recibida es: 200
Saldo tras operación 4688

Retirando 100 en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 4688 euros.
La cantidad recibida es: 100
Saldo tras operación 4588

Retirando 100 en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 4588 euros.
La cantidad recibida es: 100
Saldo tras operación 4488

Retirando 100 en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 4488 euros.
La cantidad recibida es: 100
Saldo tras operación 4388

Retirando 1000 en Cuenta Crédito
Numero de cuenta: 0049-0345-31-2710611698
Titular: Jose
Saldo: 500 euros.
La cantidad recibida es: 1000
Saldo tras operación -500

Retirando 1000 en Cuenta Depósito
Cuenta Ahorro
Numero de cuenta: 2100-1162-43-0200084482
Titular: Juan
Saldo: 9540 euros.
La cantidad recibida es: 1000
Saldo tras operación 8540

Finalizando mes actual ...
Aplicando intereses en Cuenta Ahorro
Numero de cuenta: 2085-0103-92-0300731702
Titular: Nicolas
Saldo: 3030 euros...
Saldo tras operación 3090,6

Aplicando intereses en Cuenta Depósito
Cuenta Ahorro
Numero de cuenta: 2100-1162-43-0200084482
Titular: Juan
Saldo: 8540 euros...
Saldo tras operación 9052,4

Aplicando recargos en Cuenta Corriente
Numero de cuenta: 2100-0721-09-0200601249
Titular: Jhon
Saldo: 4388 euros...
Saldo tras operación 4376

Revisando **cargo** sobre saldo negativo en Cuenta Crédito

Numero de cuenta: 0049-0345-31-2710611698

Titular: Jose

Saldo: -500 euros...

Has tenido un **cargo** de **90** por saldo negativo.

Saldo tras operación -590

Retirando **11000** en Cuenta Crédito

Numero de cuenta: 0049-0345-31-2710611698

Titular: Jose

Saldo: -590 euros.

No puedes retirar **11.000,00** ?.

Tu crédito máximo es de **2.000,00** ? del cual se te ha concedido ya **590,00** ?

Saldo tras operación -590

Ejercicio 4

Crea una clase denominada **Alarma** cuyos objetos activen un objeto de tipo **Timbre** cuando el valor medido por un **Sensor** supere un umbral preestablecido.

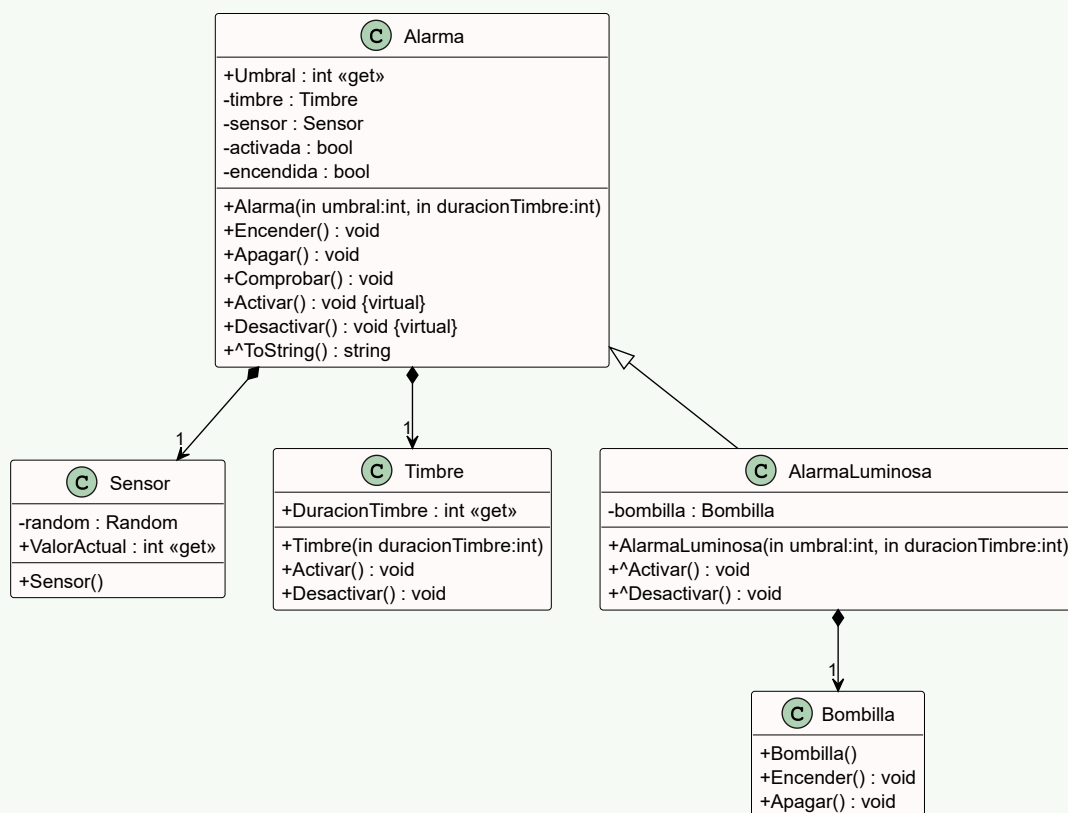
Implementa en C# todo el código necesario para el funcionamiento de la alarma, la alarma comprobará si debe activar o desactivar el timbre mediante el método **Comprobar()**. Al Encender la alarma, se pasará (mediante un bucle) a llamar a comprobar hasta que decidamos apagar la alarma.

Crea una **subclase de Alarma** denominada **AlarmaLuminosa** que, además de activar el timbre, encienda una luz (que representaremos con un objeto de tipo **Bombilla**).

💡 **Pista:** Los métodos **Activar/Desactivar** se encargarán de hacer lo propio con el timbre y en su caso con la señal luminosa. Mientras que **Encender/Apagar** enciende la alarma para que comience con la comprobación sensor-umbral. La propiedad **ValorActual** del sensor, devolverá un número aleatorio.

Como al **Encender** la alarma, lo mejor es crear un bucle que haga la comprobación, podemos acabar el bucle al pulsar una tecla, para ello puedes usar **Console.KeyAvailable**

El siguiente UML te puede ayudar a entender lo que se pide.



💡 **Nota:** Procura eliminar la aparición de código duplicado al crear la subclase de **Alarma** y asegúrate de que, cuando se activa la alarma luminosa se enciende la luz de alarma y también suena la señal sonora asociada al timbre (puedes usar mensajes o investigar para obtener un mejor resultado).

✓ Ejercicio 5

Vamos a desarrollar un sistema de gestión de ingredientes y recetas para un restaurante, para practicar profundizar en las propiedades. Para ello vamos a crear las siguientes clases:

- **Clase Ingrediente** que representa un ingrediente de cocina con su nombre, precio y cantidad en stock.
 - clase IngredienteException para manejar errores en la entrada de datos.
 - Nombre - propiedad de tipo string, pública de acceso y privada de modificación y que controle que el nombre no puede ser vacío con un IngredienteException.
 - Precio - propiedad de tipo decimal, pública de acceso y privada de modificación y que controle precio no puede ser negativo con un IngredienteException.
 - Stock - propiedad entera autoimplementada y pública de acceso y mutación.
 - EnStock - Propiedad de tipo boolean pública y calculada que indicará si hay ingredientes en stock.
 - ToString para que se muestre como en la salida
- **Clase LineaIngrediente** representa la cantidad de un ingrediente que se usa dentro de una receta. Es decir relaciona un Ingrediente con una cantidad específica.
 - Producto - propiedad autoimplementada y pública de acceso y mutación que representa el ingrediente asociado a la línea.
 - Cantidad - propiedad autoimplementada de tipo entero, pública de acceso y privada de mutación.
 - Subtotal - propiedad pública y calculada que devuelve el costo total de esa línea de ingrediente.
 - Constructor que crea una LineaIngrediente a partir del producto y la cantidad usada en la receta.
 - ToString para que se muestre como en la salida.
- **Clase Receta** representa una receta culinaria compuesta por varios ingredientes.
 - Ingredientes - que representa la relación de composición entre la receta y la línea de ingredientes de esta. Avitualmente las recetas pueden tener más de un ingrediente.
 - Nombre - propiedad pública de solo lectura y autoimplementada que almacena el nombre de la receta.
 - CosteTotal - propiedad calculada y pública que suma los subtotales de todas las líneas de ingredientes para obtener el costo total de la receta.
 - AñadeIngrediente(Ingrediente producto, int cantidad) método que agrega un ingrediente a la receta. Si el ingrediente no está en stock, lanza una excepción. Si hay suficiente stock, lo descuenta de este y lo añade a la receta.
 - ToString que devuelve una representación de la receta, listando sus ingredientes y el costo total.

👉 **Importante:** Se pasa la salida y el código de la Main, para que os ayude a deducir el formato de las salidas y otros elementos que no hayan quedado claros en el texto.

```

class Program
{
    static void Main()
    {
        Console.WriteLine("Comprando ingredientes para el restaurante:");
        Ingrediente tomate = new Ingrediente("Tomate", 1.5m, 10);
        Ingrediente cebolla = new Ingrediente("Cebolla", 0.8m, 5);
        Ingrediente pimienta = new Ingrediente("Pimiento", 2m, 3);
        Ingrediente carnePicada = new Ingrediente("Carne picada", 3.5m, 2);
        Ingrediente huevo = new Ingrediente("Huevo", 0.2m, 30);

        Console.WriteLine("Creando la receta de pinto:");
        Receta pinto = new Receta("Pinto");
        pinto.AñadeIngrediente(tomate, 3);
        pinto.AñadeIngrediente(cebolla, 2);
        pinto.AñadeIngrediente(pimiento, 1);
        Console.WriteLine(pinto);

        Console.WriteLine("Mostrando stock de tomate");
        Console.WriteLine(tomate);
        Console.ReadLine();
    }
}

```

Comprando ingredientes para el restaurante:

Creando la receta de pinto:

Receta: Pinto

Tomate x 3 = 4,5 €

Cebolla x 2 = 1,6 €

Pimiento x 1 = 2 €

Coste total: 8,1 €

Mostrando stock de tomate

Producto: Tomate

Precio: 1,5 €

Stock: 7 unidades

Disponible: Sí