Índice

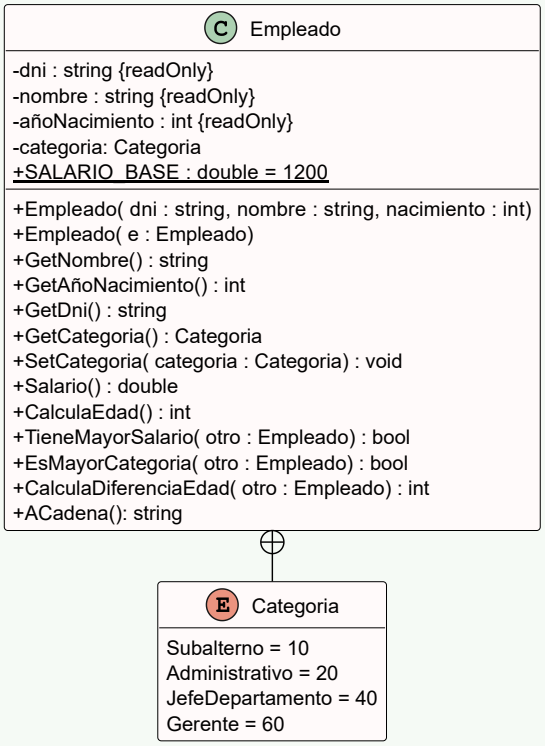
Ejercicios Unidad 13 - Definir tipos POO

Descargar estos ejercicios

> **Antes de empezar**
>
> Para realizar estos ejercicios, deberás descargar los recursos del enlace de proyecto_definir_tipos. 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. Clase Empleado con métodos Get/Set tradicionales

A partir del siguiente diagrama de clases, crea la clase necesaria que permita reflejar los elementos que ves representados en él, y la aplicación para probarla. Las categorías de los trabajadores pueden ser: `Subalterno = 10, Administrativo = 20, JefeDepartamento = 40, Gerente = 60`.



> 📌 **Nota:** El salario del empleado se calculará a partir de la categoría de trabajo que desempeñe, las categorías tienen asociadas el incremento en porcentaje que se tiene que hacer al salario base, que será una constante con el valor de 1200.

```
Ejercicio 1: Clase Empleado con métodos Get/Set

=== CREACIÓN DE EMPLEADOS ===
Introduce datos del primer empleado:
DNI: 12345678A
Nombre: Ana García Martín
Año de nacimiento: 1985
Categoría (Subalterno/Administrativo/JefeDepartamento/Gerente): Administrativo

Introduce datos del segundo empleado:
DNI: 87654321B
Nombre: Carlos López Ruiz
Año de nacimiento: 1978
Categoría (Subalterno/Administrativo/JefeDepartamento/Gerente): JefeDepartamento

=== INFORMACIÓN DE EMPLEADOS ===
--- Empleado 1 ---
DNI: 12345678A
Nombre: Ana García Martín
Año de nacimiento: 1985
Edad: 40 años
Categoría: Administrativo
Salario: 1440.00€ (base: 1200€ + 20% incremento)

--- Empleado 2 ---
DNI: 87654321B
Nombre: Carlos López Ruiz
Año de nacimiento: 1978
Edad: 47 años
Categoría: JefeDepartamento
Salario: 1680.00€ (base: 1200€ + 40% incremento)

=== PROMOCIONES Y CAMBIOS ===
Promocionando a Ana García a JefeDepartamento...
Promocionando a Carlos López a Gerente...

--- Estado después de promociones ---
Empleado 1:
DNI: 12345678A
Nombre: Ana García Martín
Año de nacimiento: 1985
Edad: 40 años
Categoría: JefeDepartamento
Salario: 1680.00€ (base: 1200€ + 40% incremento)

Empleado 2:
DNI: 87654321B
Nombre: Carlos López Ruiz
Año de nacimiento: 1978
Edad: 47 años
Categoría: Gerente
Salario: 1920.00€ (base: 1200€ + 60% incremento)

=== CREACIÓN DE COPIA ===
Creando copia de Ana García...
Empleado copiado:
DNI: 12345678A
```

```
Nombre: Ana García Martín
Año de nacimiento: 1985
Edad: 40 años
Categoría: JefeDepartamento
Salario: 1680.00€ (base: 1200€ + 40% incremento)

=== COMPARACIONES ===
¿Ana García tiene mayor salario que Carlos López? False
¿Carlos López es de mayor categoría que Ana García? True
Diferencia de edad: 7 años

Console.WriteLine("Presiona cualquier tecla para salir...");
Console.ReadKey();
```

Requisitos:

- Al crear un empleado por primera vez, se le asignará la categoría Subalterno.
- `Salario` método con cuerpo de expresión que devuelva el salario *(SALARIO_BASE + porcentaje según categoría)*.
- `CalculaEdad` método con cuerpo de expresión que devuelva el salario *(año actual - año nacimiento)*.
- `TieneMayorSalario` método con cuerpo de expresión que devolverá true si el empleado objeto tiene más salario que el pasado como parámetro.
- `EsMayorCategoria` método con cuerpo de expresión que devolverá true si el empleado objeto tiene una categoría mayor que la pasada por parámetro.
- Método `CalculaDiferenciaEdad` método con cuerpo de expresión que devolverá los años de diferencia entre ambos empleados.
- `ACadena()` devuelve una cadena con la información completa del empleado.

Métodos staticos en clase Program:

- `PideEmpleado()` método estático que solicita datos por consola y retorna un nuevo empleado. Importante!!, para mostrar las categorías existentes se deberá usar `GetNames` de la clase `Enum` y `Join` de la clase `String` .
- `ParseaCategoria(string)` . Convierte un string (ej: "Administrativo") al enum correspondiente. Debe ser case-insensitive (no distinguir mayús/minús). Si el texto no corresponde a ninguna categoría válida, retorna `Categoria.Subalterno` por defecto. Aceptará tanto el nombre completo como valores numéricos: "20" → `Categoria.Administrativo`
- Programa principal creando los objetos necesarios y llamadas a los métodos para conseguir el resultado de la salida

Ejercicio 2. Clase Producto con propiedades no autoimplementadas

Crea una clase `Producto` que represente productos de un inventario, usando propiedades con lógica de validación en los setters.

```
Ejercicio 2: Clase Producto con propiedades no autoimplementadas

=== CREACIÓN DE PRODUCTOS ===
Creando productos del inventario...

Producto 1: Laptop Gaming
- Código: PRD001
- Precio inicial: 1200.00€
- Stock inicial: 15

Producto 2: Mouse Inalámbrico
- Código: PRD002
- Precio inicial: 25.50€
- Stock inicial: 50

=== OPERACIONES CON STOCK ===
--- Venta de productos ---
Vendiendo 5 unidades de Laptop Gaming...
Stock actualizado: 10 unidades

Vendiendo 20 unidades de Mouse Inalámbrico...
Stock actualizado: 30 unidades

--- Reposición de stock ---
Reponiendo 8 unidades de Laptop Gaming...
Stock actualizado: 18 unidades

Intentando vender 25 unidades de Laptop Gaming...
ERROR: Stock insuficiente. Stock actual: 18, solicitado: 25

=== CAMBIOS DE PRECIO ===
--- Aplicar descuentos ---
Aplicando 10% de descuento a Laptop Gaming...
Precio anterior: 1200.00€
Precio nuevo: 1080.00€

Aplicando 20% de descuento a Mouse Inalámbrico...
Precio anterior: 25.50€
Precio nuevo: 20.40€

--- Intentar precio inválido ---
Intentando establecer precio negativo en Laptop Gaming...
ERROR: El precio no puede ser negativo o cero

=== ESTADO FINAL DEL INVENTARIO ===
Producto 1:
- Código: PRD001
- Nombre: Laptop Gaming

Console.WriteLine("Presiona cualquier tecla para salir...");
Console.ReadKey();
```

Requisitos:

- **Campos privados**:

    - `_codigo` (string): Código único generado automáticamente
    - `_nombre` (string): Nombre del producto
    - `_precio` (double): Precio unitario del producto
    - `_stock` (int): Cantidad en inventario
    - `codigosGenerados` (string[]): variable estática que guardará cada uno de los códigos generados, para comprobar que no se repitan.
- **Constructor**:
    - `Producto(string nombre, double precio, int stock)` . Generará un código único con las letras PR un número y tres letras mayúsculas(PR2MXC, PR9LÑK, PR9LLL, etc). Antes de asignarse el código, se tendrá que haber comprobado que no se había asignado con anterioridad. Importante, se deberán usar las propiedades para validar precio y stock iniciales.
- **Propiedades no autoimplementadas públicas**:
    - `Codigo` readonly, devuelve el código
    - `Nombre` readonly, devuelve el nombre
    - `Precio` propiedad accesora y mutadora. Validará que sea > 0, o asignará un precio por defecto de 20 euros.
    - `Stock` propiedad accesora y mutadora que validará que sea >= 0, o asignando un valor de 1 en caso contrario.
    - `CodigosGenerados` propiedad estática y pública de acceso sobre el campo de tipo array codigosGenerados.
- **Propiedades calculadas públicas**:
    - `ValorTotal` de tipo double (precio × stock)
    - `Estado` → string ("Sin stock", "Stock bajo" si < 10, "En stock")
- **Métodos**:
    - `Vende(int cantidad)` método que reduce el stock en la cantidad que llega como parámetro, solo si hay suficiente stock. Retorna true si es posible.
    - `Repone(int cantidad)` método que aumenta el stock en la cantidad que llega como parámetro.
    - `AplicaDescuento(double porcentaje)` método que reduce el precio al porcentaje dado.
    - `ACadena` método que devuelve una cadena con la información del producto
- **Métodos staticos en clase Program**:
    - `CreaProducto()` método que pide los datos necesarios para crear un producto y lo devuelve.
    - Programa principal creando los objetos necesarios y llamadas a los métodos para conseguir el resultado de la salida.

Ejercicio 3. Sistema de dibujo con herramientas

Crea un proyecto con **los TAD necesarios** para que el siguiente código perteneciente a la Main, pueda ser ejecutado sin problemas:

```
Compas compas = new Compas();
Circulo circulo = compas.DibujaCirculo(3.5f);
Rotulador rotulador = Estuche.GetRotuladores()
                      [
                          new Random().Next(0, Estuche.NUMERO_ROTULADORES)
                      ];
rotulador.Rotula(circulo.Perimetro());
Pincel pincel = new Pincel();
pincel.SetColor(Color.Verde);
pincel.Pinta(circulo.Area());
```

```
Ejercicio 3: Sistema de dibujo con herramientas

Dibujado un círculo de radio 3,5 cm
Rotulado el perímetro de 21,99 cm de color Rojo.
Pintada el área de 38,48 cm² de color Verde.

¡Dibujo completado con éxito!
Presiona cualquier tecla para salir...
```

Ejercicio 4. Clase Vehiculo con propiedades autoimplementadas

Crea una clase `Vehiculo` que represente vehículos de un concesionario usando propiedades autoimplementadas.

```
Ejercicio 4: Clase Vehículo con propiedades autoimplementadas

=== CREACIÓN DE VEHÍCULOS ===
Registrando vehículos en el concesionario...

Vehículo 1:
- Matrícula: 1234ABC
- Marca: Toyota
- Modelo: Corolla
- Año: 2022
- Precio: 25000.00€
- Kilómetros: 15000
- Estado: Usado

Vehículo 2:
- Matrícula: 5678DEF
- Marca: BMW
- Modelo: Serie 3
- Año: 2025
- Precio: 45000.00€
- Kilómetros: 0
- Estado: Nuevo

=== OPERACIONES CON VEHÍCULOS ===
--- Actualizaciones de kilometraje ---
Toyota Corolla recorre 5000 km adicionales...
Kilómetros totales: 20000 km

BMW Serie 3 recorre 1200 km (entrega a cliente)...
Kilómetros totales: 1200 km

--- Cambios de precio ---
Aplicando descuento del 8% al Toyota Corolla...
Precio anterior: 25000.00€
Precio actual: 23000.00€

Aplicando descuento del 5% al BMW Serie 3...
Precio anterior: 45000.00€
Precio actual: 42750.00€

=== ANÁLISIS DEL INVENTARIO ===
--- Depreciación por uso ---
Toyota Corolla:
- Depreciación por kilómetros: 2000.00€ (20000 km × 0.10€/km)
- Valor con depreciación: 21000.00€

BMW Serie 3:
- Depreciación por kilómetros: 120.00€ (1200 km × 0.10€/km)
- Valor con depreciación: 42630.00€

=== ESTADO FINAL DEL INVENTARIO ===
Vehículo 1:
- Matrícula: 1234ABC
- Vehículo: Toyota Corolla (2022)
- Precio de venta: 23000.00€
- Kilómetros: 20000 km
- Estado: Usado
```

```
- Valor real: 21000.00€
- Antigüedad: 3 años

Vehículo 2:
- Matrícula: 5678DEF
- Vehículo: BMW Serie 3 (2025)
- Precio de venta: 42750.00€
- Kilómetros: 1200 km
- Estado: Nuevo
- Valor real: 42630.00€
- Antigüedad: 0 años

Console.WriteLine("Presiona cualquier tecla para salir...");
Console.ReadKey();
```

Requisitos:

- **Propiedades autoimplementadas**:
  - `Matricula` string
  - `Marca` string
  - `Modelo` string
  - `Año` int
  - `Precio` double
  - `Kilometros` int

    Debes tener en cuenta que los set serán privados para que solo pueda modificarse el objeto a través de los métodos capacitados para ello.
- **Propiedades calculadas**:
  - `Estado` string ("Nuevo" si km ≤ 1000, "Usado" si > 1000)
  - `Antigüedad` int (año actual - año del vehículo)
  - `DepreciacionKilometraje` double (kilómetros × 0.10€)
  - `ValorReal` double (precio - depreciación por kilometraje)
  - `NombreCompleto` string ("Marca Modelo (Año)")
- **Métodos**:
  - `Vehiculo(string matricula, string marca, string modelo, int año, double precio, int kilometros)`
  - `AñadeKilometros(int km)` método que suma los kilómetros de entrada al total.
  - `AplicaDescuento(double porcentaje)` método que reduce el precio aplicando el porcentaje de entrada.
  - `ActualizaPrecio(double nuevoPrecio)` modifica el precio del vehículo, a el nuevo precio de entrada.
  - `ACadena` que devuelve una cadena con la información del vehículo.

Ejercicio 5. Records como Value Objects

Implementa un sistema que use dos tipos de `record` como Value Objects: uno simplificado y otro extendido. El programa debe demostrar el uso de `with` para crear copias modificadas en el record simplificado, y la creación de nuevos objetos en el record extendido.

```
Ejercicio 5: Records como Value Objects

=== RECORDS SIMPLIFICADOS (Value Objects sin validación) ===
Creando juguetes con records simplificados...

Juguete original: Juguete { Nombre = "Pelota", Precio = 15.50 }
Juguete con descuento usando 'with': Juguete { Nombre = "Pelota", Precio = 12.40 }
Juguete renombrado usando 'with': Juguete { Nombre = "Pelota Roja", Precio = 15.50 }
Juguete modificado completamente: Juguete { Nombre = "Muñeca", Precio = 25.99 }

--- Operaciones con juguetes ---
Precio promedio entre juguete original y con descuento: 13.95€
Diferencia de precio: 3.10€

=== RECORDS EXTENDIDOS (Value Objects con validación) ===
Creando temperaturas con records extendidos y validación...

--- Creación exitosa ---
Temperatura ambiente: Temperatura { Grados = 22.5, Escala = Celsius }
Temperatura agua hirviendo: Temperatura { Grados = 100.0, Escala = Celsius }
Temperatura agua congelando: Temperatura { Grados = 32.0, Escala = Fahrenheit }

--- Conversiones de temperatura (creando nuevos objetos) ---
22.5°C convertido a Fahrenheit: Temperatura { Grados = 72.5, Escala = Fahrenheit }
100°C convertido a Kelvin: Temperatura { Grados = 373.15, Escala = Kelvin }
32°F convertido a Celsius: Temperatura { Grados = 0.0, Escala = Celsius }
```

Requisitos:

- **Record simplificado Juguete**:
  - Crea el recor `Juguete` con las propiedades Nombre de tipo string y Precio de tipo double.
  - `CalculaDescuento(double porcentaje)` método que devuelve el Juguete con el precio modificado con el descuento.
- **Record extendido Temperatura**:
  Este Value Object tendrá una propiedad de tipo enumerado `EscalaTemperatura` con los valores (Celsius, Fahrenheit, Kelvin)
  - Record `Temperatura` con las propiedades Grados de tipo double y Escala de tipo EscalaTemperatura.
  - Constructor con validación (no permite temperaturas bajo cero absoluto). Validaremos según la información de la ayuda.

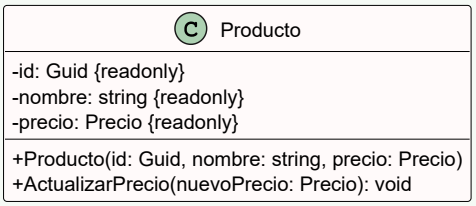
> **"Valores del cero absoluto según la escala"**
>
> - **Celsius**: -273.15°C (cero absoluto)
> - **Fahrenheit**: -459.67°F (cero absoluto)
> - **Kelvin**: 0.0ºK (cero absoluto)
>
> Si intentas crear una temperatura inferior a estos valores, el constructor debe asignar automáticamente el valor del cero absoluto correspondiente a la escala.

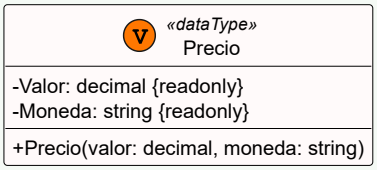
  - Métodos `ConvierteA(EscalaTemperatura nueva)` que devolverá la nueva Temperatura calculada a partir de la escala que llega.
- **En el programa principal**:
  - Tendremos el método `GestionJuguete`, que crea un juguetes base y se modifica con `with` para conseguir la salida. Además de calcular descuentos y comparar precios.
  - Tendremos el método `GestionTemperatura` que creará temperaturas válidas e inválidas, convertirá entre escalas de temperatura.

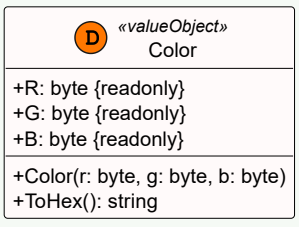
Ejercicio 6: Diagrama UML con diferentes tipos de definición

Implementa el siguiente diagrama UML que muestra los **tres tipos de definición** que hemos visto en esta unidad:

- Una **clase tradicional** (Entity)
- Un **value object por valor** (struct)
- Un **value object por referencia** (record)







```
Ejercicio 6: Diagrama UML con diferentes tipos de definición

PRODUCTO (Clase tradicional):
Producto: Laptop Gaming - 1299,99 EUR

PRECIO (Value Object por valor):
Precio: 1299,99 EUR - Válido: True

COLOR (Value Object por referencia):
Color RGB: (255, 0, 0) - Hex: FF0000
```

Requisitos:

- Implementa los tres tipos del uml como se ha explicado en los apuntes.
- Método `ToHex()` que retorne el color en formato hexadecimal (ej: "#FF0000") a partir de las propiedades.
- En el `Main`, crea instancias de los tres tipos y muestra sus valores.