Índice

Ejercicios Unidad 18 - Genéricos y extensores

Descargar estos ejercicios

> **Antes de empezar**
>
> Para realizar estos ejercicios, deberás descargar los recursos del enlace de proyecto_poo. Como puedes ver, la solución está compuesta de varios proyectos. Cada uno de ellos corresponde con un ejercicio, deberás implementar todo el código, tanto de la Main como de los métodos que se piden en cada ejercicio. Cada proyecto contiene el test correspondiente, que deberás pasar para comprobar que has hecho el ejercicio correctamente.

Ejercicio 1. Contenedor genérico de lecturas

Para practicar un primer uso de tipos genéricos vas a crear una clase que almacene lecturas de cualquier tipo. Y que proporcionará métodos para facilitar el trabajo con ellas.

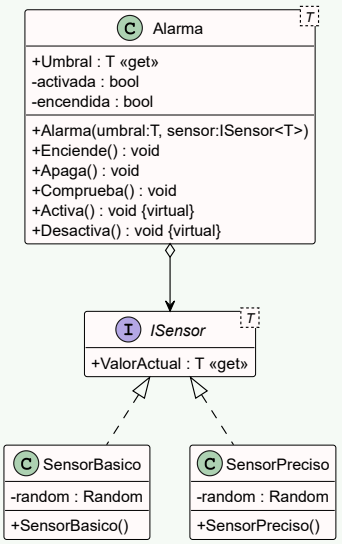
```
Ejercicio 1: Contenedor genérico de lecturas

Creando contenedor de temperaturas (double)...
Agregando: 21.5
Agregando: 22.1
Agregando: 20.9
Lecturas (3): 21,5 - 22,1 - 20,9
Última: 20,9
Creando segundo contenedor de temperaturas (double)...
Lecturas (4): 23 - 24,5 - 22,1 - 22,8
Agregando rango del segundo contenedor al primero...
Lecturas (6): 21,5 - 22,1 - 20,9 - 23 - 24,5 - 22,8
Limpiando...
Lecturas (0):


Creando contenedor de códigos (string)...
Error: No hay lecturas
Agregando: A1
Agregando: B2
Agregando: C3
Lecturas (3): A1 - B2 - C3
Pulsa una tecla para continuar...
```

Requisitos

Para ello habrá que implementa una clase genérica `ContenedorLecturas<T>` con un campo privado `List<T> contenedor` que contenga:

- Propiedad `Ultima` de solo lectura que devuelve la última lectura o lanza una excepción de tipo `ContenedorException` con el mensaje apropiado si todavía no hay lecturas.
- Propiedad `Primera` como la anterior pero que devuelva la primera lectura leída.
- Propiedad `Conteo` que devuelve cuántas lecturas hay.
- Método `Agrega` que añade una nueva lectura al contenedor, pero solo en el caso de que esa lectura no se hubiese añadido antes (no permite contener lecturas repetidas)
- Método `LecturaIndice` al que le llegue un entero de posición y devuelva la lectura correcpondiente a esa posición o lanza una excepción de tipo `ContenedorException` con el mensaje apropiado si no existe esa lectura.
- Anula el `ToString` para que devuelva una cadena formateada como: `Lecturas (N): v1, v2, v3`. Donde N es la cantidad de lecturas recogidas.
- Método `Limpia` que elimina todas las lecturas.
- Método `AgregaRango` al que le llega un objeto de tipo `ContenedorLecturas` y las añadirá al contenedor. Solo las que no sean repetidas.
- En el programa principal crea un método `GestionContenedor` para crear los objetos y realizar la llamada a los métodos para conseguir la salida como se muestra.

Ejercicio 2. Interfaz genérica para implementar parte de una alarma




```
Ejercicio 2: Interfaz Genérica Alarma

Configurando alarma genérica (umbral = 70)...
Encendiendo alarma...
Valor leído: 63  -> debajo del umbral
Valor leído: 62  -> debajo del umbral
Valor leído: 82  -> supera umbral => [ROJO] ACTIVA [/ROJO]
Valor leído: 72  -> sigue sobre umbral (ya ACTIVA)
Valor leído: 41  -> baja del umbral => [VERDE] DESACTIVA [/VERDE]
Valor leído: 100  -> supera umbral => [ROJO] ACTIVA [/ROJO]
Valor leído: 2  -> debajo del umbral
¿Desea apagar la alarma? (S/N): n
Continuando comprobaciones...
Valor leído: 26  -> debajo del umbral
Valor leído: 38  -> debajo del umbral
Valor leído: 93  -> supera umbral => [ROJO] ACTIVA [/ROJO]
Valor leído: 27  -> baja del umbral => [VERDE] DESACTIVA [/VERDE]
¿Desea apagar la alarma? (S/N): s

Configurando alarma precisa (umbral = 70)...
Encendiendo alarma precisa...
Valor leído: 36.65  -> debajo del umbral
Valor leído: 44.32  -> debajo del umbral
Valor leído: 44.12  -> debajo del umbral
Valor leído: 98.14  -> supera umbral => [ROJO] ACTIVA [/ROJO]
Valor leído: 72.11  -> sigue sobre umbral (ya ACTIVA)
¿Desea apagar la alarma? (S/N): s
Fin de la aplicación.
```

Requisitos

Implementa las siguientes necesidades:

- Clase `SensorBasico` que representa el sensor más básico y barato del mercado, que genera valores enteros aleatorios (por ejemplo 0-100).
- Clase `SensorPreciso` genera valores con decimales (por ejemplo 0-100 con dos cifras decimales).
- Clase genérica `Alarma<T>` con algunos de los siguientes requerimientos, además de los que se pueden deducir del diagrama.
  - `Enciende` y `Apaga` métodos que gestionan el estado `ON/OFF` de la Alarma, a través de la variable `encendida` y `activada`.
  - `Activa` cambia `activada` a true y escribe en consola un texto en color Rojo que muestre `ACTIVA` (en la demo textual, simulado con `[ROJO] ACTIVA [/ROJO]`).
  - `Desactiva` la inversa a la anterior con texto en verde.
  - Método `Comprueba` Se ejecuta tras encender la alarma. Entra en un bucle mientras `encendida` sea true. Cada iteración obtiene `ValorActual` del sensor y lo compara con el `Umbral` (usa `IComparable<T>`). Si el valor supera el umbral y no está activada, llama a `Activa()`. Si el valor baja del umbral y está activada, llama a `Desactiva`. Debe hacer pequeñas pausas (`Thread.Sleep`) para no saturar la

CPU. Si se detecta una tecla pulsada (`Console.KeyAvailable`), se pregunta: "¿Desea apagar la alarma? (S/N):". Si el usuario responde S (s o S), se llama `Apaga()` y se sale del bucle; en caso contrario el bucle continúa. Muestra cada lectura como se ve en la salida.

- Método private static string FormateaValor(T valor) con el siguiente código:

```
private static string FormateaValor(T valor)
{
    if (valor is float f)
    {
        return f.ToString("F2", CultureInfo.InvariantCulture);
    }
    if (valor is double d)
    {
        return d.ToString("F2", CultureInfo.InvariantCulture);
    }
    return valor?.ToString() ?? string.Empty;
}
```

- Implementa un método en la clase principal `GestionAlarma` de prueba que cree una alarma con `SensorBasico` (umbral configurable) y demuestre el flujo de: encender, comprobar, interactuar, continuar o apagar. Añade otra instancia usando `SensorPreciso` para mostrar lecturas con decimales.

Ejercicio 3. Métodos genéricos y restricciones de tipo

Este ejercicio nos va a servir para asimilar el concepto de las restricciones de tipo y para practicar los métodos genéricos.

```
Ejercicio 3. Métodos genéricos y restricciones de tipo

Probando los métodos genéricos con tipos básicos:
        Comparando si la cadena `Hola` es mayor que `Hola`: False
        Comparando si el carácter 'c' es mayor que 'b':True
        Comparando si el número 4 es mayor que 3: True

Probando los métodos genéricos con el tipo Persona:
        Mostrando las personas:
Nombre: James Raynor, Edad: 35 años.
Nombre: Sarah Kerrigan, Edad: 32 años.
Nombre: Sarah Kerrigan, Edad: 32 años.
        Comparando si Nombre: James Raynor, Edad: 35 años. es mayor
        que Nombre: Sarah Kerrigan, Edad: 32 años.: False
        Comparando si Nombre: Sarah Kerrigan, Edad: 32 años. es mayor
        que Nombre: James Raynor, Edad: 35 años.: True
        Comparando si Nombre: Sarah Kerrigan, Edad: 32 años. es menor
        que Nombre: James Raynor, Edad: 35 años.:False

Fin de la aplicación.
```

Requisitos

- Crea una clase estática llamada `Comparador` que posea a su vez dos métodos de utilidad estáticos llamados `Mayor` y `Menor` . Ambos recibirán dos parámetros del tipo genérico, y devolverán true o false en el caso de que el primer parámetro sea mayor que el segundo y viceversa.

> ⚠ **Cuando intentes comparar los parámetros de entrada de los métodos, no podrás usar los operadores lógicos. Esto es debido a que sobre los tipos genéricos solo se pueden realizar operaciones genéricas. La mejor forma de solucionarlo, es obligando a que el parámetro genérico del método implemente la interface `IComparable<T>` .**

- Una vez solucionado el problema, crea en la clase Program el código necesario para probar estos métodos, usando diferentes tipos `int` , `string` , `float` , `char` .
- Ahora crea una clase `Persona` que tenga solo dos propiedades: `Nombre` y `Edad` y que invalide el `ToString` .
- Comprueba si funcionan los métodos `Mayor` y `Menor` con el tipo Persona.

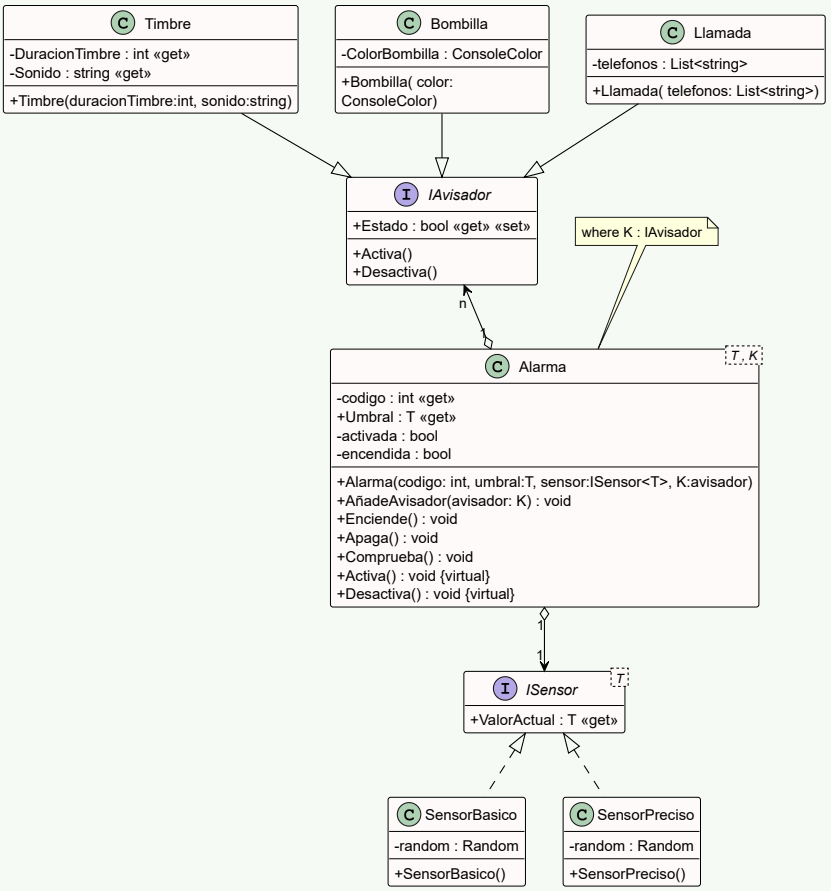
> ⚠ **No es posible usar los métodos genéricos `Mayor` y `Menor` con los tipos que no implementen IComparable (ya que se le ha puesto la restricción de tipo), por eso no funciona con Persona. Los tipos básicos con los que habías probado los métodos con anterioridad, tienen implementado en su código `IComparable<T>`.**

- Ahora haz que la clase derive de `IComparable<Persona>` y crea el código necesario par que funcione correctamente.
- Prueba ahora en el programa principal el método estatico `Mayor` para que con dos objetos Persona distintos, te diga cual es el mayor.
- Haz que Persona herede de `ICloneable` he implementa el código.
- Clona una persona y prueba los clones con el método estático `Menor`.

Ejercicio 4. Alarma genérica con sensor y avisadores (ampliación)

Extensión del Ejercicio 2, ahora la alarma admite varios avisadores intercambiables. Además del sensor genérico que supervisa el umbral, podremos agregar diferentes implementaciones de `IAvisador` (timbre, bombilla, llamadas, etc.) para reaccionar cuando el valor supere o vuelva a quedar por debajo del umbral.

```
Ejercicio 4. Alarma genérica con sensores y avisadores

Creando alarmas...
Lectura: 0  => OK
... Timbre detenido ...
   .-.
  (   )
   '-'
... Luz apagada ...
Lectura: 85  => OK ACTIVANDO
TIMBRE
... Ring.... ...
LUZ
   .-.
  (   )
   '-'
  // \\
... Luz encendida ...
LLAMADA
Llamando al +34123456789. Alarma ACTIVADA a 27/02/2026 17:48:13
Llamando al +34123456780. Alarma ACTIVADA a 27/02/2026 17:48:13
Lectura: 18  => OK
... Timbre detenido ...
   .-.
  (   )
   '-'
... Luz apagada ...
Fin de llamadas de alarma.
Lectura: 30  => OK

¿Desea apagar la alarma? (S/N):

Apagando alarmas.
Fin de la demo.

Presiona cualquier tecla para salir...
```



Requisitos

Completa el diseño mostrado en el diagrama aplicando estas pautas:

- La alarma tendrá siempre al menos un avisador (inyectado en el constructor) y permitirá añadir otros mediante `AñadeAvisador`. Rechaza valores nulos y evita duplicar la misma instancia.
- Los métodos `Activa` y `Desactiva` de alarma activarán y desactivarán todos los avisadores respectivamente (solo se desactivarán los que estén activos).
- Método privado y estatico `EtiquetaAvisador` al que le llega un IAvisador y devolverá una string indicando si es de tipo `Timbre / Luz / LLamada o Desconocida`, este método se usará en Activa para mostrar que activador está funcionando en cada momento.
- Modificaremos el método Comprueba para que:
  - Al pasar de estado “no activada” a “activada” (supera umbral) se llama a `Activa`. Mientras que el umbral este superado, se llamará en cada pasada del bucle a Activa, para simular la realidad (usando un retardo). Al volver a quedar por debajo del umbral se llama a `Desactiva`.
- `Avisador Timbre` al activarse se activa el sonido (simulado mediante el mensaje Sonido) que irá apareciendo con pequeños retardos (ya controlado en el método Comprueba). El mensaje dejará de aparecer cuando el tiempo en segundos haya transcurrido o se haya bajado del umbral (usa DateTime para controlar que el tiempo con el timbre sonando ha superado la duración en segundos de este).
- `Avisador Bombilla` muestra un mensaje coloreado (Color elegido encendida / Gris apagada).
  - Método privado `RepresentacionBombilla` que devuelva una cadena con una representación de una bombilla realizada mediante caracteres.
  - Métodos **extensores de string**: `Color`, `Gris` que cambie el color del texto a gris o del color que se le pase por parámetro al método. Usa estos métodos en RepreentacionBombilla para simular el encendido y apagado.
- `Avisador Llamada` mantiene una lista de teléfonos. Al activarse muestra, para cada número, un mensaje del tipo: “Llamando al [número]. Alarma ACTIVADA a dd/MM/yyyy HH:mm:ss”.
- En el programa principal crea un método `GestionAlarma` que implemente una Alarma, la haga funcionar con un solo avisado y que luego añada el resto de avidadores. Mostrando el flujo de activaciones/desactivaciones al variar las lecturas.

Ejercicio 5. Recetas e ingredientes con métodos de extensión

Vamos a practicar métodos de extensión a partir de un dominio sencillo: recetas de cocina. Partimos de dos representaciones de un ingrediente:

1. `Ingrediente` (catálogo): información general (nombre, unidad, tipo alimenticio y calorías por gramo)
2. `IngredienteReceta` (uso en una receta): añade la cantidad usada y las calorías totales calculadas.

Los métodos de extensión permiten pasar de un modelo al otro sin modificar los tipos originales. Una clase `Receta` almacenará los ingredientes usados y calculará el total de calorías.

```
Ejercicio 5: Recetas e ingredientes con métodos de extensión

Receta: Bizcocho ejemplo
Ingredientes:
    - 200 gramo de Harina (116,00 calorías)
    - 80 gramo de Azúcar (78,40 calorías)
    - 50 gramo de Mantequilla (360,00 calorías)
    - 2 unidad de Huevo (3,00 calorías)
Calorías totales: 557,40

Esta deliciosa receta está compuesta por la siguiente lista de ingredientes:
Harina
Azúcar
Mantequilla
Huevo
Presiona una tecla para finalizar...
```

Requisitos

- Define el enum `TipoIngrediente` con al menos [Proteina, Carbohidrato, Grasa, Vegetal, Fruta, Especia, Liquido]`.
- Crea el record base `Ingrediente(string Nombre, string Unidad, TipoIngrediente Tipo, double CaloriasPorGramo)` ; sobrescribe `ToString()` para que devuelva solo el nombre.
- Crea el record `IngredienteReceta(string Nombre, string Unidad, TipoIngrediente Tipo, double CaloriasPorGramo, int CantidadUsada, double CaloriasTotales)` .
- Crea la clase estática `IngredienteExtension` con el método de extensión `ToIngredienteReceta(this Ingrediente receta, int cantidadUsada)` que devuelve un `IngredienteReceta` calculando las calorías totales (cantidad * calorías/gramo).
- Crea la clase estática `IngredienteRecetaExtension` con el método `ToIngrediente(this IngredienteReceta receta)` que reconstruye el record base (sin cantidad ni calorías totales).
- Implementa la clase `Receta` con: propiedad `Nombre` , lista `List<IngredienteReceta> Ingredientes` , método `AgregaIngrediente(Ingrediente ingrediente, int cantidadUsada)` que usa el extensor para agregar un ingrediente con su cantidad a los ingredientes de la receta.Método `ListaIngredientes()` que devuelve una lista nueva de tipo `Ingrediente` (usando los métodos extensores de conversión) y método `CaloriasTotales()` que suma las calorías totales de la receta y las devuelve.

Ejercicio 6. Ampliando Interfaces genéricas. Temperaturas por provincia

Vamos utilizar interfaces para implementar una similitud del **patrón estratégia** a través de métodos estáticos.

```
Ejercicio 6. Ampliando Interfaces genéricas. Temperaturas por provincia


De cuantas provincias quieres recoger la temperatura: 3
Introduce la provincia nº1:  Alicante


Introduce la provincia nº2:  Castellón


Introduce la provincia nº3:  Valencia


Provincia: Alicante, Temperatura máxima:22ºC, Temperatura mínima: -2ºC.
Provincia: Castellón, Temperatura máxima:23ºC, Temperatura mínima: 2ºC.
Provincia: Valencia, Temperatura máxima:19ºC, Temperatura mínima: 5ºC.

Muestra las provincias con temperatura máxima superior a la media: 21,333334
Alicante
Castellón

Muestra las provincias con temperatura mínima inferior a la media: 1,6666666
Alicante

Muestra las provincias con temperatura mínima igual a la media: 1,6666666

Fin de la aplicación.
```

Requisitos

Para ello, vamos a definir en primer lugar la clase `TemperaturasXProvincia` que contendrá el nombre de una província y sus temperaturas máxima y mínima respectivamente.

```
class TemperaturasXProvincia
{
    public string Provincia { get; }
    public float TemperaturaMaxima { get; }
    public float TemperaturaMinima { get; }
    public TemperaturasXProvincia(string provincia,
                float temperaturaMaxima,
                float temperaturaMinima)
    {
        Provincia = provincia;
        TemperaturaMaxima = temperaturaMaxima;
        TemperaturaMinima = temperaturaMinima;
    }
    override public string ToString()
    {
        return $"Provincia: {Provincia}, Temperatura máxima:" +
               $"{TemperaturaMaxima}ºC, Temperatura mínima: {TemperaturaMinima}ºC.";
    }
}
```

Definiremos el interfaz **`IObtenTemperatura`** que obligará a implementar una '*estratégia*' de obtención de temperatura sobre un objeto de tipo **`TemperaturasXProvincia`**. Esto es, dado un objeto de tipo **`TemperaturasXProvincia`** me devolverá una de las temperaturas que contiene. En este caso la másxiam o la mínima pero piensa que en el futuro este tipo de objetos podría contener una propiedad `TemperaturaMedia` (fíjate en el programa principal para deducir el método de IObtenTemperatura).

Además, vamos a definir un interfaz parametrizado **`ICumplePredicado`** que oblige a implementar un método **`bool Predicado(T o1, T o2)`** al que le lleguen dos objetos y me devuelva true si cumplen un determinado predicado.

En la clase del programa principal, tendremos este método de utilidad que pedirá nombres de província y asignará aleatoriamente ambas temperaturas devolviéndome un array de **`TemperaturasXProvincia`**.

```
static TemperaturasXProvincia[] RecogeTemperaturasPorProvincia()
{
    Console.Write("\nDe cuantas provincias quieres recoger la temperatura: ");
    var temperaturasPorProvincia =
        new TemperaturasXProvincia[int.Parse(Console.ReadLine())];
    Random seed = new Random();
    for (int i = 0; i < temperaturasPorProvincia.Length; i++)
    {
        Console.Write($"Introduce la provincia nº{i + 1}:  ");
        string provincia = Console.ReadLine();
        float temperaturaMaxima = seed.Next(17, 25);
        float temperaturaMinima = seed.Next(-5, 17);
        Console.Write("\n\n");
        temperaturasPorProvincia[i] = new TemperaturasXProvincia(
                                        provincia,
                                        temperaturaMaxima,
                                        temperaturaMinima);
    }
    return temperaturasPorProvincia;
}
```

Se pide:

1. Implementar en la clase principal un método llamado `MediaTemperaturas` al que le pasemos el array de `TemperaturasXProvincia` y un objeto que implemente la estrategia definida en `IObtenTemperatura`. De tal manera que, sin cambiar el método, pueda calcular la media de las máximas, de las mínimas o en un futuro de las medias.
2. Implementar en la clase principal un método llamado `MuestraProvincias` al que le pasemos el array de `TemperaturasXProvincia` un valor de `temperatura`, un objeto que implemente la estrategia definida en `IObtenTemperatura` y un objeto que implemente un predicado definido en `ICumplePredicado`. De tal manera que me muestre aquellas provincias cuya temperatura obtenida por `IObtenTemperatura` cumpla un determinado predicado.
3. Crea un programa principal que usando los métodos definidos anteriormente...
    i. Muestre las provincias cuya máxima sea mayor a la media de las máximas.
    ii. Muestre las provincias cuya mínima sea menor a la media de las mínimas.
    iii. Muestre las provincias cuya mínima sea igual a la media de las mínimas.

> 💡 **Pista:** Puedes definir los siguientes tipos/clases públicas para usar en el `Main` que implementen las estrategias de obtención de temperatura y los predicados necesarios **dentro de la clase** `TemperaturasXProvincia`

- class `ObtenMaxima` que me permita obtener la temperatura máxima.
- class `ObtenMinima` que me permita obtener la temperatura máxima.
- class `MayorQue` que me si una temperatura es mayor que la otra.
- class `MenorQue` que me si una temperatura es menor que la otra.
- class `IgualQue` que me si dos temperaturas son iguales.