C# Y .NET 10
Parte 4.
Programación Orientada a Objetos

2026-01

Rafael Alberto Moreno Parra
ramsoftware@gmail.com

Contenido

Tabla de ilustraciones

Acerca del autor

Rafael Alberto Moreno Parra

ramsoftware@gmail.com o enginelife@hotmail.com

Sitio Web: http://darwin.50webs.com (dedicado a la investigación de algoritmos evolutivos y vida artificial).

Github: https://github.com/ramsoftware

Youtube: https://www.youtube.com/@RafaelMorenoP

Licencia de este libro

Licencia del software

Todo el software desarrollado aquí tiene licencia LGPL "Lesser General Public License" [1]

Marcas registradas

En este libro se hace uso de las siguientes tecnologías registradas:

Microsoft ® Windows ® Enlace: http://windows.microsoft.com/en-US/windows/home

Microsoft ® Visual Studio 2026 ® Enlace: https://visualstudio.microsoft.com/es/vs/

Dedicatoria

A mis padres, a mi hermana….

Y a mi tropa gatuna: Suini, Grisú, Milú, Arián, Frac y mis recordados Sally, Capuchina, Tinita, Tammy, Vikingo y Michu.

Definir una clase

Al definir una clase en C#, se pueden hacer uso de atributos privados (con la palabra reservada private), atributos públicos (con la palabra reservada public, pero no es recomendado), métodos privados y públicos.

D/001.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus atributos
//y métodos (encapsulación)
internal class MiClase {
   //Atributos privados
   private int Numero;
   private char Letra;
   private string Cadena;
   private double Valor;

   //Atributos públicos (no recomendado)
   public int Acumula;
   public char Caracter;

   //Métodos privados
   private double Maximo(double numA, double numB, double numC) {
      double max = numA;
      if (max < numB) max = numB;
      if (max < numC) max = numC;
      return max;
   }

   //Métodos públicos
   public double Promedio(double numA, double numB, double numC) {
      return (numA + numB + numC) / 3;
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
      MiClase Objeto = new();

      //Solo puede llamar al método público de MiClase
      double resultado = Objeto.Promedio(1, 7, 8);
      Console.WriteLine(resultado);
   }
}
```


Autor: Rafael Alberto Moreno Parra

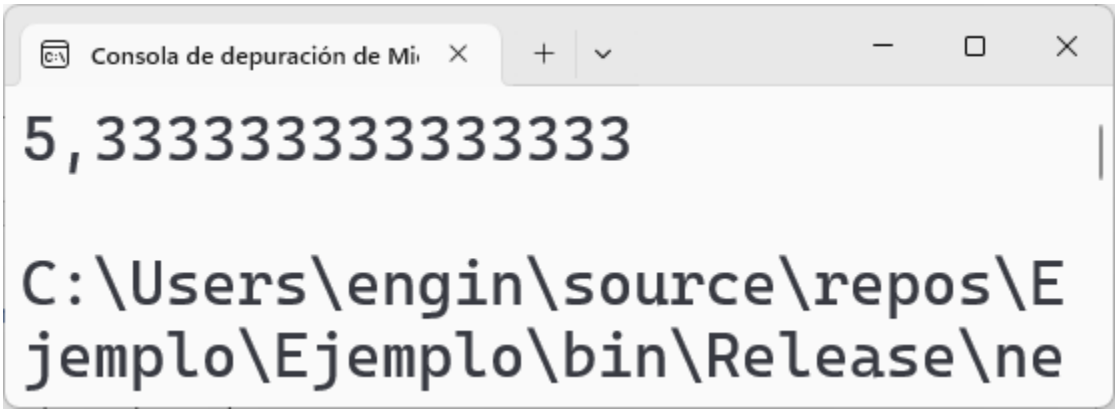

Ilustración 1: Atributos/Métodos privados y públicos

Errores al tratar de acceder a atributos o métodos privados

Si se intenta hacer uso de un método privado o acceder a un atributo privado, se generará un error en tiempo de compilación.

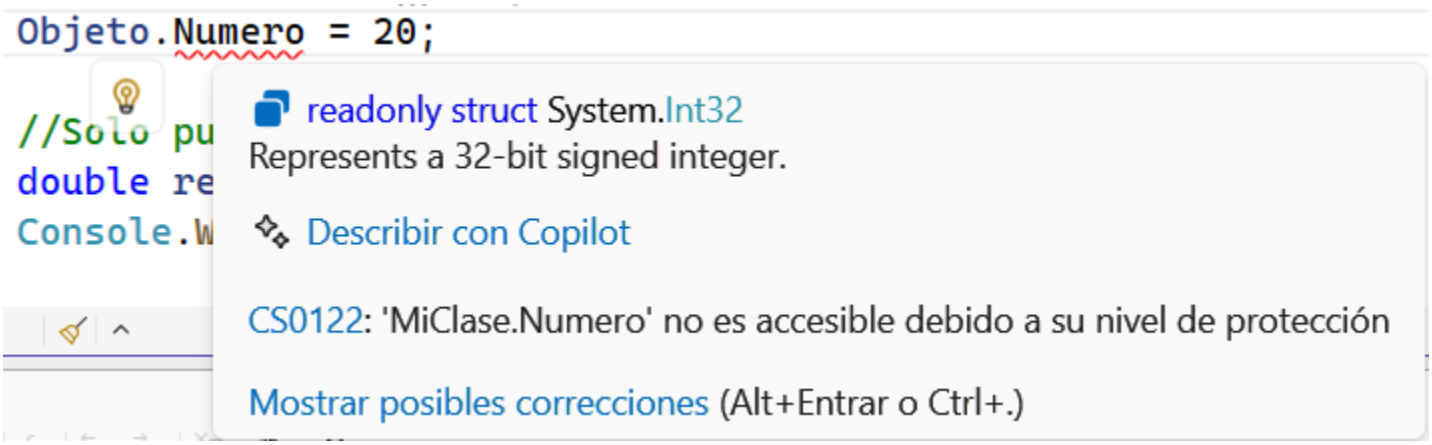

Ilustración 2: Intento de acceder a un atributo privado

Los atributos deben ser privados. Accediendo a ellos.

La recomendación es que los atributos de una clase siempre sean privados. Así que, para acceder a ellos desde una instancia, se debe hacer a través de métodos públicos de lectura y escritura. Esos métodos son conocidos en el medio como getters y setters.

Cada atributo se le crea un método que en su interior tiene el "get" (para leer) o el "set" (para darle valor)

D/002.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus atributos
//y métodos (encapsulación)
class MiClase {
   //Atributos privados
   private int numero;
   private char letra;
   private string cadena;
   private double valor;

   //Los getters y setters
   public int Numero { get => numero; set => numero = value; }
   public char Letra { get => letra; set => letra = value; }
   public string Cadena { get => cadena; set => cadena = value; }
   public double Valor { get => valor; set => valor = value; }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
      MiClase Objeto = new();

      //Llama los setters
      Objeto.Cadena = "Suini, Capuchina, Grisú, Milú";
      Objeto.Numero = 7;
      Objeto.Letra = 'R';
      Objeto.Valor = 16.83;

      //Usa los getters
      Console.WriteLine(Objeto.Letra);
      Console.WriteLine(Objeto.Valor);
      Console.WriteLine(Objeto.Cadena);
      Console.WriteLine(Objeto.Numero);
   }
}
```


Autor: Rafael Alberto Moreno Parra

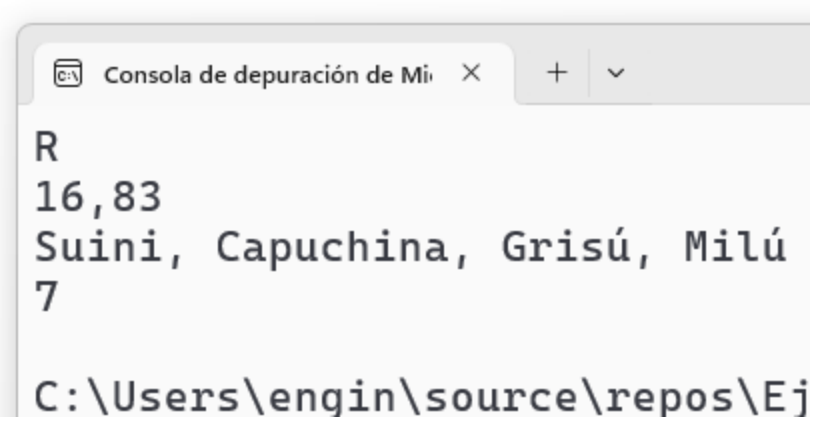

Ilustración 3: Los atributos deben ser privados. Accediendo a ellos.

Más información en: https://learn.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties

Forma reducida de los getters y setters

D/003.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus atributos
//y métodos (encapsulación)
class MiClase {
   //Otra forma de definir atributos con los getters y setters
   public int Numero { get; set; }
   public char Letra { get; set; }
   public string Cadena { get; set; }
   public double Valor { get; set; }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
      MiClase Objeto = new();

      //Llama los setters
      Objeto.Cadena = "Suini, Capuchina, Grisú, Milú";
      Objeto.Cadena += ", Sally, Vikingo, Arian, Frac";
      Objeto.Numero = 7;
      Objeto.Letra = 'R';
      Objeto.Valor = 93.5;

      //Usa los getters
      Console.WriteLine(Objeto.Letra);
      Console.WriteLine(Objeto.Valor);
      Console.WriteLine(Objeto.Cadena);
      Console.WriteLine(Objeto.Numero);
   }
}
```

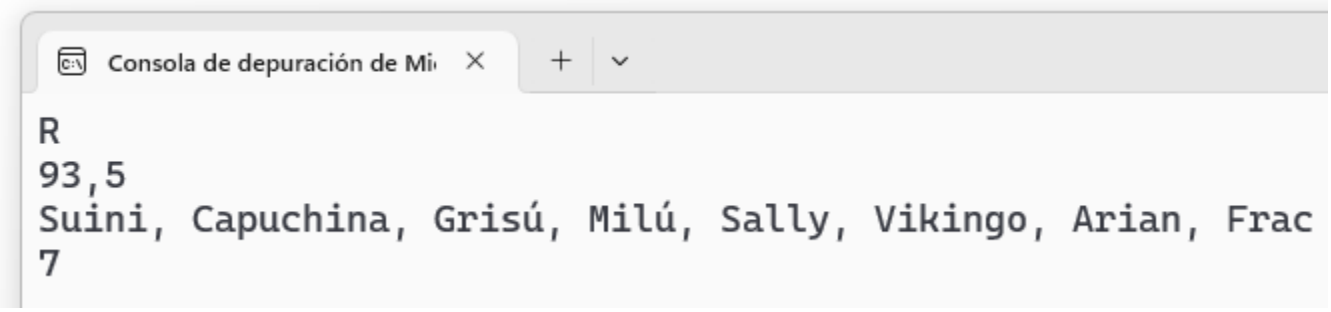

Ilustración 4: Forma reducida de los getters y setters

Uso de los getters/setters

Observe que el atributo (como variable) es privado, por lo general se escribe en minúsculas, luego se crea el acceso a ese atributo, copiando el nombre del atributo pero poniendo la primera en mayúsculas (es recomendado hacerlo así, no obligatorio). Es en ese acceso donde se pone el código para el get y el set.

D/004.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus atributos
//y métodos (encapsulación)
class MiClase {
   //Atributos privados. Un uso de los getters y setters
   private int numero;
   private char letra;
   private string cadena;
   private double valor;

   //Puede auditar cuando se leyó o
   //cambió el valor de un atributo
   public int Numero {
      get {
         Console.WriteLine("Lee numero: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return numero;
      }
      set {
         Console.WriteLine("Cambia numero: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         numero = value;
      }
   }

   public char Letra {
      get {
         Console.WriteLine("Lee letra: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return letra;
      }
      set {
         Console.WriteLine("Cambia letra: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         letra = value;
```

```
      }
   }

   public string Cadena {
      get {
         Console.WriteLine("Lee cadena: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return cadena;
      }
      set {
         Console.WriteLine("Cambia cadena: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         cadena = value;
      }
   }

   public double Valor {
      get {
         Console.WriteLine("Lee valor: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return valor;
      }
      set {
         Console.WriteLine("Cambia valor: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         valor = value;
      }
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
      MiClase Objeto = new();

      //Llama los setters
      Objeto.Cadena = "Suini, Capuchina, Grisú, Milú";
      Objeto.Numero = 7;
      Objeto.Letra = 'R';
      Objeto.Valor = 93.5;

      //Usa los getters
      char unaletra = Objeto.Letra;
```

```
        double unvalor = Objeto.Valor;
        string unacadena = Objeto.Cadena;
        int unnumero = Objeto.Numero;

        Console.WriteLine("Letra es: " + unaletra);
        Console.WriteLine("Valor es: " + unvalor);
        Console.WriteLine("Cadena es: " + unacadena);
        Console.WriteLine("Número es: " + unnumero);
    }
}
```

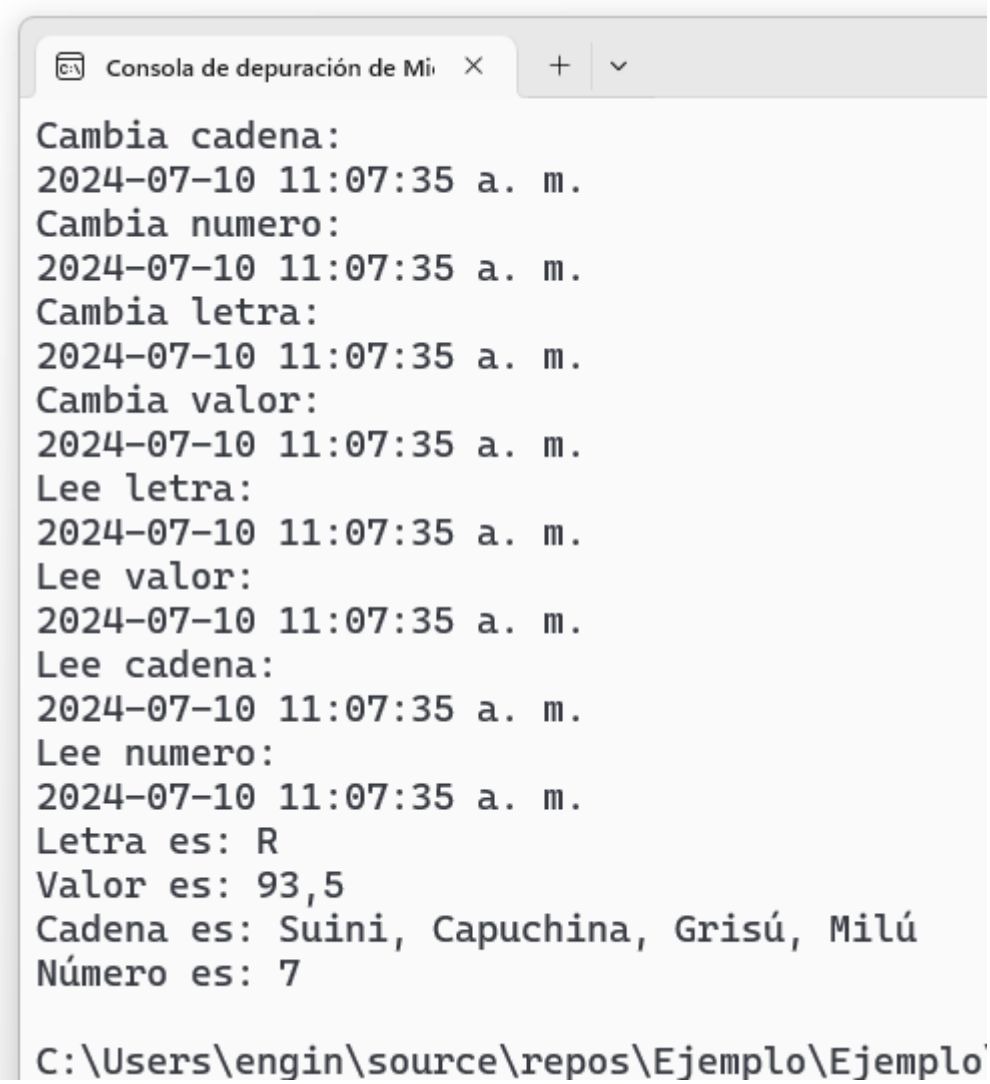

Ilustración 5: Uso de los getters/setters

Otro uso de los getters y setters

Para validar el dato de entrada

D/005.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus
//atributos y métodos (encapsulación)
class MiClase {
   //Atributos privados. Un uso de los getters y setters
   private int edad;

   //Puede validar el dato de inicialización
   public int Edad {
      get {
         return edad;
      }
      set {
         if (value < 0)
            Console.WriteLine("Error: edad negativa");
         else
            edad = value;
      }
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
      MiClase Objeto = new();
      MiClase Otro = new();

      //Llama los setters
      Objeto.Edad = 17;
      Otro.Edad = -8;

      Console.WriteLine("Edad es: " + Objeto.Edad);
      Console.WriteLine("Edad es: " + Otro.Edad);
   }
}
```

Al intentar dar un valor a un atributo el setter valida si ese dato es válido. Dado el caso, asigna el valor, de lo contrario, muestra un mensaje y el dato no es asignado.

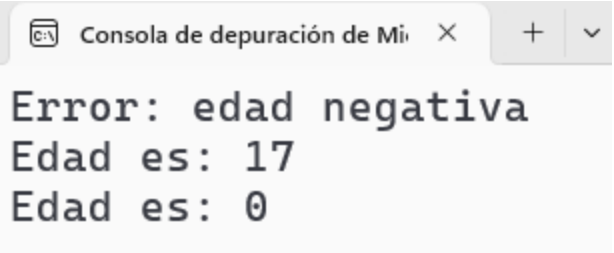

Ilustración 6: Otro uso de los getters y setters

Forma de inicializar los objetos llamando los setters

D/006.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus
//atributos y métodos (encapsulación)
class MiClase {
   //Atributos privados. Un uso de los getters y setters
   private int numero;
   private char letra;
   private string cadena;
   private double valor;

   //Puede auditar cuando se leyó o cambió
   //el valor de un atributo
   public int Numero {
      get {
         Console.WriteLine("Lee numero: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return numero;
      }
      set {
         Console.WriteLine("Cambia numero: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         numero = value;
      }
   }

   public char Letra {
      get {
         Console.WriteLine("Lee letra: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return letra;
      }
      set {
         Console.WriteLine("Cambia letra: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         letra = value;
      }
   }

   public string Cadena {
```

```
      get {
         Console.WriteLine("Lee cadena: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return cadena;
      }
      set {
         Console.WriteLine("Cambia cadena: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         cadena = value;
      }
   }

   public double Valor {
      get {
         Console.WriteLine("Lee valor: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         return valor;
      }
      set {
         Console.WriteLine("Cambia valor: ");
         string T = DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt");
         Console.WriteLine(T);
         valor = value;
      }
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase.
      //Otra forma de inicializar los atributos.
      MiClase Objeto = new MiClase {

         //Llama los setters
         Cadena = "Suini, Capuchina, Grisú, Milú, Sally, Vikingo",
         Numero = 7,
         Letra = 'R',
         Valor = 93.5
      };

      //Usa los getters
      char unaletra = Objeto.Letra;
      double unvalor = Objeto.Valor;
      string unacadena = Objeto.Cadena;
```

```
        int unnumero = Objeto.Numero;

        Console.WriteLine("Letra es: " + unaletra);
        Console.WriteLine("Valor es: " + unvalor);
        Console.WriteLine("Cadena es: " + unacadena);
        Console.WriteLine("Número es: " + unnumero);
    }
}
```

Ilustración 7: Forma de inicializar los objetos llamando los setters

Dos variables refiriendo al mismo objeto

D/007.cs

```
namespace Ejemplo;

//Esta es una clase propia con sus
//atributos y métodos (encapsulación)
class MiClase {
   //Otra forma de definir atributos con
   //los getters y setters
   public int Numero { get; set; }
   public char Letra { get; set; }
   public string Cadena { get; set; }
   public double Valor { get; set; }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase.
      MiClase Mascotas = new MiClase {
         Cadena = "Suini, Capuchina, Grisú, Milú",
         Numero = 7,
         Letra = 'R',
         Valor = 93.5
      };

      //Crea una variable de tipo MiClase
      MiClase otra;

      //Asigna el primer objeto a esa variable
      otra = Mascotas;

      //¿Qué sucede? Que tenemos dos variables
      //apuntando al mismo objeto en memoria

      //Se imprimen los valores de ambas variables
      Console.WriteLine("Letra en Mascotas: " + Mascotas.Letra);
      Console.WriteLine("Valor en Mascotas: " + Mascotas.Valor);
      Console.WriteLine("Letra en otraVariable: " + otra.Letra);
      Console.WriteLine("Valor en otraVariable: " + otra.Valor);

      //Si se modifican los valores en "otra"
      //afecta a Mascotas porque ambas apuntan
      //al mismo objeto en memoria
      otra.Valor = 12345.67;
      Console.WriteLine("Nuevo Valor: " + Mascotas.Valor);
```

```
    }
}
```

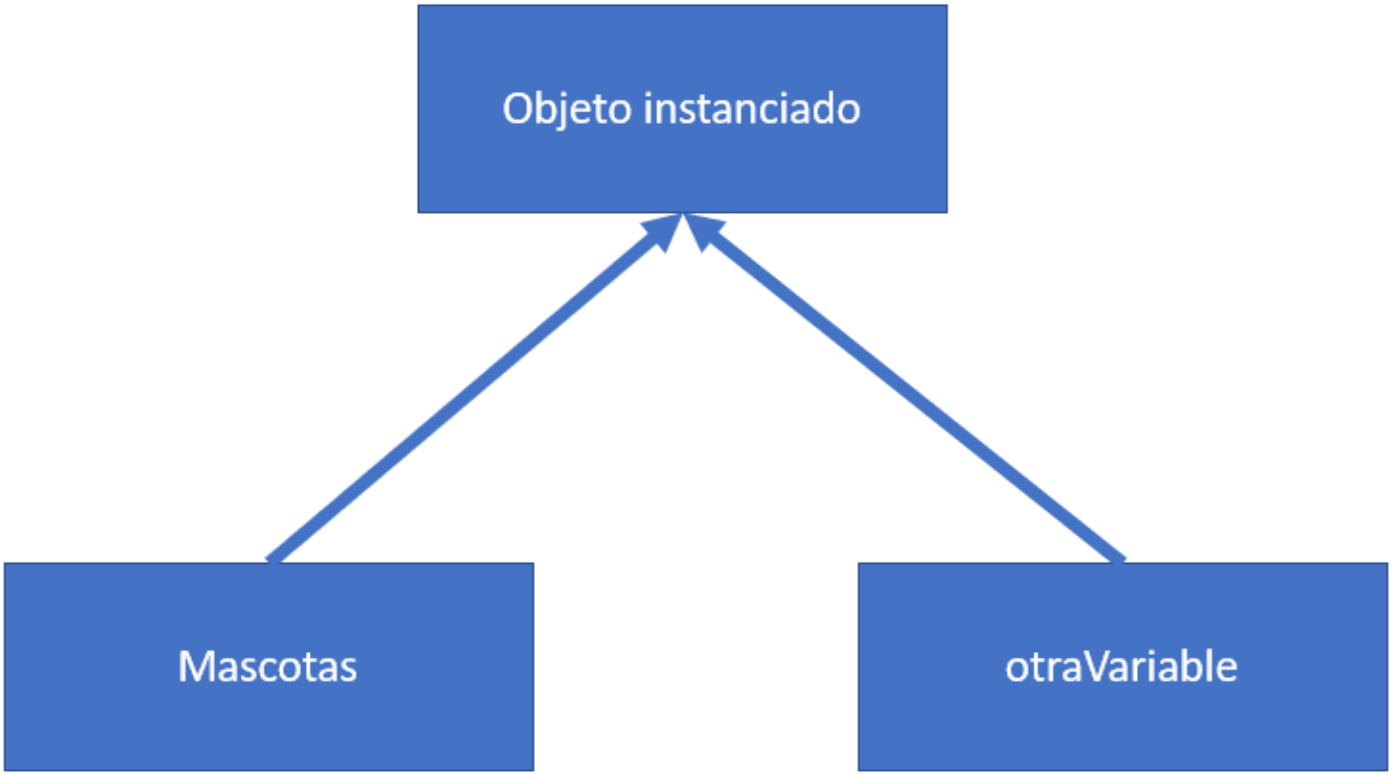

Ilustración 8: Ambas variables apuntan al mismo objeto instanciado

Dos variables apuntando al mismo objeto en memoria, eso es una copia superficial o “Shallow Copy”.

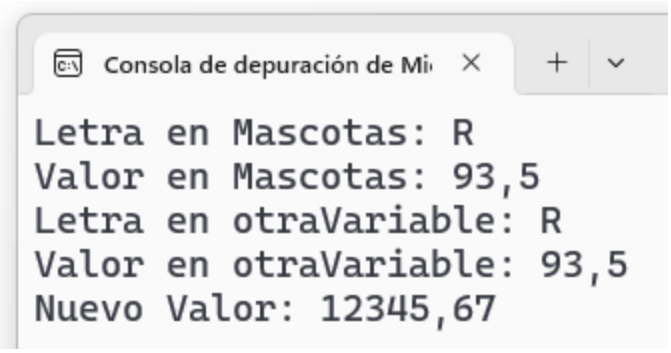

Ilustración 9: Dos variables refiriendo al mismo objeto

Sobrecarga de métodos

Dependiendo del número y tipo de parámetros C# sabe que método usar así tenga el mismo nombre.

A continuación, una clase con tres métodos con el mismo nombre, pero cada uno de los métodos tiene diferente número de parámetros.

Por número de parámetros

D/008.cs

```
namespace Ejemplo;

class Geometria {
   //Calcula el área del círculo
   public double Area(double radio) {
      return Math.PI * Math.Pow(radio, 2);
   }

   //Calcula el área del rectángulo
   public double Area(double baseR, double alturaR) {
      return baseR * alturaR;
   }

   //Calcula el área del triángulo
   public double Area(double ladoA, double ladoB, double ladoC) {
      double S = (ladoA + ladoB + ladoC) / 2;
      return Math.Sqrt(S * (S - ladoA) * (S - ladoB) * (S - ladoC));
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto Geometria
      Geometria geometria = new();

      //Dependiendo del número de parámetros
      //llama a un método u otro
      double areaCirculo = geometria.Area(8);
      double areaTriangulo = geometria.Area(4, 5, 6);
      double areaRectangulo = geometria.Area(17, 19);

      Console.WriteLine("Área del círculo: " + areaCirculo);
      Console.WriteLine("Área del triángulo: " + areaTriangulo);
      Console.WriteLine("Área del rectángulo: " + areaRectangulo);
```

```
    }
}
```

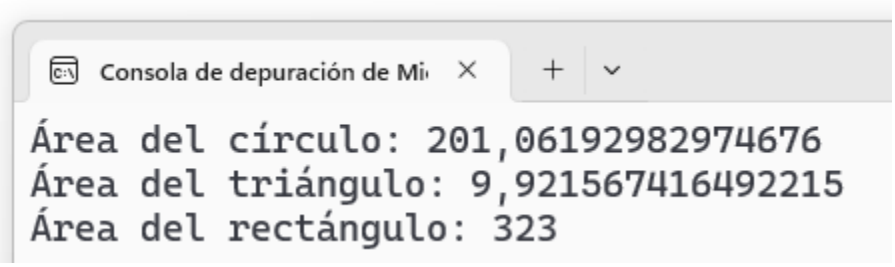

Ilustración 10: Sobrecarga de métodos

Por tipo de parámetros

C# selecciona el método dependiendo del tipo de parámetros. A continuación, una clase que implementa varios métodos con el mismo nombre, sólo que varía el tipo de parámetro.

D/009.cs

```
namespace Ejemplo;

class MiClase {
   private int valor;
   private string cadena;
   private double costo;

   public void UnMetodo(int valor, string cadena) {
      this.valor = valor;
      this.cadena = cadena;
      Console.WriteLine("Un método B");
   }

   public void UnMetodo(string cadena, int valor) {
      this.cadena = cadena;
      this.valor = valor;
      Console.WriteLine("Segundo método");
   }

   public void UnMetodo(double costo, int valor) {
      this.costo = costo;
      this.valor = valor;
      Console.WriteLine("Tercer método");
   }

   public void UnMetodo(string cadena, double costo) {
      this.cadena = cadena;
      this.costo = costo;
      Console.WriteLine("Cuarto método");
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto
      MiClase objetoA = new();
      objetoA.UnMetodo(48, "Rafael");
      objetoA.UnMetodo("Alberto", 26);
      objetoA.UnMetodo(1994.06, 48);
      objetoA.UnMetodo("Moreno Parra", 1683.29);
```

```
    }
}
```

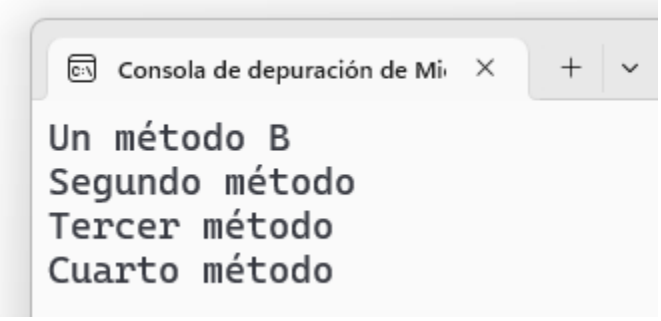

Ilustración 11: Sobrecarga de métodos

Constructores

En C# los constructores se escriben con "public Nombre_de_la_clase". Los constructores tienen estas características:

1. Deben tener el mismo nombre de la clase
2. Se ejecutan cuando el objeto es instanciado
3. No pueden retornar valores
4. Sólo ejecutan una sola vez (cuando el objeto se instancia)

Constructor sin parámetros

D/010.cs

```
namespace Ejemplo;

class MiClase {
   public MiClase() {
      Console.WriteLine("Constructor por defecto");
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto
      MiClase instancia = new();
   }
}
```

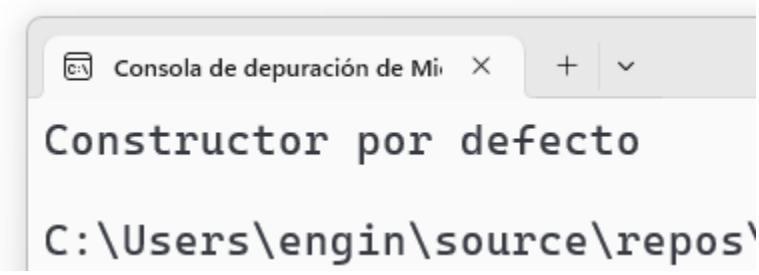

Ilustración 12: Constructor sin parámetros

Constructor con parámetros

Se envía los datos del objeto al instanciarlo.

D/011.cs

```
namespace Ejemplo;

//Esta es una clase propia con
//sus atributos y métodos (encapsulación)
class MiClase {
   //Un constructor
   public MiClase(int Numero, char Letra, string Cadena, double Valor) {
      //Se asigna así this.atributo = valor parámetro
      this.Numero = Numero;
      this.Letra = Letra;
      this.Cadena = Cadena;
      this.Valor = Valor;
   }

   //Otra forma de definir atributos con los getters y setters
   public int Numero { get; set; }
   public char Letra { get; set; }
   public string Cadena { get; set; }
   public double Valor { get; set; }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
      //llamando el constructor
      MiClase Mascotas = new(2016, 'T', "Tammy", 12.17);

      //Se imprimen los valores de ambas variables
      Console.WriteLine("Letra en Mascotas es: " + Mascotas.Letra);
      Console.WriteLine("Valor en Mascotas es: " + Mascotas.Valor);
   }
}
```

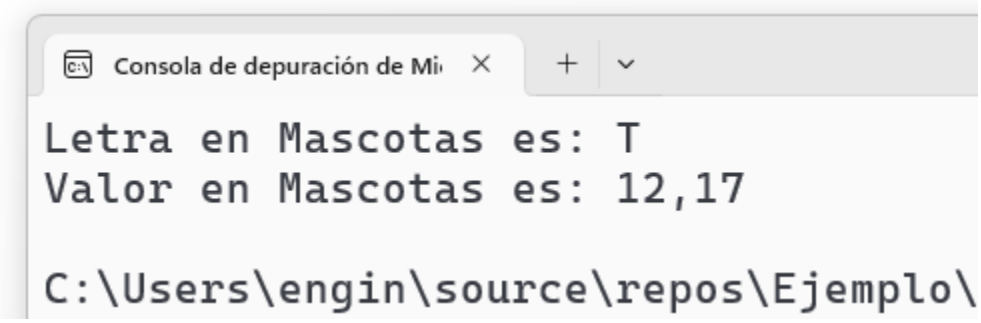

Ilustración 13: Constructor con parámetros

Sobrecarga de constructores

A continuación, una clase con varios constructores:

D/012.cs

```
namespace Ejemplo;

class MiClase {
   private int valor;
   private string cadena;
   private double costo;
   public MiClase() {
      Console.WriteLine("Constructor por defecto");
   }

   public MiClase(int valor) {
      this.valor = valor;
      this.cadena = "por defecto";
      this.costo = 0;
      Console.WriteLine("Constructor B");
   }

   public MiClase(string cadena, int valor) {
      this.cadena = cadena;
      this.valor = valor;
      this.costo = 0;
      Console.WriteLine("Tercer Constructor");
   }

   public MiClase(double costo, int valor) {
      this.cadena = "por defecto";
      this.costo = costo;
      this.valor = valor;
      Console.WriteLine("El cuarto constructor");
   }

   public MiClase(string cadena, double costo, int valor) {
      this.cadena = cadena;
      this.costo = costo;
      this.valor = valor;
      Console.WriteLine("Quinto constructor");
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
```

```
        //Instancia o crea un objeto
        MiClase objetoA = new();
        MiClase objetoB = new(48);
        MiClase objetoC = new(1972.06, 26);
        MiClase objetoD = new("Ramp", 48);
        MiClase objetoE = new("Moreno Parra", 1683.29, 29);
    }
}
```

Dependiendo del número de parámetros, se llama a un constructor o a otro

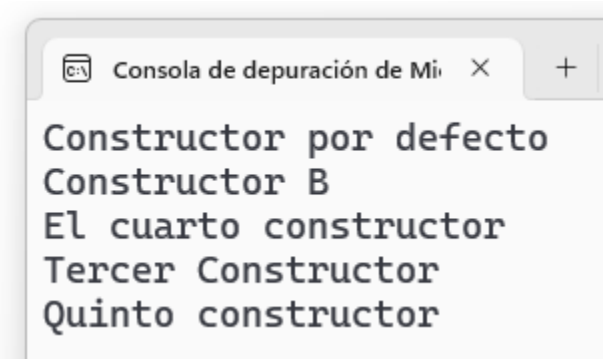

Ilustración 14: Sobrecarga de constructores

Usando el constructor para copiar objetos

La asignación de una variable objeto a otra variable objeto da como resultado que ambas variables apunten al mismo objeto en memoria, ese se le conoce como una copia superficial "Shallow Copy". ¿Cómo entonces generar una copia total del objeto, es decir, copiar los valores de los atributos también conocida como copia profunda o "Deep Copy"? Usualmente se busca el término "clonar el objeto", en el pasado, se usaba la instrucción ICloneable (considerada obsoleta o mejor no usarla: https://stackoverflow.com/questions/536349/why-no-icloneablet). A continuación, se muestra una técnica para hacer una copia profunda:

Se pone un método al que se le puede llamar CopiarObjeto() que retorna una nueva instancia de la clase y se le envía por el **constructor** los valores que tienen en los atributos.

D/013.cs

```
namespace Ejemplo;
//Esta es una clase propia con sus
//atributos y métodos (encapsulación)
internal class MiClase {
   //Un constructor
   public MiClase(int Num, char Car, string Cad, double Valor) {
      //Se asigna así this.atributo = valor parámetro
      this.Numero = Num;
      this.Letra = Car;
      this.Cadena = Cad;
      this.Valor = Valor;
   }

   //Método que permite copiar un objeto
   public MiClase CopiarObjeto() {
      MiClase copia = new MiClase(Numero, Letra, Cadena, Valor);
      return copia;
   }

   //Otra forma de definir atributos
   //con los getters y setters
   public int Numero { get; set; }
   public char Letra { get; set; }
   public string Cadena { get; set; }
   public double Valor { get; set; }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Instancia o crea un objeto de MiClase
```

```
        //llamando el constructor
        MiClase Mascotas = new(2016, 'T', "Tammy", 12.17);

        //Hace una copia de ese objeto
        MiClase UnaCopia = Mascotas.CopiarObjeto();

        //Se imprimen los valores de los dos objetos
        Console.WriteLine("Después de copiar");
        Console.WriteLine("Cadena en Mascotas: " + Mascotas.Cadena);
        Console.WriteLine("Cadena en UnaCopia: " + UnaCopia.Cadena);

        //Cambia el valor de cadena en UnaCopia
        UnaCopia.Cadena = "Krousky";

        //Imprime de nuevo los valores
        Console.WriteLine("\r\nDespués de cambiar la cadena");
        Console.WriteLine("Cadena en Mascotas: " + Mascotas.Cadena);
        Console.WriteLine("Cadena en UnaCopia: " + UnaCopia.Cadena);
    }
}
```

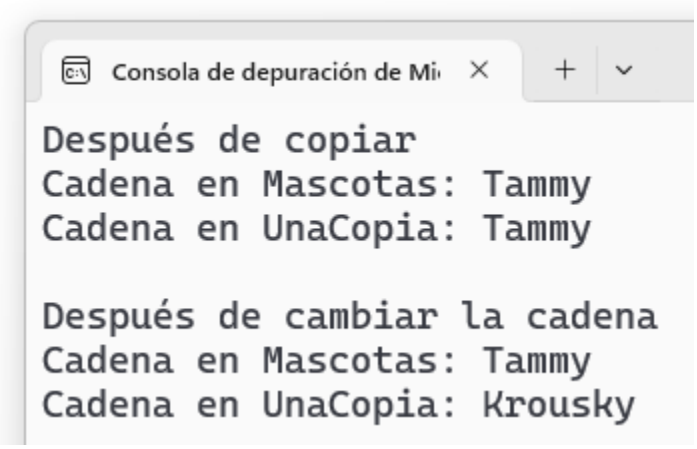

Ilustración 15: Usando el constructor para copiar objetos

Un constructor puede llamar a otros métodos

Al instanciar una clase, el constructor puede llamar a otros métodos.

D/014.cs

```
namespace Ejemplo;

internal class MiClase {
   //Constructor
   public MiClase() {
      //Llama a otros métodos
      MetodoA();
      MetodoB();
   }

   public void MetodoA() {
      Console.WriteLine("Ha llamado método A");
   }

   private void MetodoB() {
      Console.WriteLine("Ha llamado método B");
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      MiClase objeto = new();
   }
}
```

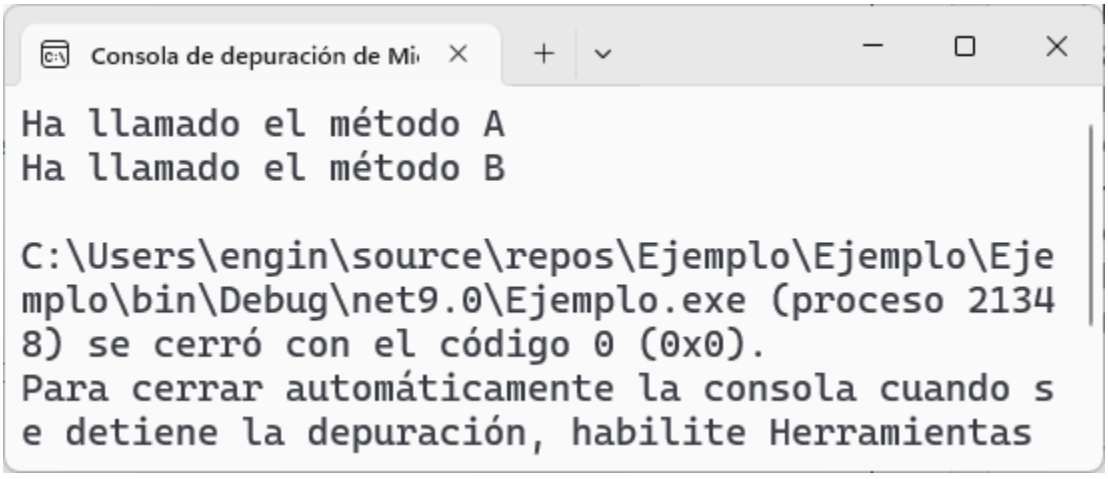

Ilustración 16: Un constructor puede llamar a otros métodos

Herencia

En C# se implementa así: *nombre clase:clase madre*

D/015.cs

```
namespace Ejemplo;

class Mascota {
   public string SerialChip { get; set; }
   public string Nombre { get; set; }
   public DateTime FechaNacimiento { get; set; }
   public char Sexo { get; set; } //Macho, Hembra

   //Nombre del propietario
   public string Propietario { get; set; }

   //Teléfono del propietario
   public string Telefono { get; set; }

   //Correo electrónico del propietario
   public string Correo { get; set; }
   public double Peso { get; set; }

   //0. Azul, 1. Verde, 2. Dorado, 3. Dispar
   public int ColorOjos { get; set; }

   public int EsperanzaVidaMinimo { get; set; }
   public int EsperanzaVidaMaximo { get; set; }

   //0. Baja, 1. Media, 2. Alta
   public int NecesidadAtencion { get; set; }

   public string Raza { get; set; }
}

class Gato : Mascota {
   public string PatronPelo { get; set; }
   public int TendenciaPerderPelo { get; set; }

   //Asociación de Criadores de Gatos
   public string ReconocimientoCFA { get; set; }

   //Asociación Americana de Criadores de Gatos
   public string ReconocimientoACFA { get; set; }

   //Fédération Internationale Féline
   public string ReconocimientoFIFe { get; set; }
```

```
   //Asociación Internacional de Gatos
   public string ReconocimientoTICA { get; set; }
}

class Perro : Mascota {
   //Real Sociedad Canina de España
   public string ReconocimientoRSCE { get; set; }

   //United Kennel Club
   public string ReconocimientoUKC { get; set; }

   //Crianza
   public string CriadoPara { get; set; }

   public double AlturaALaCruz { get; set; }

   //0. Ninguna, 1. Baja, 2. Moderada
   public int TendenciaBabear { get; set; }

   public int TendenciaRoncar { get; set; }
   public int TendenciaLadrar { get; set; }
   public int TendenciaExcavar { get; set; }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      Mascota objMascota = new();
      Gato objGato = new();
      Perro objPerro = new();

      //Da valores a la instancia de mascota
      objMascota.Correo = "enginelife@hotmail.com";
      objMascota.ColorOjos = 1;

      //Da valores a la instancia de gato
      objGato.Correo = "ramsoftware@gmail.com";
      objGato.Propietario = "Rafael Alberto Moreno Parra";
      objGato.Nombre = "Sally";
      objGato.Sexo = 'H';
      objGato.PatronPelo = "Tricolor";

      //Da valores a la instancia de perro
      objPerro.Raza = "Pastor Alemán";
      objPerro.Sexo = 'M';
      objPerro.Nombre = "Firuláis";
      objPerro.TendenciaLadrar = 1;
```

```
    }
}
```

Clases abstractas y herencia

Una clase abstracta solo permite heredar, no se puede instanciar. Si se intenta instanciar dará un mensaje de error en tiempo de compilación. La palabra reservada "abstract" es para definir clases abstractas.

D/016.cs

```
namespace Ejemplo;

//Esta clase solo puede heredar, no se puede instanciar
abstract class Mascota {
   public string SerialChip { get; set; }

   public string Nombre { get; set; }

   public DateTime FechaNacimiento { get; set; }

   //Macho, Hembra
   public char Sexo { get; set; }

   //Nombre del propietario
   public string Propietario { get; set; }

   //Teléfono del propietario
   public string Telefono { get; set; }

   //Correo electrónico del propietario
   public string Correo { get; set; }

   public double Peso { get; set; }

   //0. Azul, 1. Verde, 2. Dorado, 3. Dispar
   public int ColorOjos { get; set; }

   public int EsperanzaVidaMinimo { get; set; }

   public int EsperanzaVidaMaximo { get; set; }

   //0. Baja, 1. Media, 2. Alta
   public int NecesidadAtencion { get; set; }

   public string Raza { get; set; }
}

class Perro : Mascota {
   //Real Sociedad Canina de España
   public string ReconocimientoRSCE { get; set; }
```

```
   //United Kennel Club
   public string ReconocimientoUKC { get; set; }

   //Crianza
   public string CriadoPara { get; set; }

   public double AlturaALaCruz { get; set; }

   //0. Ninguna, 1. Baja, 2. Moderada
   public int TendenciaBabear { get; set; }

   public int TendenciaRoncar { get; set; }
   public int TendenciaLadrar { get; set; }
   public int TendenciaExcavar { get; set; }
}

class Gato : Mascota {
   public string PatronPelo { get; set; }
   public int TendenciaPerderPelo { get; set; }

   //Asociación de Criadores de Gatos
   public string ReconocimientoCFA { get; set; }

   //Asociación Americana de Criadores de Gatos
   public string ReconocimientoACFA { get; set; }

   //Fédération Internationale Féline
   public string ReconocimientoFIFe { get; set; }

   //Asociación Internacional de Gatos
   public string ReconocimientoTICA { get; set; }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      Mascota objMascota = new();
      Gato objGato = new();
      Perro objPerro = new();

      //Da valores a la instancia de mascota
      objMascota.Correo = "enginelife@hotmail.com";
      objMascota.ColorOjos = 1;

      //Da valores a la instancia de gato
      objGato.Correo = "ramsoftware@gmail.com";
      objGato.Propietario = "Rafael Alberto Moreno Parra";
      objGato.Nombre = "Sally";
```

```
        objGato.Sexo = 'H';
        objGato.PatronPelo = "Tricolor";

        //Da valores a la instancia de perro
        objPerro.Raza = "Pastor Alemán";
        objPerro.Sexo = 'M';
        objPerro.Nombre = "Firuláis";
        objPerro.TendenciaLadrar = 1;
    }
}
```

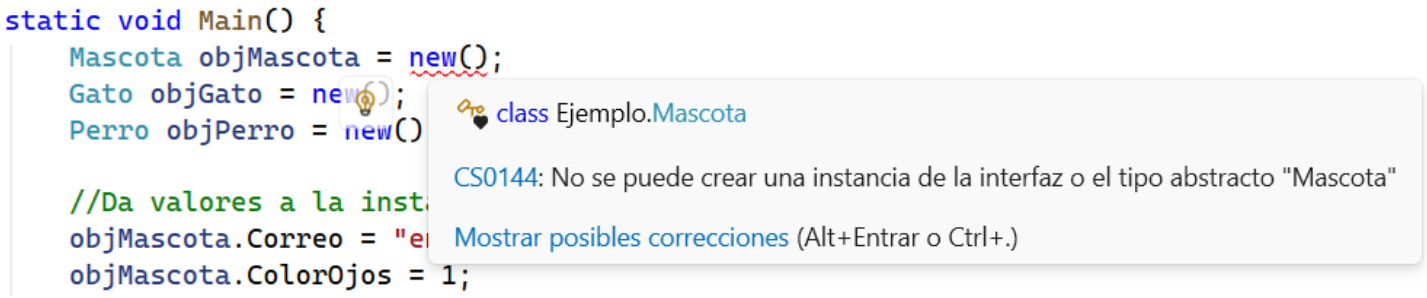

Ilustración 17: No se puede crear instancia

Nivel de protección en los métodos y atributos

Private

En la clase madre, los atributos o métodos con el atributo private no pueden ser usados por las clases hijas.

D/017.cs

```
namespace Ejemplo;

//Clase madre con atributos privados
class Mascota {
   private string Nombre { get; set; }

   //Macho, Hembra
   private char Sexo { get; set; }

   //Nombre del propietario
   private string Dueno { get; set; }
}

class Gato : Mascota {
   public string PatronPelo { get; set; }
   public int TendenciaPerderPelo { get; set; }
   public void DatosGato(string Nombre, char Sexo, string Dueno) {
      this.Nombre = Nombre;
      this.Sexo = Sexo;
      this.Dueno = Dueno;
   }
}

class Perro : Mascota {
   //Crianza
   public string CriadoPara { get; set; }
   public double AlturaALaCruz { get; set; }

   public void DatosPerro(string Nombre, char Sexo, string Dueno) {
      this.Nombre = Nombre;
      this.Sexo = Sexo;
      this.Dueno = Dueno;
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      Gato objGato = new();
```

```
        Perro objPerro = new();

        //Da valores a la instancia de gato
        objGato.DatosGato("Sally", 'H', "Rafael Moreno");

        //Da valores a la instancia de perro
        objPerro.DatosPerro("Kitty", 'H', "Chloe Perry");
    }
}
```

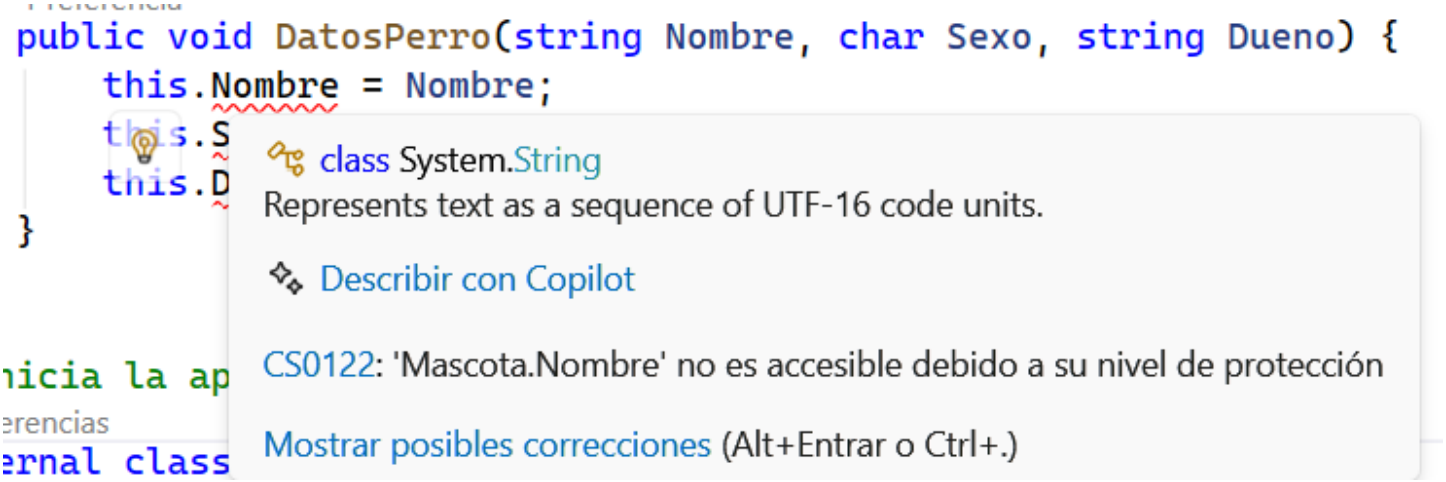

Ilustración 18: "Private" impide usar métodos o atributos en clases hijas

Protected

Si la clase madre tiene atributos o métodos con la palabra "protected", significa que esos atributos pueden ser accedidos por las clases hijas, pero no pueden ser accedidos por instancias.

D/018.cs

```
namespace Ejemplo;

//Clase madre con atributos privados
class Mascota {
   protected string Nombre { get; set; }

   //Macho, Hembra
   protected char Sexo { get; set; }

   //Nombre del propietario
   protected string Dueno { get; set; }
}

class Gato : Mascota {
   public string PatronPelo { get; set; }
   public int TendenciaPerderPelo { get; set; }
   public void DatosGato(string Nombre, char Sexo, string Dueno) {
      this.Nombre = Nombre;
      this.Sexo = Sexo;
      this.Dueno = Dueno;
   }
}

class Perro : Mascota {
   //Crianza
   public string CriadoPara { get; set; }
   public double AlturaALaCruz { get; set; }

   public void DatosPerro(string Nombre, char Sexo, string Dueno) {
      this.Nombre = Nombre;
      this.Sexo = Sexo;
      this.Dueno = Dueno;
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      Mascota objMascota = new();
      Gato objGato = new();
      Perro objPerro = new();
```

```
        //Da valores a la instancia de gato
        objGato.DatosGato("Sally", 'H', "Rafael Moreno");

        //Da valores a la instancia de perro
        objPerro.DatosPerro("Kitty", 'H', "Chloe Perry");

        //Intenta acceder a los métodos protegidos de Mascota
        objMascota.Nombre = "Milú";
    }
}
```

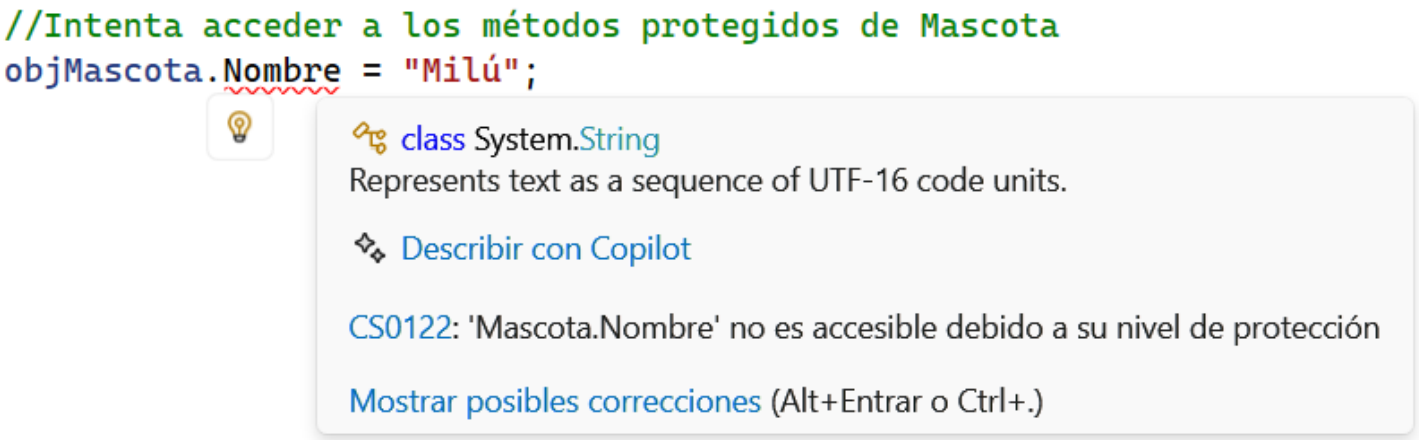

Ilustración 19: "Protected" permite uso en clases hijas, pero no en instancias

Public

Cuando los atributos o métodos tienen la palabra “public”, entonces pueden ser accedidos por las clases hijas y también por las instancias.

D/019.cs

```
namespace Ejemplo;

//Clase madre con atributos privados
class Mascota {
   public string Nombre { get; set; }

   //Macho, Hembra
   public char Sexo { get; set; }

   //Nombre del propietario
   public string Dueno { get; set; }
}

class Gato : Mascota {
   public string PatronPelo { get; set; }
   public int TendenciaPerderPelo { get; set; }
   public void DatosGato(string Nombre, char Sexo, string Dueno) {
      this.Nombre = Nombre;
      this.Sexo = Sexo;
      this.Dueno = Dueno;
   }
}

class Perro : Mascota {
   //Crianza
   public string CriadoPara { get; set; }
   public double AlturaALaCruz { get; set; }

   public void DatosPerro(string Nombre, char Sexo, string Dueno) {
      this.Nombre = Nombre;
      this.Sexo = Sexo;
      this.Dueno = Dueno;
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      Mascota objMascota = new();
      Gato objGato = new();
      Perro objPerro = new();
```

```
        //Da valores a la instancia de gato
        objGato.DatosGato("Sally", 'H', "Rafael Moreno");

        //Da valores a la instancia de perro
        objPerro.DatosPerro("Kitty", 'H', "Chloe Perry");

        //Intenta acceder a los métodos protegidos de Mascota
        objMascota.Nombre = "Milú";
    }
}
```

Herencia y métodos iguales en clase madre e hija

¿Qué sucede si un método están en la clase madre y se escribe un método con el mismo nombre en la clase hija y luego se instancia la clase hija y se ejecuta ese método? El término se le conoce como ocultamiento de método.

D/020.cs

```
namespace Ejemplo;

internal class Madre {
   public void Procedimiento() {
      Console.WriteLine("En la clase madre");
   }
}

internal class Hija : Madre {
   public new void Procedimiento() {
      Console.WriteLine("En la clase hija");
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      Hija objHija = new();
      objHija.Procedimiento();
   }
}
```

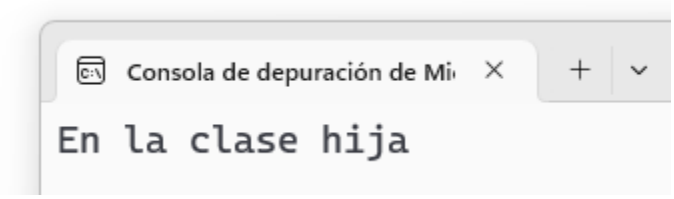

Ilustración 20: Herencia y métodos iguales en clase madre e hija

Polimorfismo

Obsérvese este código:

```
namespace Ejemplo;

internal class Empleado {
   public void Procedimiento() {
      Console.WriteLine("Procedimiento de empleado");
   }
}

internal class Operario : Empleado {
   public void Procedimiento() {
      Console.WriteLine("Operario, ejecuta");
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Todo operario es un empleado
      Empleado objeto = new Operario();
      objeto.Procedimiento();
   }
}
```

Al ejecutar sucede esto:

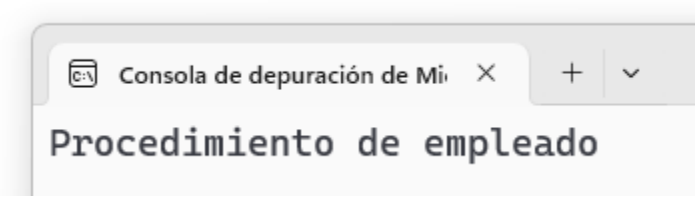

Ilustración 21: Polimorfismo clásico

En C#, una referencia de tipo base (clase Madre) puede apuntar a un objeto de tipo derivado (clase Hija). Esto se llama "upcasting", y es seguro porque todo Operario es un Empleado. Sin embargo, al llamar a "Procedimiento" se ejecuta el método de la clase Madre. ¿Cómo ejecutar el "Procedimiento" de la clase Hija? Usando "virtual" y "override"

Polimorfismo, virtual, override

Ahora este código:

```
namespace Ejemplo;

internal class Empleado {
   public virtual void Procedimiento() {
      Console.WriteLine("Procedimiento de empleado");
   }
}

internal class Operario : Empleado {
   public override void Procedimiento() {
      Console.WriteLine("Operario, ejecuta");
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Todo operario es un empleado
      Empleado objeto = new Operario();
      objeto.Procedimiento();
   }
}
```

Al ejecutar sucede esto:

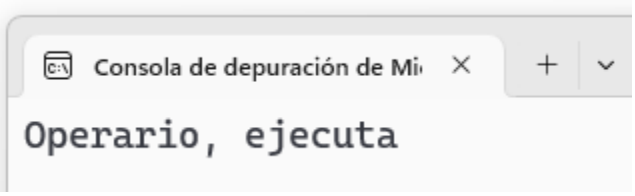

Ilustración 22: virtual, override

Este patrón permite **polimorfismo**, es decir, que se pueda tratar diferentes tipos derivados como si fueran del tipo base, y aun así obtener comportamientos específicos.

```
namespace Ejemplo;

internal class Empleado {
   public virtual void Procedimiento() {
      Console.WriteLine("Procedimiento de empleado");
   }
}

internal class Operario : Empleado {
   public override void Procedimiento() {
      Console.WriteLine("Ejecuta método de Operario");
   }
}

internal class Ingeniero : Empleado {
   public override void Procedimiento() {
      Console.WriteLine("Ejecuta método de Ingeniero");
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Todo operario es un empleado
      Empleado objetoA = new Operario();
      Empleado objetoB = new Ingeniero();
      objetoA.Procedimiento();
      objetoB.Procedimiento();
   }
}
```

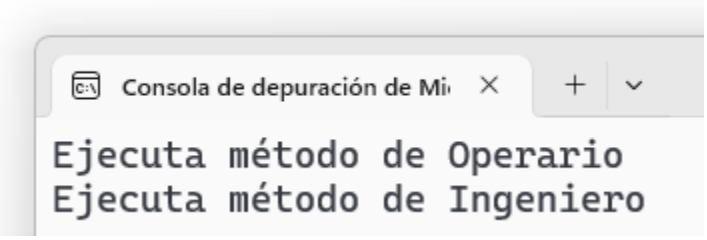

Ilustración 23: Polimorfismo

Herencia y constructores

¿Qué sucede si la clase abuela o madre o hija tienen todas constructores? ¿Se ejecutan todos? ¿En qué orden?

D/021.cs

```
namespace Ejemplo;

class Abuela {
   //Constructor
   public Abuela() {
      Console.WriteLine("Constructor de la clase abuela");
   }

   //Método
   public void Mostrar() {
      Console.WriteLine("Mostrar en Abuela");
   }
}

class Madre : Abuela {
   //Constructor
   public Madre() {
      Console.WriteLine("Constructor de la clase madre");
   }

   public new void Mostrar() {
      Console.WriteLine("Mostrar en Madre");
   }
}

class Hija : Madre {
   //Constructor
   public Hija() {
      Console.WriteLine("Constructor de la clase hija");
   }
   public new void Mostrar() {
      Console.WriteLine("Mostrar en Hija");
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia a la hija
      Hija objHija = new();
```

```
        //Ejecuta método
        objHija.Mostrar();
    }
}
```

La ejecución del programa hace lo siguiente:

1. Ejecuta el constructor de la clase abuela
2. Ejecuta el constructor de la clase madre
3. Ejecuta el constructor de la clase hija
4. Si un método, diferente al constructor, tiene el mismo nombre en las clases abuela, madre e hija. Al ser ejecutado por la instancia de la clase hija, sólo ejecutará el método de la clase hija.

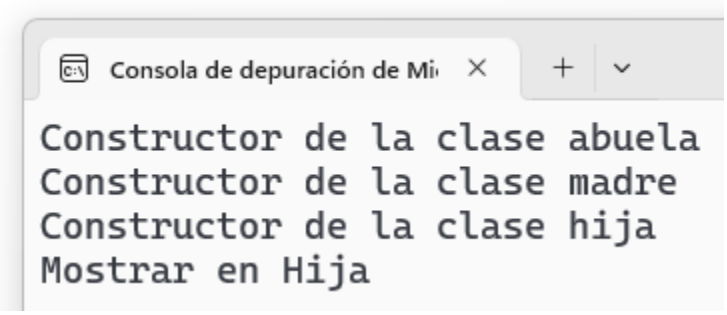

Ilustración 24: Constructores y herencia

Llamando a métodos de clases madres

Desde el método de la clase hija se hace uso de la instrucción "base" y así se llama al método de la clase madre

D/022.cs

```
namespace Ejemplo;

internal class Abuela {
   //Constructor
   public Abuela() {
      Console.WriteLine("Constructor de la clase abuela");
   }

   //Método
   public void Mostrar() {
      Console.WriteLine("Mostrar en Abuela");
   }
}

internal class Madre : Abuela {
   //Constructor
   public Madre() {
      Console.WriteLine("Constructor de la clase madre");
   }

   public new void Mostrar() {
      base.Mostrar(); //Llama al método de la clase abuela
      Console.WriteLine("Mostrar en Madre");
   }
}

internal class Hija : Madre {
   //Constructor
   public Hija() {
      Console.WriteLine("Constructor de la clase hija");
   }

   public new void Mostrar() {
      base.Mostrar(); //Llama al método de la clase madre
      Console.WriteLine("Mostrar en Hija");
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
```

```
        //Instancia a la hija
        Hija objHija = new();

        //Ejecuta método
        objHija.Mostrar();
    }
}
```

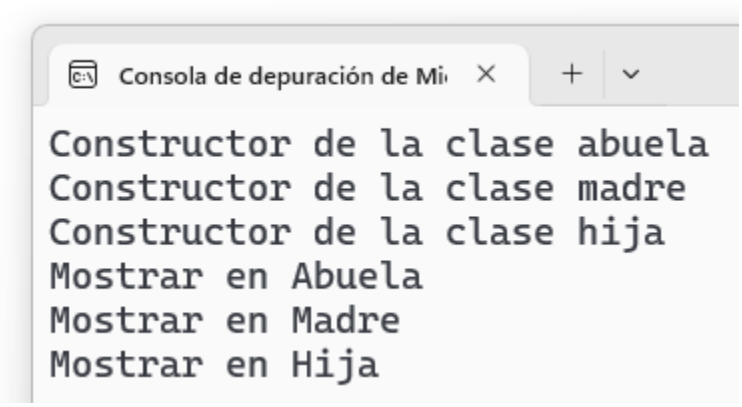

Ilustración 25: Llamando a métodos de clases madres

Evitar la herencia

Con la palabra reservada “sealed” se evita que de esa clase se pueda heredar. Si se intenta habrá un error en tiempo de compilación.

D/023.cs

```
namespace Ejemplo;

sealed class Madre {
   public void Aviso() {
      Console.WriteLine("Método en clase madre");
   }
}

class Hija : Madre {
   public void Mensaje() {
      Console.WriteLine("Método en clase hija");
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      Hija objHija = new();
      objHija.Mensaje();
   }
}
```

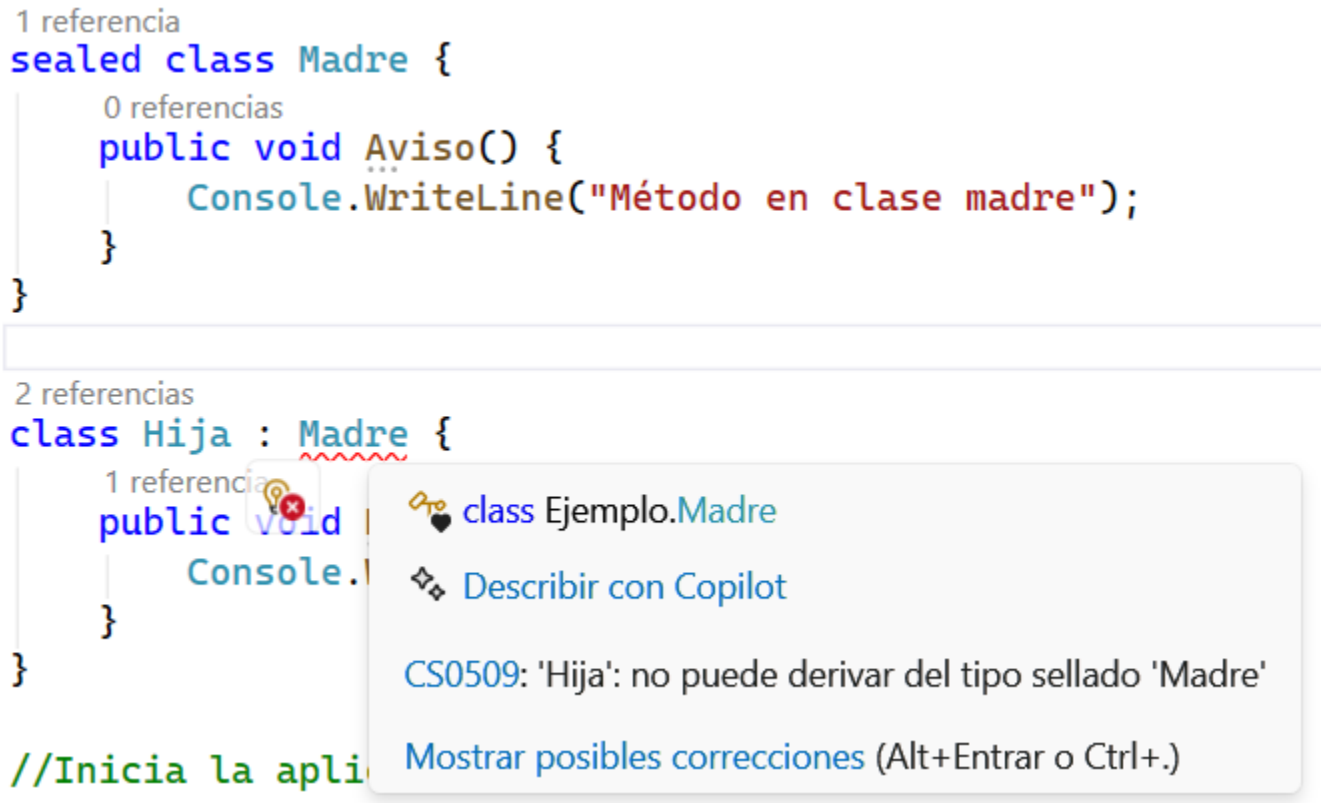

Ilustración 26: Evitar la herencia

Clases estáticas

Una clase estática no requiere instanciarse para ser usada.

D/024.cs

```
namespace Ejemplo;

//Clase estática, no necesita instanciarse
static class Geometria {
   //Todos los métodos deben ser static
   public static double AreaTriangulo(double baseT, double alturaT) {
      return baseT * alturaT / 2;
   }

   public static double AreaTriangulo(double ladoA, double ladoB,
                              double ladoC) {
      double s = (ladoA + ladoB + ladoC) / 2;
      return Math.Sqrt(s * (s - ladoA) * (s - ladoB) * (s - ladoC));
   }

   public static double AreaCirculo(double radio) {
      return Math.PI * radio * radio;
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Hace uso de la clase sin instanciarla
      double unRadio = 7;
      double AreaUnCirculo = Geometria.AreaCirculo(unRadio);
      Console.WriteLine("Área círculo es: " + AreaUnCirculo);

      double AreaTri = Geometria.AreaTriangulo(3, 4, 5);
      Console.WriteLine("Área triángulo es: " + AreaTri);
   }
}
```

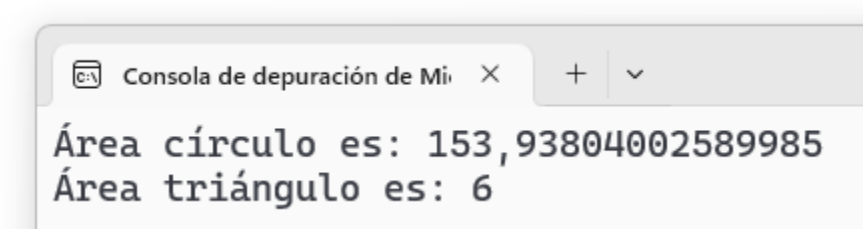

Ilustración 27: Clases estáticas

Métodos estáticos

Una clase tradicional puede tener métodos estáticos y estos métodos pueden ser accedidos sin necesidad de instanciar la clase, los otros métodos no estáticos si requieren que se instancie la clase.

D/025.cs

```
namespace Ejemplo;

//Clase con métodos estáticos
internal class Geometria {
   //Estos métodos estáticos pueden ser usados sin instanciar la clase
   public static double AreaTriangulo(double baseT, double alturaT) {
      return baseT * alturaT / 2;
   }

   public static double AreaTriangulo(double ladoA, double ladoB,
                              double ladoC) {
      double s = (ladoA + ladoB + ladoC) / 2;
      return Math.Sqrt(s * (s - ladoA) * (s - ladoB) * (s - ladoC));
   }

   public static double AreaCirculo(double radio) {
      return Math.PI * radio * radio;
   }

   //Este método requiere instanciar la clase
   public double VolumenEsfera(double radio) {
      return 4 / 3 * Math.PI * Math.Pow(radio, 3);
   }
}

internal class Program {
   static void Main() {
      double unRadio = 7;
      double AreaUnCirculo = Geometria.AreaCirculo(unRadio);
      Console.WriteLine("Área círculo es: " + AreaUnCirculo);

      double AreaTri = Geometria.AreaTriangulo(3, 4, 5);
      Console.WriteLine("Área triángulo es: " + AreaTri);

      //Instancio la clase
      Geometria objGeometria = new();
      double Esfera = objGeometria.VolumenEsfera(7);
      Console.WriteLine("Volumen Esfera: " + Esfera);
   }
}
```

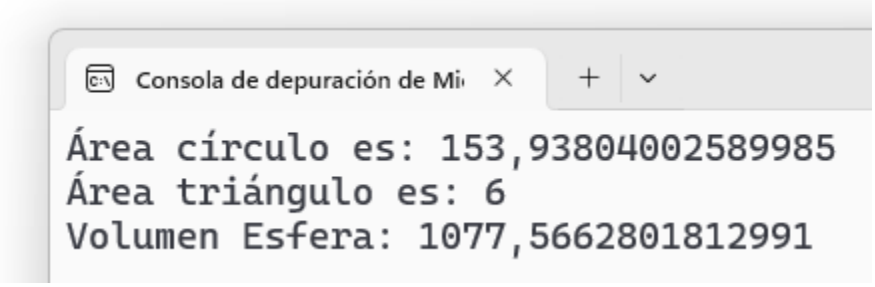

Ilustración 28: Métodos estáticos

Constructor static

Para inicializar los atributos static de una clase, se hace uso de los constructores static. Una clase no static puede tener un constructor static y un constructor normal, el primero se ejecuta siempre al usarse la clase o instanciarse el objeto, en cambio, el constructor normal sólo se ejecuta al instanciarse la clase.

D/026.cs

```
namespace Ejemplo;

//Clase con métodos estáticos
class Geometria {
   public static int valorEntero;
   public static double valorReal;
   public static string unaCadena;

   //Constructor static (para inicializar atributos static)
   static Geometria() {
      valorEntero = 7;
      valorReal = 16.832;
      unaCadena = "Rafael";
      Console.WriteLine("Se ha ejecutado el constructor static");
   }

   //Constructor de clase
   public Geometria() {
      Console.WriteLine("Ejecuta el constructor de la clase");
   }

   //Este método estático puede ser usado sin instanciar la clase
   public static double AreaCirculo(double radio) {
      return Math.PI * radio * radio;
   }

   //Este método requiere instanciar la clase
   public double VolumenEsfera(double radio) {
      return 4 / 3 * Math.PI * Math.Pow(radio, 3);
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Acediendo a métodos estáticos
      double AreaUnCirculo = Geometria.AreaCirculo(7);
      Console.WriteLine("Área círculo es: " + AreaUnCirculo);
```

```
        //Accediendo a atributos estáticos
        Console.WriteLine("Cadena es: " + Geometria.unaCadena);
        Console.WriteLine("Valor entero es: " + Geometria.valorEntero);
        Console.WriteLine("Valor real es: " + Geometria.valorReal);

        //Se instancia la clase
        Geometria objGeometria = new Geometria();
        double Esfera = objGeometria.VolumenEsfera(7);
        Console.WriteLine("Volumen Esfera: " + Esfera);
    }
}
```

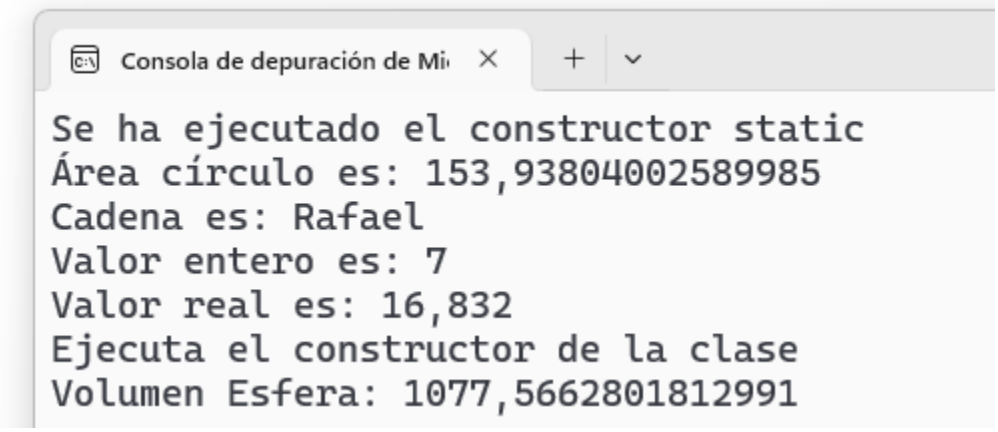

Ilustración 29: Constructor static

Cuidado con el constructor static

El constructor static se ejecuta en el momento que es usada la clase por primera vez (por ejemplo, cuando se instancia), no se vuelve a usar más.

D/027.cs

```
namespace Ejemplo;

//Clase con métodos estáticos
internal class Geometria {
   public static int valorEntero;
   public static double valorReal;
   public static string unaCadena;

   //Constructor static (para inicializar atributos static)
   static Geometria() {
      valorEntero = 7;
      valorReal = 16.832;
      unaCadena = "Rafael";
      Console.WriteLine("Se ha ejecutado el constructor static");
   }

   //Constructor de clase
   public Geometria() {
      Console.WriteLine("Ejecuta el constructor de la clase");
   }

   //Este método estático puede ser usado sin instanciar la clase
   public static double AreaCirculo(double radio) {
      return Math.PI * radio * radio;
   }

   //Este método requiere instanciar la clase
   public double VolumenEsfera(double radio) {
      return 4 / 3 * Math.PI * Math.Pow(radio, 3);
   }
}

//Inicia la aplicación aquí
internal class Program {
   static void Main() {
      //Se instancia la clase la primera vez
      Geometria objGeometria = new();
      double Esfera = objGeometria.VolumenEsfera(7);
      Console.WriteLine("Volumen Esfera A: " + Esfera);
```

```
        //Se instancia la clase la segunda vez
        Geometria objOtro = new();
        double OtraEsfera = objOtro.VolumenEsfera(7);
        Console.WriteLine("Volumen Esfera B: " + OtraEsfera);
    }
}
```

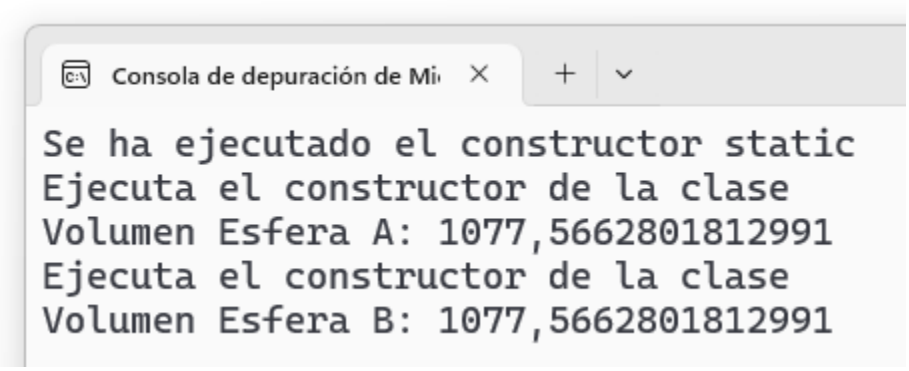

Ilustración 30: Cuidado con el constructor static

Interface

Con la palabra reservada “interface” se crean las definiciones de métodos y propiedades que deben ser escritos en las clases que implementen esa “interface”. El estándar solicita que el nombre de las “interface” empiece por I. Nota: Las interfaces no implementan ningún código, obligan a la clase que hereda a implementar los métodos.

D/028.cs

```
namespace Ejemplo;

//Declara una interface (el estándar dice
//que debe empezar con la letra "I")
interface IMetodosRequeridos {
   //Métodos requeridos en las clases que
   //implementen esta interface
   double Area();
   double Perimetro();
}

//Esta clase debe implementar lo que dice la interface
class Circulo : IMetodosRequeridos {
   public double Radio { get; set; }

   public Circulo(double Radio) {
      this.Radio = Radio;
   }

   //Implementa los métodos señalados por la interface
   public double Area() {
      return Math.PI * Radio * Radio;
   }

   public double Perimetro() {
      return 2 * Math.PI * Radio;
   }
}

//Esta clase debe implementar lo que dice la interface
class Cuadrado : IMetodosRequeridos {
   public double Lado { get; set; }

   public Cuadrado(double Lado) {
      this.Lado = Lado;
   }

   //Implementa los métodos señalados por la interface
```

```
   public double Area() {
      return Lado * Lado;
   }

   public double Perimetro() {
      return 4 * Lado;
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia las clases
      Cuadrado cuad = new(5);
      Circulo circ = new(5);

      //Imprime los valores
      Console.WriteLine("Área círculo: " + circ.Area());
      Console.WriteLine("Área cuadrado: " + cuad.Area());
      Console.WriteLine("Perímetro círculo: " + circ.Perimetro());
      Console.WriteLine("Perímetro cuadrado: " + cuad.Perimetro());
   }
}
```

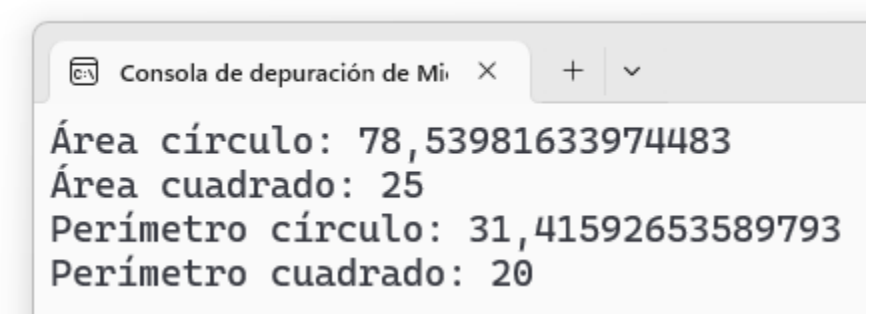

Ilustración 31: Interface

Interface múltiple

Una clase puede "heredar" de dos o más interfaces.

D/029.cs

```
namespace Ejemplo;

//Declara una interface (el estándar
//dice que debe empezar con la letra "I")
interface ICalculos {
   //Métodos requeridos en las clases
   //que implementen esta interface
   double Area();
   double Perimetro();

}

//Otra interface para obligar a mostrar los resultados
interface IMuestra {
   void VerArea();
   void VerPerimetro();
}

//Implementa de varios Interface
class Circulo : ICalculos, IMuestra {
   public double Radio { get; set; }

   public Circulo(double Radio) {
      this.Radio = Radio;
   }

   //Implementa los métodos señalados por la interface
   public double Area() {
      return Math.PI * Radio * Radio;
   }

   public double Perimetro() {
      return 2 * Math.PI * Radio;
   }

   public void VerArea() {
      Console.WriteLine("Área círculo: " + Area());
   }

   public void VerPerimetro() {
      Console.WriteLine("Perímetro círculo: " + Perimetro());
   }
```

```
}

class Cuadrado : ICalculos, IMuestra {
   public double Lado { get; set; }

   public Cuadrado(double Lado) {
      this.Lado = Lado;
   }

   //Implementa los métodos señalados por la interface
   public double Area() {
      return Lado * Lado;
   }

   public double Perimetro() {
      return 4 * Lado;
   }

   public void VerArea() {
      Console.WriteLine("Área cuadrado: " + Area());
   }

   public void VerPerimetro() {
      Console.WriteLine("Perímetro cuadrado: " + Perimetro());
   }
}

//Inicia la aplicación aquí
class Program {
   static void Main() {
      //Instancia las clases
      Cuadrado objCuadrado = new(5);
      Circulo objCirculo = new(5);

      //Imprime los valores
      objCuadrado.VerArea();
      objCuadrado.VerPerimetro();

      objCirculo.VerArea();
      objCirculo.VerPerimetro();
   }
}
```

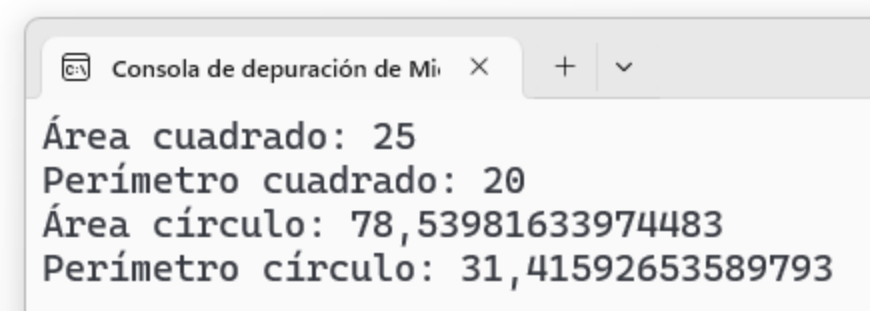

Ilustración 32: Interface múltiple

Interface (coinciden métodos)

Si una clase implementa dos o más interfaces que tienen métodos con el mismo nombre y firma, se puede generar una ambigüedad.

```
namespace Ejemplo;

internal interface InterfazA {
   public void Imprime();
}

internal interface InterfazB {
   public void Imprime();
}

internal class MiClase : InterfazA, InterfazB {
   public void Imprime() {
      Console.WriteLine("Esto es una prueba");
   }
}

internal class Program {
   static void Main(string[] args) {
      MiClase objeto = new();
      objeto.Imprime();
   }
}
```

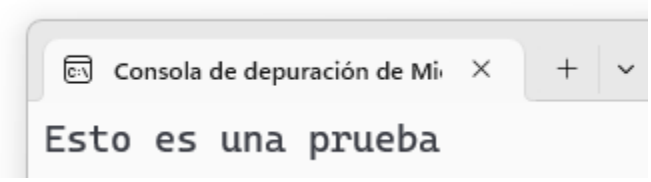

Ilustración 33: Dos "interface" con el mismo método

Implementación explícita para resolver ambigüedad

Para resolver la ambigüedad en el caso de "heredar" dos interfaces con el mismo método, se hace uso de la implementación explícita:

```
namespace Ejemplo;

internal interface InterfazA {
   public void Imprime();
}

internal interface InterfazB {
   public void Imprime();
}

internal class MiClase : InterfazA, InterfazB {
   void InterfazA.Imprime() {
      Console.WriteLine("Imprimiendo desde A");
   }

   void InterfazB.Imprime() {
      Console.WriteLine("Desde B imprime");
   }
}

internal class Program {
   static void Main() {
      InterfazA objA = new MiClase();
      InterfazB objB = new MiClase();

      objA.Imprime();
      objB.Imprime();
   }
}
```

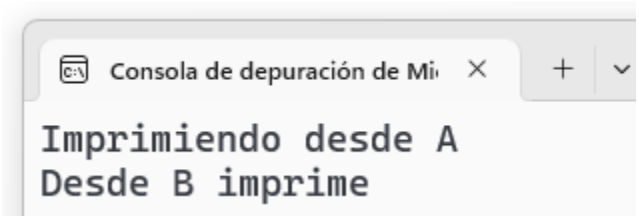

Ilustración 34: Implementación explícita en interfaces

Herencia múltiple con Interface

Sólo se puede heredar de **una clase** pero se puede "heredar" de varios Interface.

D/030.cs

```
namespace Ejemplo;

interface IMetodos {
   void MetodoA();
   void MetodoB();
}

interface IProcedimientos {
   void ProcedimientoA();
   void ProcedimientoB();
}

class Madre {
   public void Aviso() {
      Console.WriteLine("Método de clase madre");
   }
}

//Hereda de Madre e implementa de IMetodos e IProcedimientos
class Hija : Madre, IMetodos, IProcedimientos {
   public void Mensaje() {
      Console.WriteLine("En clase hija");
   }

   public void MetodoA() {
      Console.WriteLine("En MetodoA");
   }

   public void MetodoB() {
      Console.WriteLine("En MetodoB");
   }

   public void ProcedimientoA() {
      Console.WriteLine("En ProcedimientoA");
   }

   public void ProcedimientoB() {
      Console.WriteLine("En ProcedimientoB");
   }
}

//Inicia la aplicación aquí
```

```
class Program {
   static void Main() {
      Hija objHija = new();
      objHija.Aviso();
      objHija.Mensaje();
      objHija.MetodoA();
      objHija.ProcedimientoA();
   }
}
```

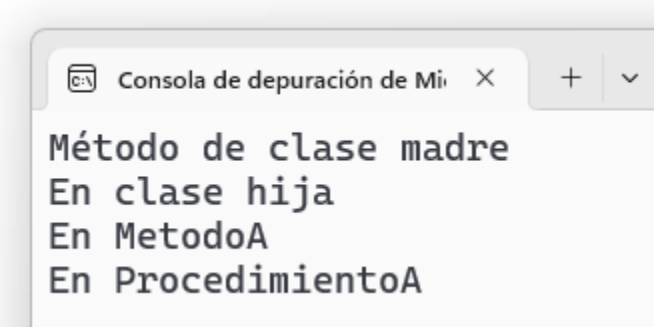

Ilustración 35: Herencia e Interface

Enums

Es una “clase especial” para guardar constantes

D/031.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una "clase especial" para almacenar constantes
   enum Meses {
      Enero, //0
      Febrero, //1
      Marzo, //2
      Abril, //3
      Mayo,  //4
      Junio,  //5
      Julio,  //6
      Agosto,  //7
      Septiembre, //8
      Octubre,  //9
      Noviembre,  //10
      Diciembre  //11
   }

   static void Main() {
      Meses unMes = Meses.Junio;
      Console.WriteLine(unMes);
      Console.WriteLine((int)unMes);
   }
}
```

Por defecto, las constantes de un enum comienzan a enumerarse desde cero, pero eso pueden cambiarse.

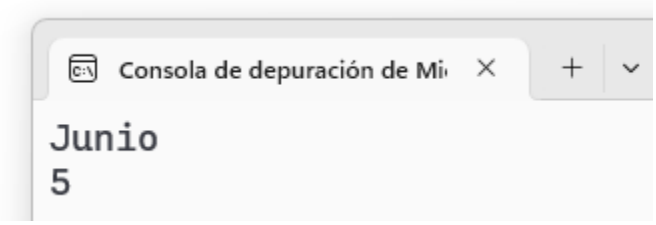

Ilustración 36: Enums

Cambiando los valores de las constantes en enums

Ejemplo 1

El método para que cada constante tenga su propio valor es constante = valor

D/032.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una "clase especial" para almacenar constantes
   enum Meses {
      Enero = 1,
      Febrero = 2,
      Marzo = 3,
      Abril = 4,
      Mayo = 5,
      Junio = 6,
      Julio = 7,
      Agosto = 8,
      Septiembre = 9,
      Octubre = 10,
      Noviembre = 11,
      Diciembre = 12
   }

   static void Main() {
      Meses unMes = Meses.Junio;
      Console.WriteLine(unMes);
      Console.WriteLine((int)unMes);
   }
}
```

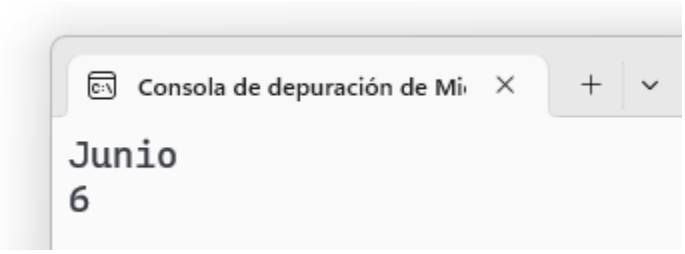

Ilustración 37: Cambiando los valores de las constantes en enums

Ejemplo 2

D/033.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una "clase especial" para almacenar constantes
   enum Valores {
      valorA = 89,
      valorB = 12,
      valorC = 47,
      valorD = 63
   }

   static void Main() {
      Valores unosValores = Valores.valorC;
      Console.WriteLine(unosValores);
      Console.WriteLine((int)unosValores);
   }
}
```

Nota 1: Los valores sólo pueden ser de tipo entero, long, byte, sbyte, short, ushort, uint, long, ulong

Nota 2: Los valores no pueden ser reales o cadenas o caracteres

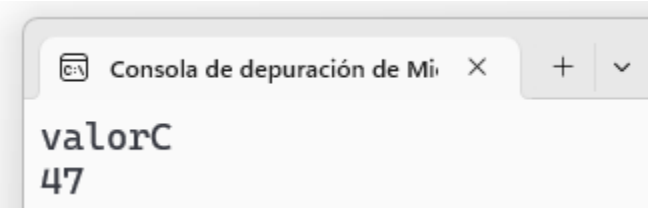

Ilustración 38: Cambiando los valores de las constantes en enums

Structs

C# trae las estructuras, que tienen un gran parecido a las clases, a tal punto que podrían ser su reemplazo en varias ocasiones porque tienen características interesantes como poder copiar el contenido de un struct en otro con el simple operador de asignación (algo que requiere un tratamiento si se usaran clases). Los structs tienen sus diferencias con respecto a las clases: sólo pueden definir constructores propios (con parámetros de entrada), pero no se puede crear un constructor simple, no pueden heredar, ni se puede heredar de estos, tampoco se pueden hacer structs abstractos.

D/034.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una estructura
   struct Valores {
      public int valorA;
      public char valorB;
      public double valorC;
      public string valorD;
   }

   static void Main() {
      //Crea una variable de tipo struct
      Valores unosValores;
      unosValores.valorA = 13;
      unosValores.valorB = 'R';
      unosValores.valorC = 16.832;
      unosValores.valorD = "Milú";

      //Puede imprimir esos valores
      Console.WriteLine(unosValores.valorA);
      Console.WriteLine(unosValores.valorB);
      Console.WriteLine(unosValores.valorC);
      Console.WriteLine(unosValores.valorD);
   }
}
```

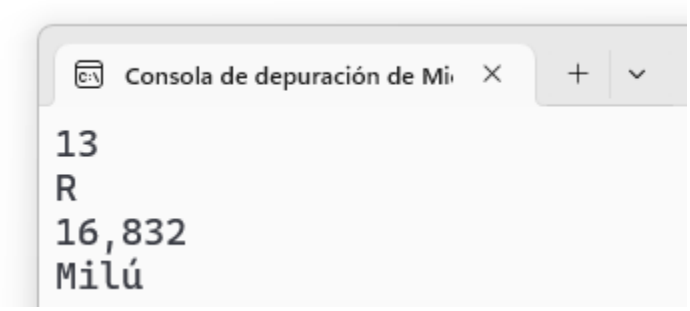

Ilustración 39: Structs

Un struct se puede copiar fácilmente

Los valores de una variable struct se pueden copiar en otra variable struct de la misma estructura usando el operador de asignación (=). Si se modifica el original, no afecta a la copia.

D/035.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una estructura
   struct Valores {
      public int valorA;
      public char valorB;
      public double valorC;
      public string valorD;
   }

   static void Main() {
      //Crea una variable de tipo struct
      Valores unosValores;
      unosValores.valorA = 13;
      unosValores.valorB = 'R';
      unosValores.valorC = 16.832;
      unosValores.valorD = "Milú";

      //Crea una segunda variable y le asigna la primera
      //creando una copia
      Valores otro;
      otro = unosValores;

      //Puede imprimir esos valores
      Console.WriteLine("Valores copiados");
      Console.WriteLine(otro.valorA);
      Console.WriteLine(otro.valorB);
      Console.WriteLine(otro.valorC);
      Console.WriteLine(otro.valorD);

      //Modifica la original
      unosValores.valorA = -9876;

      //Imprime la copia
      Console.WriteLine("\nDespués de modificar el original");
      Console.WriteLine(otro.valorA);
   }
```

```
}
```

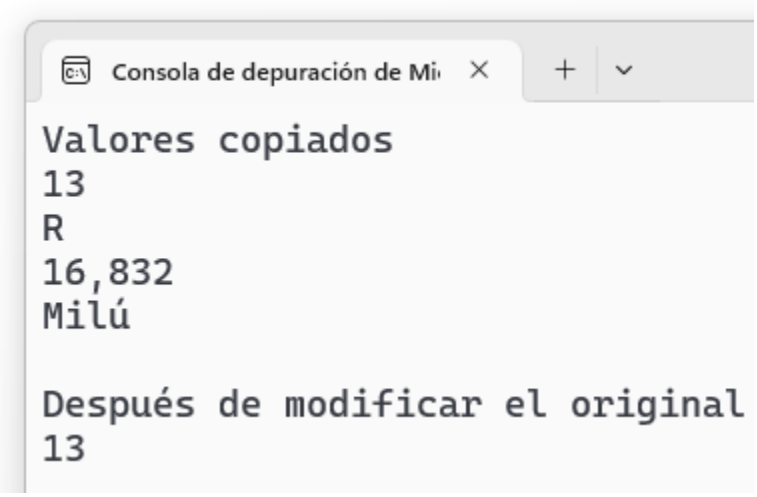

Ilustración 40: Un struct se puede copiar fácilmente

Métodos en un struct

Un struct puede tener métodos.

D/036.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una estructura
   struct Valores {
      private int valorA;
      private char valorB;
      private double valorC;
      private string valorD;

      public void Asigna(int A, char B, double C, string D) {
         this.valorA = A;
         this.valorB = B;
         this.valorC = C;
         this.valorD = D;
      }

      public void ImprimeValores() {
         Console.WriteLine(valorA);
         Console.WriteLine(valorB);
         Console.WriteLine(valorC);
         Console.WriteLine(valorD);
      }

      public double RetornaValor(double numero) {
         return valorA * valorC + numero;
      }
   }

   static void Main() {
      //Crea una variable de tipo struct y
      //la inicializa por defecto
      Valores unosValores = default;

      //Llama a los métodos del struct
      unosValores.Asigna(13, 'R', 16.832, "Milú");
      unosValores.ImprimeValores();

      //Y a la función del struct
      Console.WriteLine(unosValores.RetornaValor(5));
```

```
    }
}
```

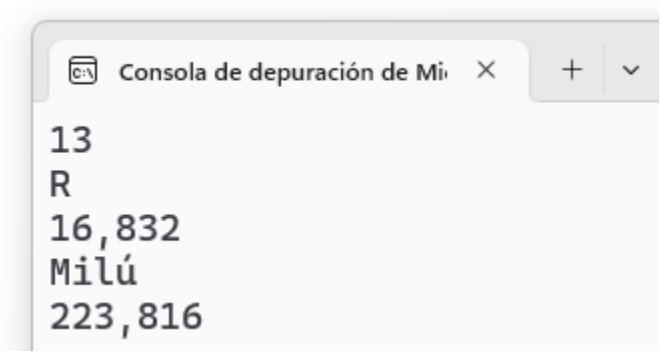

Ilustración 41: Métodos en un struct

Structs y constructores

Un struct puede tener un constructor que tenga parámetros (no se puede generar un constructor sin parámetros). Ejemplo:

D/037.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {

   //Una estructura
   struct Valores {
      private int valorA;
      private char valorB;
      private double valorC;
      private string valorD;

      public Valores(int valA, char valB, double valC, string valD) {
         this.valorA = valA;
         this.valorB = valB;
         this.valorC = valC;
         this.valorD = valD;
      }

      public void ImprimeValores() {
         Console.WriteLine(valorA);
         Console.WriteLine(valorB);
         Console.WriteLine(valorC);
         Console.WriteLine(valorD);
      }
   }

   static void Main() {
      //Crea una variable de tipo struct y
      //la inicializa con un constructor
      Valores unosValores = new Valores(13, 'R', 16.832, "Milú");
      unosValores.ImprimeValores();
   }
}
```

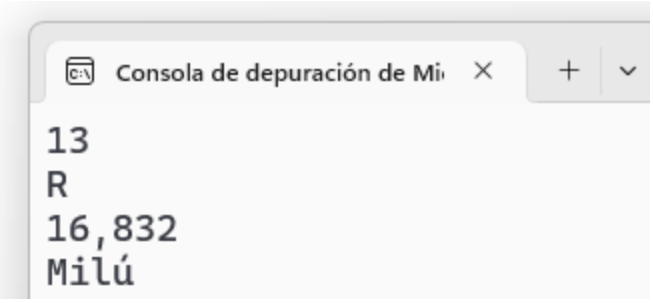

Ilustración 42: Structs y constructores

Clases parciales

C# puede tener partes de una misma clase en diferentes archivos. Útil si dos o más programadores quieren trabajar en la misma clase al tiempo. En Visual Studio, se hace para separar la parte lógica de un formulario, de la parte de diseño GUI de ese mismo formulario.

D/038a.cs

```
namespace Ejemplo;

partial class MiClase {
   public MiClase(int valA, double valB, char valC, string valD) {
      ValorA = valA;
      ValorB = valB;
      ValorC = valC;
      ValorD = valD;
   }

   public void Imprime() {
      Console.WriteLine("Valores");
      Console.WriteLine(ValorA);
      Console.WriteLine(ValorB);
      Console.WriteLine(ValorC);
      Console.WriteLine(ValorD);
   }
}

//Inicia la aplicación aquí
class Program {
   public static void Main() {
      MiClase objClase = new MiClase(2010, 7.15, 'S', "Sally");
      objClase.Imprime();
   }
}
```

D/038b.cs

```
namespace Ejemplo;

partial class MiClase {
   public int ValorA { get; set; }
   public double ValorB { get; set; }
   public char ValorC { get; set; }
   public string ValorD { get; set; }
}
```

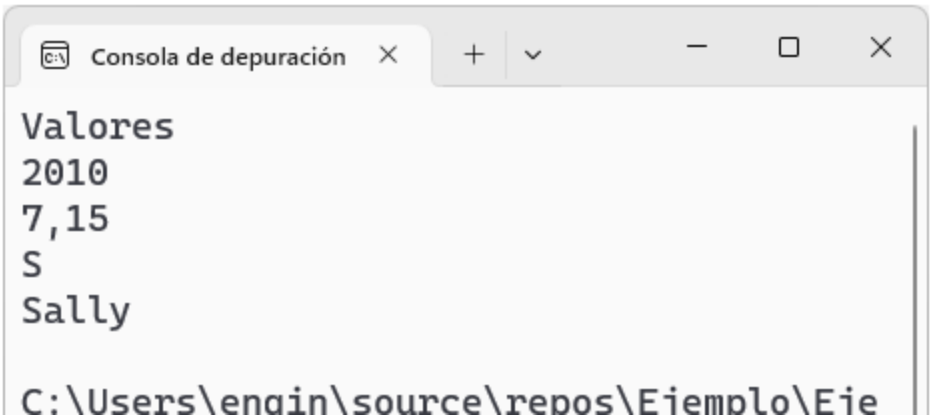

Ilustración 43: Clases parciales

Destructores

Por lo general, C# se encarga automáticamente de liberar la memoria de los objetos que ya no pueden ser usados (por ejemplo, cuando se crea un objeto dentro de una función con una variable local y luego termina la función). Aun así, en raras ocasiones, es necesario tener un método que se ejecuta cuando el objeto es eliminado, sería la contraparte del constructor y es conocido como el destructor. A pesar de su existencia, los destructores no pueden llamarse explícitamente. Eso sucede cuando el "Garbage Collector" lo considere oportuno. Entonces una forma de ejecutar el destructor es llamando explícitamente al Garbage Collector (con las siglas GC) para que haga limpieza y liberación de memoria.

Los destructores se nombran iniciando con el símbolo ~ seguido del nombre de la clase. No tienen parámetros.

D/039a.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {
   public static void Main() {
      Procedimiento();

      //Ejecuta el Garbage Collector
      GC.Collect(); //Limpia todo
      GC.WaitForPendingFinalizers(); //Espera que se limpie todo

      Console.WriteLine("Termina el programa");
   }

   public static void Procedimiento() {
      //Se instancia la clase con una variable local
      MiClase objClase = new MiClase(2010, 7.15, 'S', "Sally");
      objClase.Imprime();

      //Aquí debería ejecutarse el destructor de esa clase
   }
}
```

D/039b.cs

```
namespace Ejemplo;

partial class MiClase {
   public int ValorA { get; set; }
   public double ValorB { get; set; }
   public char ValorC { get; set; }
```

```
    public string ValorD { get; set; }

    public MiClase(int valA, double valB, char valC, string valD) {
        ValorA = valA;
        ValorB = valB;
        ValorC = valC;
        ValorD = valD;
    }

    public void Imprime() {
        Console.WriteLine("Valores");
        Console.WriteLine(ValorA);
        Console.WriteLine(ValorB);
        Console.WriteLine(ValorC);
        Console.WriteLine(ValorD);
    }

    //Destructor
    ~MiClase() {
        Console.WriteLine("Ejecuta el destructor");
    }
}
```

Nota: Por lo publicado en diversos foros sobre los destructores y el uso del Garbage Collector, esto debe hacerlo con mucho cuidado, así que se recomienda no hacer uso de destructores, ni llamar al Garbage Collector.

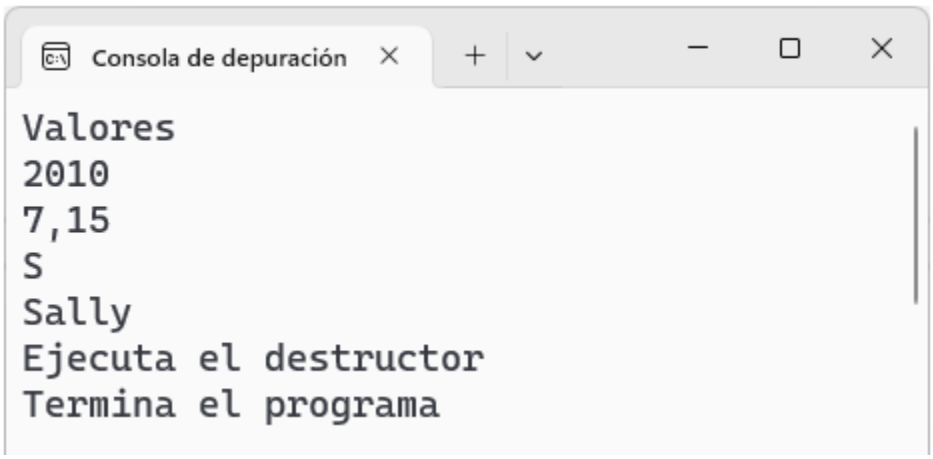

Ilustración 44: Destructores

Patrones de diseño

A continuación, algunos patrones de diseño implementados en C#:

Factory Method

El patrón de diseño Factory Method es un patrón creacional que se utiliza para crear objetos sin tener que especificar su clase exacta.

Se necesita crear objetos de una clase, pero no se sabe qué clase exacta necesita hasta que se ejecuta el programa. El patrón Factory Method permite crear objetos sin tener que especificar su clase exacta. En lugar de crear objetos directamente, se utiliza un método de fábrica para crear objetos. Este método de fábrica se encarga de crear el objeto correcto según los parámetros que se le pasen.

```
namespace Ejemplo;
//Patrón: Factory Method

//Interface que obliga a definir el método dibujar
interface IFigura {
   void Dibujar();
}

class Circulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Se hace el dibujo de un círculo");
   }
}

class Rectangulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Estoy dibujando un rectángulo");
   }
}

class Triangulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Ahora se dibuja un triángulo");
   }
}

class FabricaFiguras {
   //Dependiendo del parámetro retorna uno u otro objeto
   public IFigura GetFigura(string TipoFigura) {
      if (TipoFigura.Equals("CIRCULO"))
         return new Circulo();

      if (TipoFigura.Equals("RECTANGULO"))
         return new Rectangulo();

      if (TipoFigura.Equals("TRIANGULO"))
         return new Triangulo();

      return null;
   }
}

class Program {
   static void Main() {
      FabricaFiguras objeto = new();

      //Obtiene un objeto círculo
```

```
        IFigura Figura1 = objeto.GetFigura("CIRCULO");

        //Llama el método de dibujar del objeto círculo
        Figura1.Dibujar();

        //Obtiene un objeto rectángulo
        IFigura Figura2 = objeto.GetFigura("RECTANGULO");

        //Llama el método de dibujar del objeto rectángulo
        Figura2.Dibujar();

        //Obtiene un objeto triángulo
        IFigura Figura3 = objeto.GetFigura("TRIANGULO");

        //Llama el método de dibujar del objeto triángulo
        Figura3.Dibujar();
    }
}
```

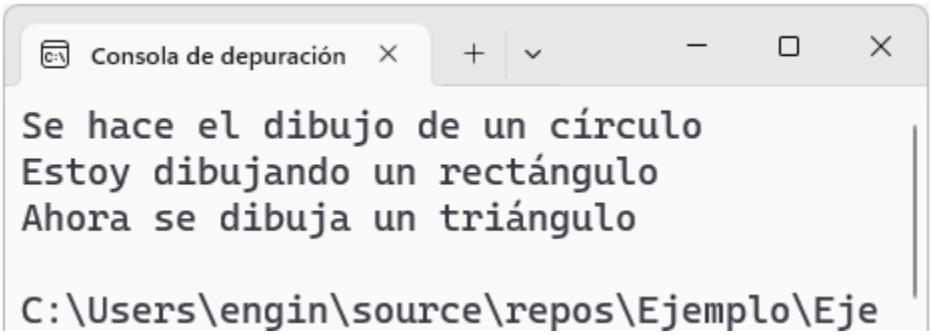

Ilustración 45: Factory Method

Abstract Factory

El patrón de diseño Abstract Factory es un patrón creacional que se utiliza para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.

Es necesario crear un conjunto de objetos que trabajen juntos, pero no se sabe qué objetos específicos se necesitan hasta que se ejecuta el programa: El patrón Abstract Factory permite crear una fábrica abstracta que define una interfaz para crear objetos relacionados o dependientes. Luego, puede crear fábricas concretas que implementan la fábrica abstracta y crean objetos específicos. De esta manera, puede crear diferentes familias de objetos relacionados o dependientes sin tener que cambiar el código del cliente.

D/041.cs

```
namespace Ejemplo;
//Patrón: Abstract Factory

//Interface que obliga a definir el método dibujar
public interface IFigura {
   void Dibujar();
}

class Rectangulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Estoy dibujando un rectángulo");
   }
}

class Triangulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Ahora se dibuja un triángulo");
   }
}

class Circulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Se hace el dibujo de un círculo");
   }
}

public interface IColor {
   void Rellenar();
}

class Rojo : IColor {
   public void Rellenar() {
      Console.WriteLine("Pinta de rojo");
   }
```

```
}

class Verde : IColor {
   public void Rellenar() {
      Console.WriteLine("Un verde es pintado");
   }
}

class Azul : IColor {
   public void Rellenar() {
      Console.WriteLine("Ahora de azul es rellenado");
   }
}

public abstract class Fabrica {
   public abstract IFigura GetFigura(string TipoFigura);
   public abstract IColor GetColor(string color);
}

public class FabricaFiguras : Fabrica {
   //Dependiendo del parámetro retorna uno u otro objeto
   public override IFigura GetFigura(string TipoFigura) {

      if (TipoFigura.Equals("CIRCULO"))
         return new Circulo();

      if (TipoFigura.Equals("RECTANGULO"))
         return new Rectangulo();

      if (TipoFigura.Equals("TRIANGULO"))
         return new Triangulo();

      return null;
   }

   public override IColor GetColor(string color) {
      return null;
   }
}

class FabricaColores : Fabrica {
   //Dependiendo del parámetro retorna uno u otro objeto
   public override IFigura GetFigura(string TipoFigura) {
      return null;
   }

   public override IColor GetColor(string color) {
```

```
      if (color.Equals("ROJO"))
         return new Rojo();

      if (color.Equals("VERDE"))
         return new Verde();

      if (color.Equals("AZUL"))
         return new Azul();

      return null;
   }
}

class CreaFabricas {
   public static Fabrica GetFabrica(string seleccion) {

      if (seleccion.Equals("FIGURA"))
         return new FabricaFiguras();

      if (seleccion.Equals("COLOR"))
         return new FabricaColores();

      return null;
   }
}

class Program {
   static void Main() {
      //Trae una determinada fábrica
      //en este caso de FIGURA
      Fabrica fig = CreaFabricas.GetFabrica("FIGURA");

      //Obtenida la fábrica, se solicita
      //un tipo de objeto de esa fábrica
      IFigura figura1 = fig.GetFigura("CIRCULO");

      //Llama un método de ese objeto
      //dado por la fábrica en particular
      figura1.Dibujar();

      //Obtenida la fábrica, se solicita
      //un tipo de objeto de esa fábrica
      IFigura figura2 = fig.GetFigura("RECTANGULO");

      //Llama un método de ese objeto dado
      //por la fábrica en particular
      figura2.Dibujar();
```

```
        //Obtenida la fábrica, se solicita
        //un tipo de objeto de esa fábrica
        IFigura figura3 = fig.GetFigura("TRIANGULO");

        //Llama un método de ese objeto dado
        //por la fábrica en particular
        figura3.Dibujar();

        //Trae una determinada fábrica
        //en este caso de COLOR
        Fabrica color = CreaFabricas.GetFabrica("COLOR");

        //Obtenida la fábrica, se solicita
        //un tipo de objeto de esa fábrica
        IColor color1 = color.GetColor("ROJO");

        //Llama un método de ese objeto dado
        //por la fábrica en particular
        color1.Rellenar();

        //Obtenida la fábrica, se solicita
        //un tipo de objeto de esa fábrica
        IColor color2 = color.GetColor("VERDE");

        //Llama un método de ese objeto dado
        //por la fábrica en particular
        color2.Rellenar();

        //Obtenida la fábrica, se solicita un
        //tipo de objeto de esa fábrica
        IColor color3 = color.GetColor("AZUL");

        //Llama un método de ese objeto dado por
        //la fábrica en particular
        color3.Rellenar();
    }
}
```

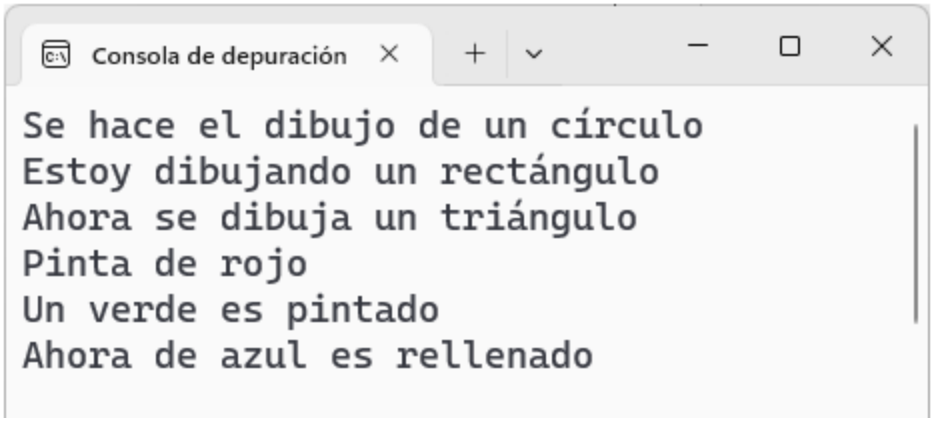

Ilustración 46: Abstract Factory

Singleton

El patrón de diseño Singleton es un patrón creacional que se utiliza para garantizar que una clase tenga exactamente una instancia y proporcionar un punto de acceso global a esta. En otras palabras, el patrón Singleton se utiliza para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

D/042.cs

```
namespace Ejemplo;
//Patrón: Singleton

class ObjetoUnico {
   //Genera un objeto de ObjetoUnico
   private static ObjetoUnico instancia = new ObjetoUnico();

   //Hace el constructor privado por lo que
   //no puede ser instanciado
   private ObjetoUnico() { }

   //Retorna la única instancia de esta clase
   public static ObjetoUnico GetInstancia() {
      return instancia;
   }

   public void Mensaje() {
      Console.WriteLine("Esta es una prueba");
   }
}

class Program {
   static void Main() {
      //Quite el comentario de esta instrucción
      //y generará un error al compilar
      //ObjetoUnico pruebaObjeto = new ObjetoUnico();

      //Obtiene el único objeto instanciable
      ObjetoUnico miObjeto = ObjetoUnico.GetInstancia();

      //Muestra un mensaje
      miObjeto.Mensaje();
   }
}
```

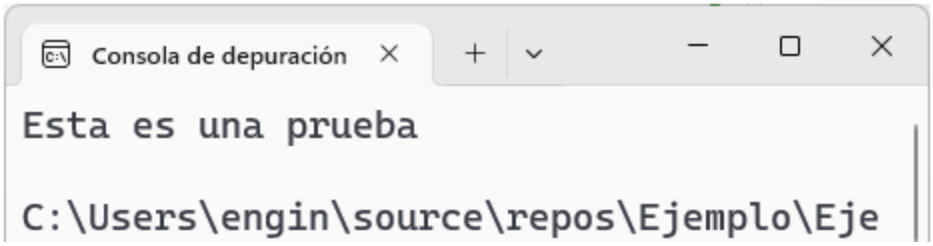

Ilustración 47: Singleton

Builder

El patrón de diseño Builder es un patrón creacional que se utiliza para crear objetos complejos paso a paso. Este patrón permite la creación de diferentes tipos y representaciones de un objeto utilizando el mismo proceso de construcción.

Cuando es necesario crear un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados: Si se intenta crear el objeto pasando argumentos a un constructor, puede que se termine con un constructor con muchos parámetros, muchos de los cuales no se usarían en la mayoría de los casos. El patrón Builder permite abstraer el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

D/043.cs

```
namespace Ejemplo;
//Patrón: Builder

public interface IEmpacado {
   //Obligar a hacer el método Empaque()
   string Empaque();
}

class Envoltura : IEmpacado {
   public string Empaque() {
      return "Empaque Ecológico";
   }
}

public class Botella : IEmpacado {
   public string Empaque() {
      return "Botella biodegradable";
   }
}

//Todo producto en la comida tendrá estos
//ítems: Nombre, como se empaca, precio
public interface Item {
   string Nombre();
   IEmpacado Empacando();
   float Precio();
}

public abstract class Hamburguesa : Item {
   public IEmpacado Empacando() {
      return new Envoltura();
   }
   public abstract float Precio();
   public abstract string Nombre();
```

```
}

public class HamburguesaPollo : Hamburguesa {
   public override float Precio() {
      return 7000;
   }

   public override string Nombre() {
      return "Hamburguesa de pollo";
   }
}

class HamburguesaVegetariana : Hamburguesa {
   public override float Precio() {
      return 5000;
   }

   public override string Nombre() {
      return "Hamburguesa vegetariana";
   }
}

public abstract class BebidaFria : Item {
   public IEmpacado Empacando() {
      return new Botella();
   }
   public abstract float Precio();
   public abstract string Nombre();
}

class Malteada : BebidaFria {
   public override float Precio() {
      return 4700;
   }

   public override string Nombre() {
      return "Malteada";
   }
}

class CaFeFrio : BebidaFria {
   public override float Precio() {
      return 4000;
   }

   public override string Nombre() {
      return "Café frío";
   }
```

```
}

class Comida {
   private List<Item> items = new List<Item>();

   public void AddItem(Item item) {
      items.Add(item);
   }

   public float GetCosto() {
      float costo = 0.0f;
      foreach (Item item in items) {
         costo += item.Precio();
      }
      return costo;
   }

   public void MostrarItems() {
      foreach (Item item in items) {
         Console.Write("Item: " + item.Nombre());
         Console.Write(", Empaque: " + item.Empacando().Empaque());
         Console.WriteLine(", Precio: " + item.Precio());
      }
   }
}

//Prepara la comida dependiendo si es vegetariana o no
class FabricaComida {
   public Comida Vegetariano() {
      Comida miComida = new();
      miComida.AddItem(new HamburguesaVegetariana());
      miComida.AddItem(new CaFeFrio());
      return miComida;
   }

   public Comida NoVegetariano() {
      Comida miComida = new();
      miComida.AddItem(new HamburguesaPollo());
      miComida.AddItem(new Malteada());
      return miComida;
   }
}

class Program {
   static void Main() {
      FabricaComida miComida = new();

      Comida vegetariano = miComida.Vegetariano();
```

```
        Console.WriteLine("Comida vegetariana");
        vegetariano.MostrarItems();
        Console.WriteLine("Costo: " + vegetariano.GetCosto());

        Comida noVegetariano = miComida.NoVegetariano();
        Console.WriteLine("\n\nComida No vegetariana");
        noVegetariano.MostrarItems();
        Console.WriteLine("Costo: " + noVegetariano.GetCosto());
    }
}
```

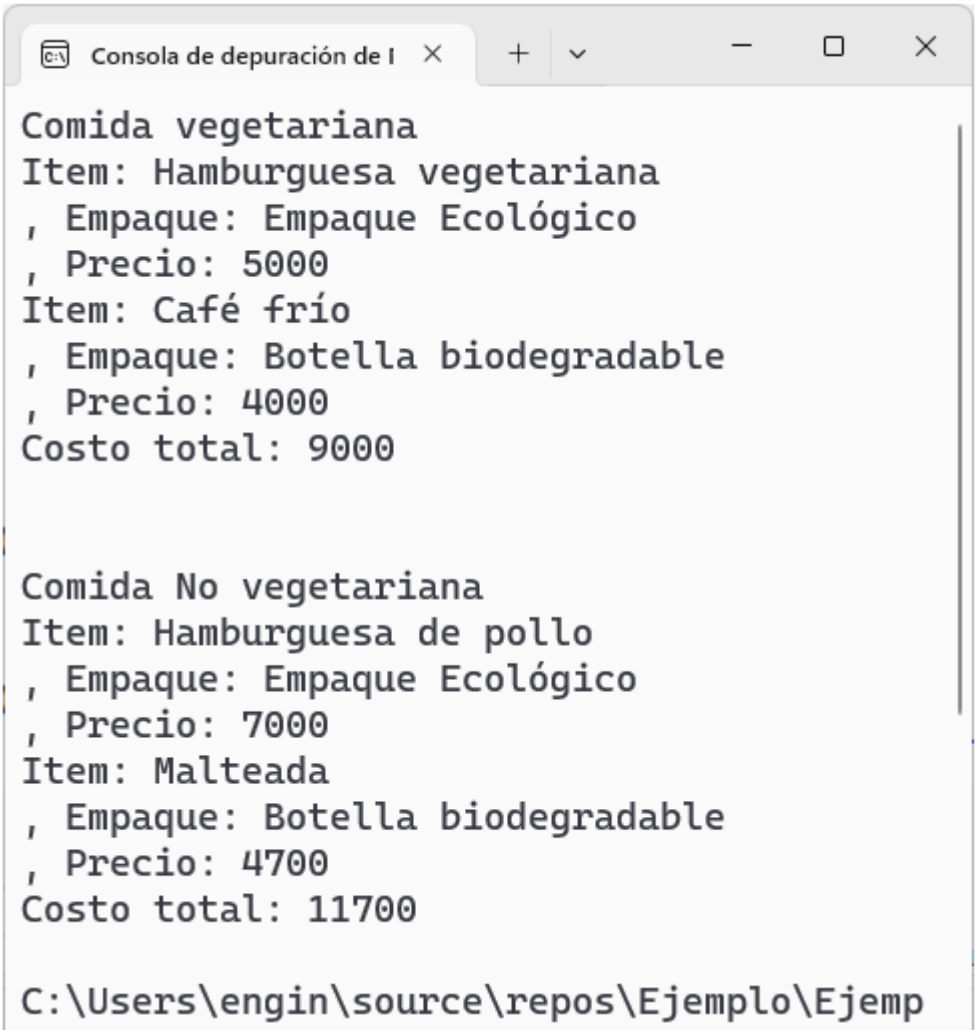

Ilustración 48: Builder

Adapter

El patrón de diseño Adapter es un patrón estructural que se utiliza para adaptar una interfaz existente a otra interfaz. En otras palabras, el adaptador actúa como un intermediario entre dos interfaces incompatibles y proporciona una capa adicional de abstracción para permitir que los objetos trabajen juntos.

Se tienen dos clases con interfaces incompatibles, y es necesario que trabajen juntas. El patrón Adapter permite crear una clase intermedia que actúa como un traductor entre las dos interfaces. El adaptador implementa la interfaz del cliente y utiliza la interfaz del servicio para realizar la traducción.

D/044.cs

```
namespace Ejemplo;
//Patrón de diseño: Adapter

public interface IEjecutorMultimedia {
   void Ejecutar(string TipoAudio, string NombreArchivo);
}

public interface IEjecutorAvanzadoArchivosMultimedia {
   void EjecutaVLC(string NombreArchivo);
   void EjecutaMP4(string NombreArchivo);
}

class EjecutorVLC : IEjecutorAvanzadoArchivosMultimedia {
   public void EjecutaVLC(string NombreArchivo) {
      Console.WriteLine("Ejecutando VLC: " + NombreArchivo);
   }
   public void EjecutaMP4(string NombreArchivo) {
   }
}

class EjecutorMP4 : IEjecutorAvanzadoArchivosMultimedia {
   public void EjecutaVLC(string NombreArchivo) {
   }
   public void EjecutaMP4(string NombreArchivo) {
      Console.WriteLine("Ejecutando MP4: " + NombreArchivo);
   }
}

class AdaptadorMultimedia : IEjecutorMultimedia {
   IEjecutorAvanzadoArchivosMultimedia ejecutorAvanzado;

   //Constructor
   public AdaptadorMultimedia(string TipoAudio) {
      if (TipoAudio.Equals("vlc")) {
         ejecutorAvanzado = new EjecutorVLC();
```

```
      }
      if (TipoAudio.Equals("mp4")) {
         ejecutorAvanzado = new EjecutorMP4();
      }
   }

   //Dependiendo del tipo de audio llama a VLC o MP4
   public void Ejecutar(string TipoAudio, string NombreArchivo) {
      if (TipoAudio.Equals("vlc")) {
         ejecutorAvanzado.EjecutaVLC(NombreArchivo);
      }
      else if (TipoAudio.Equals("mp4")) {
         ejecutorAvanzado.EjecutaMP4(NombreArchivo);
      }
   }
}

class EjecutorAudio : IEjecutorMultimedia {

   AdaptadorMultimedia adaptadorMultimedia;

   public void Ejecutar(string TipoAudio, string NombreArchivo) {
      //Archivos MP3
      if (TipoAudio.Equals("mp3")) {
         Console.WriteLine("Ejecutando MP3: " + NombreArchivo);
      } //Otros formatos
      else if (TipoAudio.Equals("vlc") || TipoAudio.Equals("mp4")) {
         adaptadorMultimedia = new AdaptadorMultimedia(TipoAudio);
         adaptadorMultimedia.Ejecutar(TipoAudio, NombreArchivo);
      }
      else {
         Console.Write("Medio inválido. (" + TipoAudio);
         Console.WriteLine(") es un formato no soportado");
      }
   }
}

class Program {
   static void Main() {
      EjecutorAudio Multimedia = new EjecutorAudio();

      Multimedia.Ejecutar("mp3", "MiMusica.mp3");
      Multimedia.Ejecutar("mp4", "unSonido.mp4");
      Multimedia.Ejecutar("vlc", "FondoMusical.vlc");
      Multimedia.Ejecutar("avi", "unAudio.avi");
   }
}
```

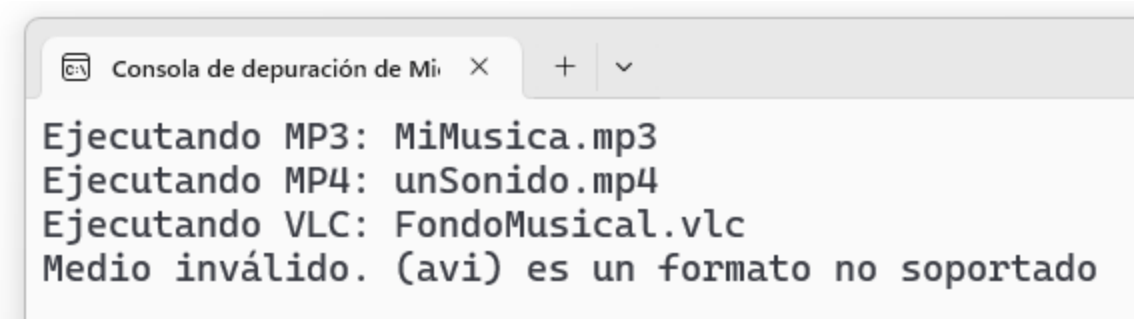

Ilustración 49: Adapter

Composite

El patrón de diseño Composite es un patrón estructural que se utiliza para representar jerarquías parte-todo como un árbol. Este patrón permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme.

Se tiene una estructura jerárquica de objetos, donde cada objeto puede ser un objeto simple o un objeto compuesto: el patrón Composite permite tratar tanto los objetos simples como los objetos compuestos de la misma manera, como si fueran una instancia única de un objeto.

D/045.cs

```
namespace Ejemplo;
//Patrón de diseño: Composite

public class Empleado {
   private string nombre;
   private string departamento;
   private int salario;
   private List<Empleado> subordinados;

   //Constructor
   public Empleado(string nombre, string departamento, int salario) {
      this.nombre = nombre;
      this.departamento = departamento;
      this.salario = salario;
      subordinados = new List<Empleado>();
   }

   public void Adicionar(Empleado objEmpleado) {
      subordinados.Add(objEmpleado);
   }

   public void Quitar(Empleado objEmpleado) {
      subordinados.Remove(objEmpleado);
   }

   public List<Empleado> GetSubordinados() {
      return subordinados;
   }

   public new string ToString() {
      string Cad = "Empleado => Nombre: " + nombre;
      Cad += ", departamento: " + departamento;
      Cad += ", salario: " + salario;
      return Cad;
   }
}
```

```
internal class Program {
   static void Main() {
      Empleado Gerente = new("Laura", "Gerente", 5000);
      Empleado jefeVentas = new("Patricia", "Ventas", 3000);
      Empleado jefeMercadeo = new("Adriana", "Mercadeo", 3000);
      Empleado disenador1 = new("Sandra", "Marketing", 2000);
      Empleado disenador2 = new("Alejandra", "Marketing", 2000);
      Empleado vendedor1 = new("Francisca", "Ventas", 2000);
      Empleado vendedor2 = new("Flor", "Ventas", 2000);

      Gerente.Adicionar(jefeVentas);
      Gerente.Adicionar(jefeMercadeo);

      jefeVentas.Adicionar(vendedor1);
      jefeVentas.Adicionar(vendedor2);

      jefeMercadeo.Adicionar(disenador1);
      jefeMercadeo.Adicionar(disenador2);

      //Imprime todos los empleados de la organización
      Console.WriteLine(Gerente.ToString());

      foreach (Empleado jefe in Gerente.GetSubordinados()) {
         Console.WriteLine(jefe.ToString());

         foreach (Empleado empleado in jefe.GetSubordinados()) {
            Console.WriteLine(empleado.ToString());
         }
      }
   }
}
```

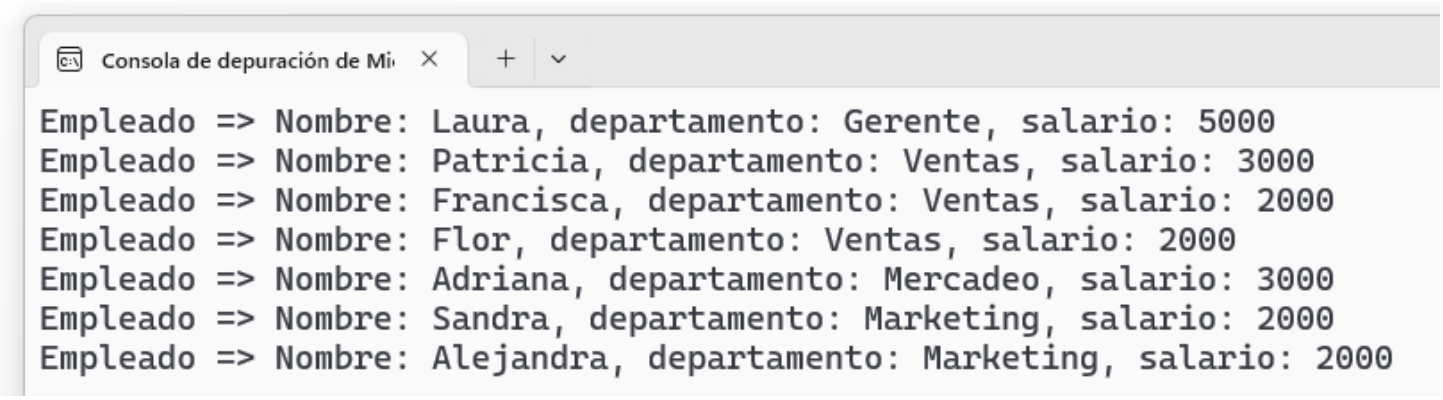

Ilustración 50: Composite

Facade

El patrón de diseño Facade es un patrón estructural que se utiliza para simplificar la complejidad de un sistema. Imaginarse tener un sistema complejo con muchos subsistemas, cada uno con su propia interfaz. Si un cliente quiere interactuar con el sistema, tendría que conocer todas las interfaces de los subsistemas, lo que puede ser muy complicado. El patrón Facade proporciona una interfaz unificada para un conjunto de interfaces en un subsistema. La idea detrás de este patrón es proporcionar una interfaz simple para un subsistema complejo, reduciendo así la complejidad del sistema y minimizando las comunicaciones y dependencias entre los subsistemas.

D/046.cs

```
namespace Ejemplo;
//Patrón de diseño: Facade

interface IFigura {
   void Dibujar();
}

class Circulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Dibujando un círculo");
   }
}

class Rectangulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Traza un rectángulo");
   }
}

class Triangulo : IFigura {
   public void Dibujar() {
      Console.WriteLine("Delinea un triángulo");
   }
}

class HacerFigura {
   private IFigura circulo;
   private IFigura rectangulo;
   private IFigura triangulo;

   public HacerFigura() {
      circulo = new Circulo();
      rectangulo = new Rectangulo();
      triangulo = new Triangulo();
   }
```

```
    public void DibujaCirculo() {
        circulo.Dibujar();
    }

    public void DibujaRectangulo() {
        rectangulo.Dibujar();
    }

    public void DibujaTriangulo() {
        triangulo.Dibujar();
    }
}

class Program {
    static void Main() {
        HacerFigura hacefigura = new();

        hacefigura.DibujaCirculo();
        hacefigura.DibujaRectangulo();
        hacefigura.DibujaTriangulo();
    }
}
```

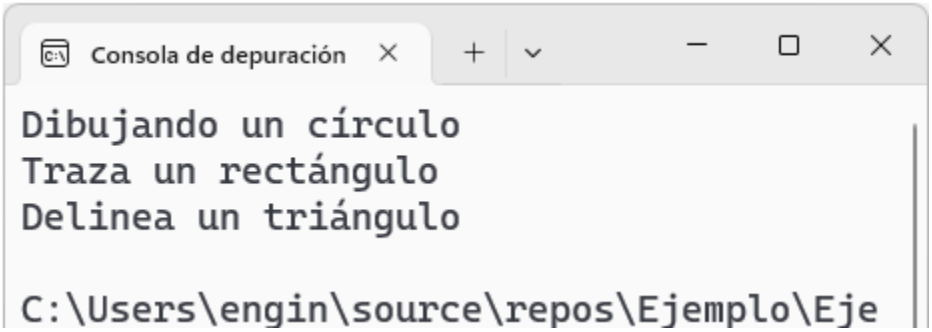

Ilustración 51: Facade

Modelo Vista Controlador

El patrón de diseño modelo vista controlador (MVC) es una forma de organizar una aplicación en tres componentes principales: el modelo, la vista y el controlador. Cada componente tiene una función específica y se comunica con los otros para lograr una interacción fluida entre el usuario y la aplicación. Una breve explicación de cada componente:

Modelo: Es el componente que gestiona los datos y la lógica de la aplicación. Se encarga de almacenar, manipular y validar los datos, así como de interactuar con fuentes externas como bases de datos o APIs. El modelo no tiene conocimiento de la interfaz de usuario, solo se ocupa de los datos.

Vista: Es el componente que muestra los datos al usuario. Se encarga de generar la interfaz gráfica de usuario (GUI). La vista recibe los datos del modelo a través del controlador y los presenta de forma atractiva y comprensible. La vista también puede capturar las acciones del usuario, como hacer clic en un botón o introducir un texto, y enviarlas al controlador.

Controlador: Es el componente que coordina la comunicación entre el modelo y la vista. Se encarga de procesar las peticiones del usuario, como solicitar una página o enviar un formulario, y de invocar al modelo para obtener o modificar los datos necesarios. El controlador también decide qué vista mostrar al usuario según el resultado del modelo.

D/047.cs

```
namespace Ejemplo;
//Patrón de diseño: Modelo Vista Controlador
class Gente {
   public string Codigo { get; set; }
   public string Nombre { get; set; }
}

class ControladorGente {
   private Gente modelo;
   private VisorGente vista;

   public ControladorGente(Gente modelo, VisorGente vista) {
      this.modelo = modelo;
      this.vista = vista;
   }

   public void setNombreGente(string nombre) {
      modelo.Nombre = nombre;
   }

   public string getNombreGente() {
      return modelo.Nombre;
   }

   public void setCodigoGente(string codigo) {
```

```
      modelo.Codigo = codigo;
   }

   public string getCodigoGente() {
      return modelo.Codigo;
   }

   public void ActualizarVista() {
      vista.ImprimeGente(modelo.Nombre, modelo.Codigo);
   }
}

class VisorGente {
   public void ImprimeGente(string Nombre, string Codigo) {
      Console.WriteLine("Gente: ");
      Console.WriteLine("Nombre: " + Nombre);
      Console.WriteLine("Código: " + Codigo);
   }
}

class Program {
   static void Main() {
      Gente modelo = TraeGenteBaseDatos();
      VisorGente vista = new();
      ControladorGente control = new(modelo, vista);

      control.ActualizarVista();
      control.setNombreGente("Laura");
      control.ActualizarVista();
   }

   private static Gente TraeGenteBaseDatos() {
      Gente Gente = new();
      Gente.Nombre = "Johanna";
      Gente.Codigo = "17123456";
      return Gente;
   }
}
```

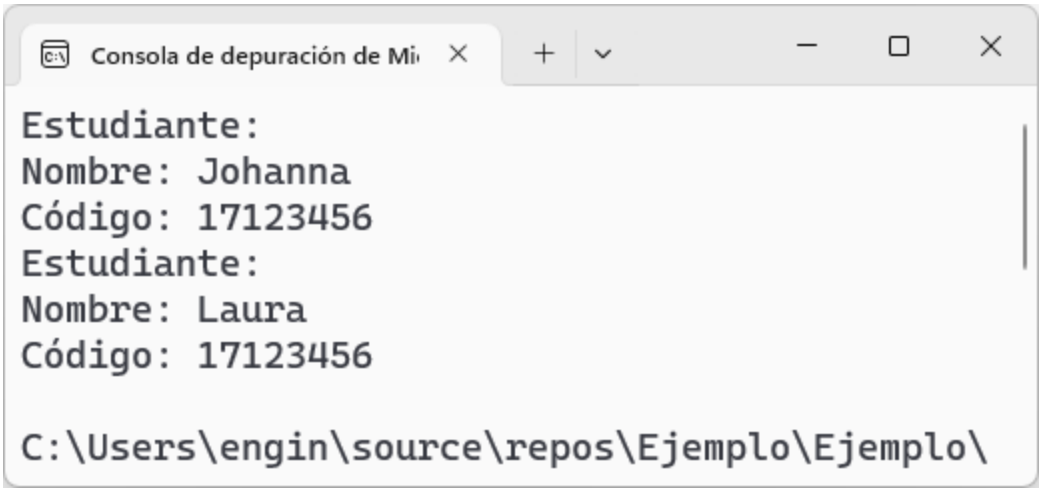

Ilustración 52: Modelo Vista Controlador

SOLID

SOLID es un conjunto de principios diseñados para mejorar el diseño y la mantenibilidad del código en la Programación Orientada a Objetos (POO). Estos son:

S: Single Responsibility Principle (Principio de Responsabilidad Única) Cada clase debe tener una única responsabilidad, es decir, debe realizar una sola tarea o propósito.

O: Open/Closed Principle (Principio Abierto/Cerrado) Las clases deben estar abiertas para ser extendidas, pero cerradas para modificaciones. Esto significa que puedes agregar nuevas funcionalidades sin cambiar el código existente.

L: Liskov Substitution Principle (Principio de Sustitución de Liskov) Los objetos de una clase derivada deben poder sustituir a los objetos de la clase base sin alterar el funcionamiento del programa.

I: Interface Segregation Principle (Principio de Segregación de Interfaces) Las interfaces deben ser específicas y pequeñas, de modo que las clases que implementan esas interfaces no se vean obligadas a usar métodos que no necesitan.

D: Dependency Inversion Principle (Principio de Inversión de Dependencias) Las clases deben depender de abstracciones y no de implementaciones concretas, favoreciendo el uso de interfaces o clases abstractas.

Ejemplo:

D/048.cs

```
namespace Ejemplo;
// S: Single Responsibility Principle (SRP)
// sólo se encarga de los datos del pedido.
public class Pedido {
   public int Codigo { get; set; }
   public decimal TotalCosto { get; set; }

   public Pedido(int Codigo, decimal TotalCosto) {
      this.Codigo = Codigo;
      this.TotalCosto = TotalCosto;
   }
}
```

```
// Clase con la responsabilidad de persistir los pedidos
// se encarga de guardar los pedidos, separando la responsabilidad
public class PersistePedidos {
   public void Guardar(Pedido objPedido) {
      Console.WriteLine("Pedido: " + objPedido.Codigo + " guardado con un
total de: " + objPedido.TotalCosto);
   }
}

// O: Open/Closed Principle (OCP)
//La interfaz `IDescuento` permite extender el sistema con nuevos tipos de
descuento
//(`SinDescuento`, `DescuentoTemporada`) sin modificar las clases
existentes.
public interface IDescuento {
   decimal AplicaDescuento(decimal TotalCosto);
}

public class SinDescuento : IDescuento {
   public decimal AplicaDescuento(decimal TotalCosto) {
      return TotalCosto;
   }
}

public class DescuentoTemporada : IDescuento {
   public decimal AplicaDescuento(decimal TotalCosto) {
      return TotalCosto * 0.9m; // 10% de descuento
   }
}

// L: Liskov Substitution Principle (LSP)
//La clase abstracta `Notificacion` asegura que sus subclases
//(`EmailNotificacion`, `SmsNotificacion`) sean intercambiables sin
afectar el comportamiento.
public abstract class Notificacion {
   public abstract void Notificar(string Mensaje);
}

public class EmailNotificacion : Notificacion {
   public override void Notificar(string Mensaje) {
      Console.WriteLine("Correo enviado: " + Mensaje);
   }
}

public class SmsNotificacion : Notificacion {
   public override void Notificar(string Mensaje) {
      Console.WriteLine("SMS enviado: " + Mensaje);
   }
```

```
}

// I: Interface Segregation Principle (ISP)
//Las interfaces separadas, como `IProcesoPago`, aseguran que
//las clases solo implementen métodos que necesitan
//(`TarjetaCredito`, `PorPayPal`)
public interface IProcesoPago {
   void Pago(decimal Precio);
}

public class TarjetaCredito : IProcesoPago {
   public void Pago(decimal Precio) {
      Console.WriteLine("Pago con tarjeta crédito de: " + Precio + " fue
procesado.");
   }
}

public class PorPayPal : IProcesoPago {
   public void Pago(decimal Precio) {
      Console.WriteLine("Pago con PayPal de: " + Precio + " fue
procesado.");
   }
}

// D: Dependency Inversion Principle (DIP)
// La clase `ServicioPedido` no depende de implementaciones concretas,
// sino de abstracciones (`IDescuento`, `Notificacion`, `IProcesoPago`),
// lo que mejora la flexibilidad y el testeo
public class ServicioPedido {
   private readonly IDescuento _descuento;
   private readonly Notificacion _notificacion;
   private readonly PersistePedidos _repositorio;
   private readonly IProcesoPago _formadePago;

   public ServicioPedido(IDescuento descuento, Notificacion notificacion,
PersistePedidos repositorio, IProcesoPago formadePago) {
      _descuento = descuento;
      _notificacion = notificacion;
      _repositorio = repositorio;
      _formadePago = formadePago;
   }

   public void ProcesarPedido(Pedido objPedido) {
      // Aplicar descuento
      objPedido.TotalCosto =
_descuento.AplicaDescuento(objPedido.TotalCosto);

      // Procesar pago
```

```
        _formadePago.Pago(objPedido.TotalCosto);

        // Guardar pedido
        _repositorio.Guardar(objPedido);

        // Enviar notificación
        _notificacion.Notificar("Pedido " + objPedido.Codigo + " procesado
exitósamente con un total de: " + objPedido.TotalCosto);
    }
}

// Uso
class Program {
    static void Main() {
        var objPedido = new Pedido(1, 100m);

        // Inyectar dependencias
        var Descuento = new DescuentoTemporada();
        var Notificacion = new EmailNotificacion();
        var Persistencia = new PersistePedidos();
        var MododePago = new TarjetaCredito();

        var objServicioPedido = new ServicioPedido(Descuento, Notificacion,
Persistencia, MododePago);

        // Procesar pedido
        objServicioPedido.ProcesarPedido(objPedido);
    }
}
```

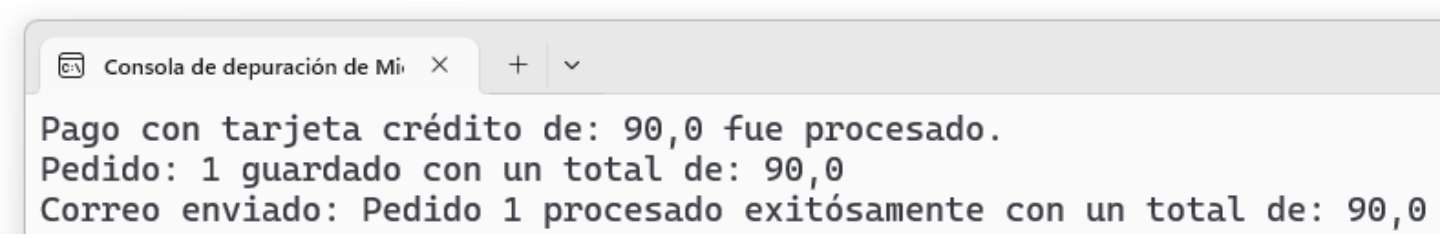

Ilustración 53: Uso de SOLID

Arreglos de objetos

Es posible tener un conjunto de objetos de la misma clase.

D/049.cs

```
namespace Ejemplo;

class Valores {
   private int Entero;
   private double ValorReal;
   private bool Booleano;
   private char Caracter;
   private string Cadena;

   public Valores(int entero, double valorReal, bool booleano, char
caracter, string cadena) {
      Entero = entero;
      ValorReal = valorReal;
      Booleano = booleano;
      Caracter = caracter;
      Cadena = cadena;
   }

   public void Imprime() {
      Console.WriteLine(Entero + ";" + ValorReal + ";" + Booleano + ";" +
Caracter + ";" + Cadena);
   }
}

internal class Program {
   static void Main(string[] args) {
      //Arreglo unidimensional de objetos
      Valores[] objetos = new Valores[10];

      //Se crean los objetos
      objetos[0] = new Valores(16, 83.2, true, 'R', "probar");
      objetos[1] = new Valores(72, 6.26, false, 'a', "koala");
      objetos[2] = new Valores(95, -5.21, false, 'K', "Rinoceronte");

      //Se imprimen los objetos
      objetos[1].Imprime();
      objetos[2].Imprime();
      objetos[0].Imprime();

      //¿Que pasaría aquí? Mensaje de error: el objeto no ha sido creado
      //objetos[4].Imprime();
   }
```


Autor: Rafael Alberto Moreno Parra

```
}
```

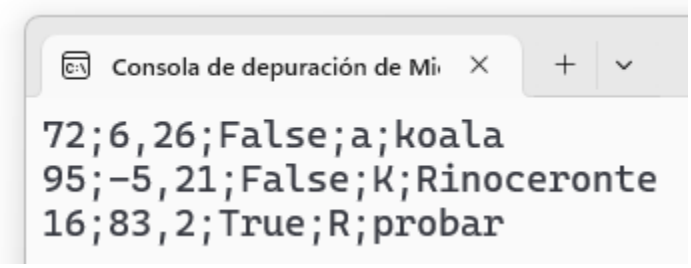

Ilustración 54: Arreglo de objetos

Llenado con ciclos

Creando conjunto de objetos usando ciclos

D/050.cs

```
namespace Ejemplo;

class Valores {
   private int Entero;
   private double ValorReal;

   public Valores(int entero, double valorReal) {
      Entero = entero;
      ValorReal = valorReal;
   }

   public void Imprime() {
      Console.WriteLine(Entero + ";" + ValorReal);
   }
}

internal class Program {
   static void Main(string[] args) {
      //Arreglo unidimensional de objetos
      Valores[] objetos = new Valores[10];

      Random Azar = new();

      //Se crean los objetos
      for (int Cont = 0; Cont < objetos.Length; Cont++)
         objetos[Cont] = new Valores(Cont, Azar.NextDouble());

      //Se imprimen los objetos
      for (int Cont = 0; Cont < objetos.Length; Cont++)
         objetos[Cont].Imprime();
   }
}
```

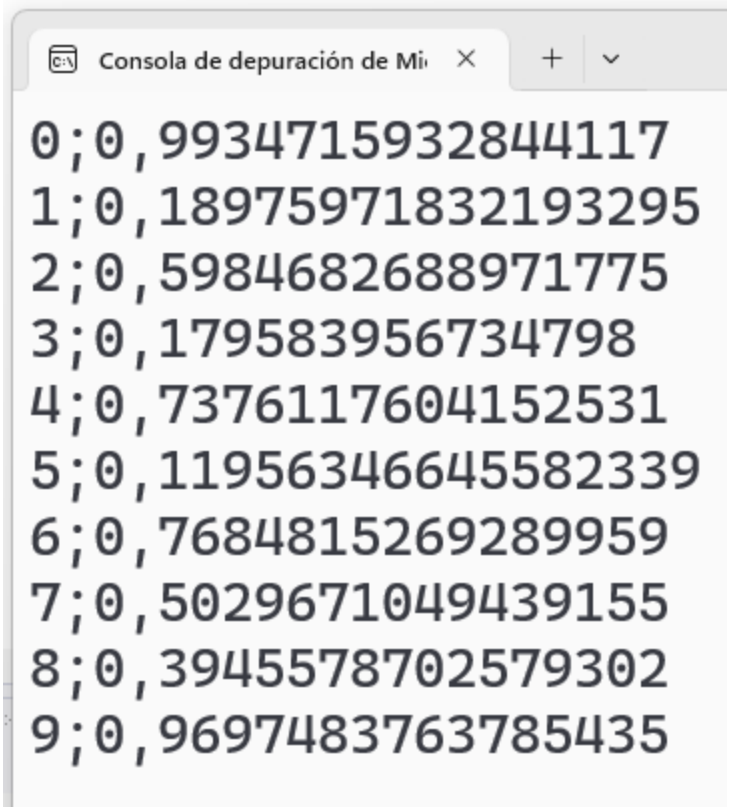

Ilustración 55: Arreglo de objetos

Ordenando

Ordenando los objetos dentro de un arreglo dependiendo del valor de algún atributo.

D/051.cs

```
namespace Ejemplo;

class Valores {
   public int Entero;
   public double ValorReal;

   public Valores(int entero, double valorReal) {
      Entero = entero;
      ValorReal = valorReal;
   }

   public void Imprime() {
      Console.WriteLine(Entero + ";" + ValorReal);
   }
}

internal class Program {
   static void Main() {
      //Arreglo unidimensional de objetos
      Valores[] objetos = new Valores[20];

      Random Azar = new();

      //Se crean los objetos
      for (int Cont = 0; Cont < objetos.Length; Cont++)
         objetos[Cont] = new Valores(Azar.Next(100), Azar.NextDouble());

      //Imprime los objetos
      Console.WriteLine("Arreglo de objetos original");
      for (int Cont = 0; Cont < objetos.Length; Cont++)
         objetos[Cont].Imprime();

      //Se procede a ordenar usando ShellSort
      Shell(objetos);

      //Imprime los objetos
      Console.WriteLine("\r\nArreglo de objetos ordenado por atributo
entero");
      for (int Cont = 0; Cont < objetos.Length; Cont++)
         objetos[Cont].Imprime();
   }
```

```
   //Ordenamiento por Shell
   static void Shell(Valores[] arr) {
      int incr = arr.Length;
      do {
         incr /= 2;
         for (int k = 0; k < incr; k++) {
            for (int i = incr + k; i < arr.Length; i += incr) {
               int j = i;
               while (j - incr >= 0 && arr[j].Entero < arr[j - incr].Entero)
{
                  int tmp = arr[j].Entero;
                  arr[j].Entero = arr[j - incr].Entero;
                  arr[j - incr].Entero = tmp;

                  double tmp2 = arr[j].ValorReal;
                  arr[j].ValorReal = arr[j - incr].ValorReal;
                  arr[j - incr].ValorReal = tmp2;

                  j -= incr;
               }
            }
         }
      } while (incr > 1);
   }
}
```

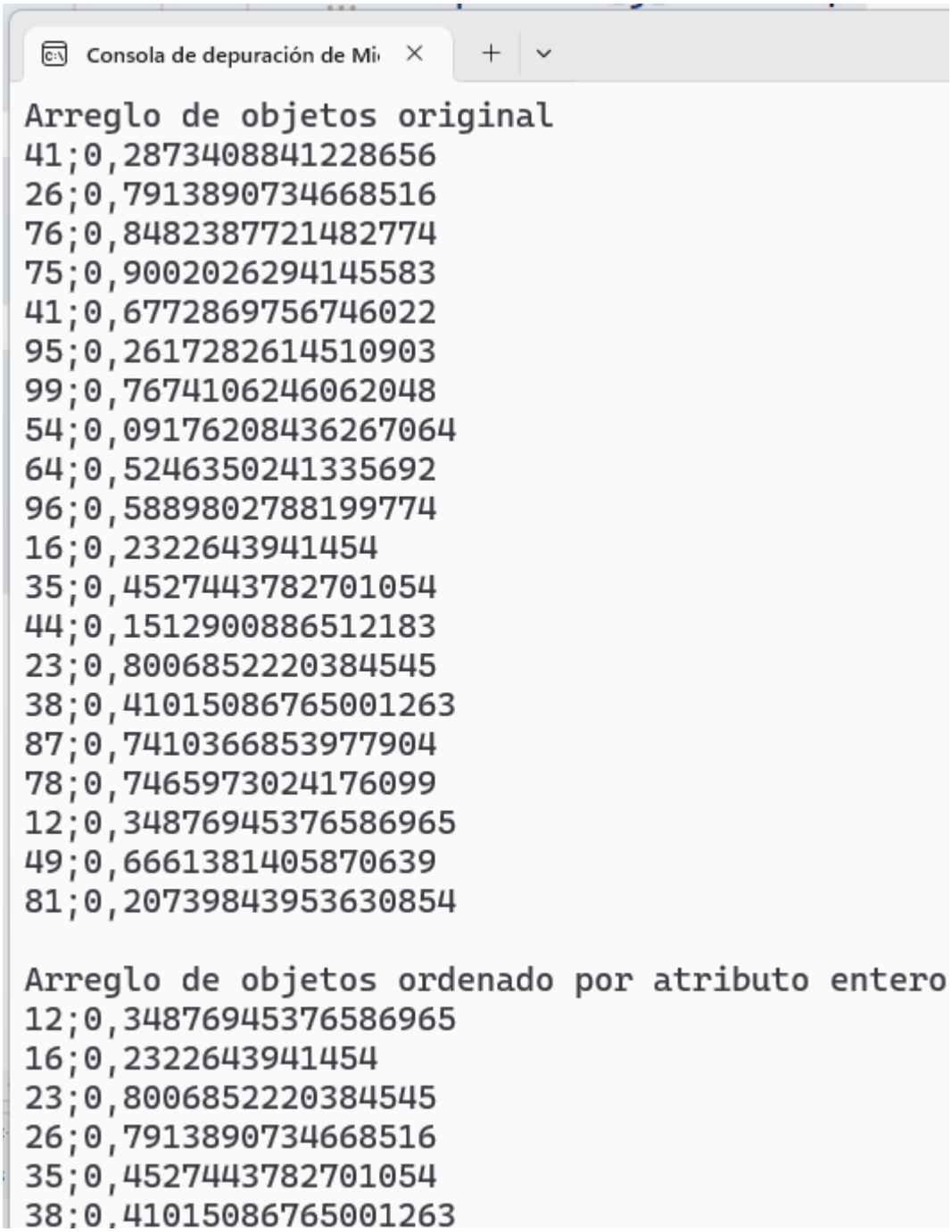

Ilustración 56: Ordenando arreglo de objetos

Delegados

Usados para mejorar la reutilización de código.

D/052.cs

```
namespace Ejemplo;

//Inicia la aplicación aquí
internal class Program {
   // 1. Se declara el delegado
   public delegate double EstrategiaDescuento(double precioOriginal);

   // 2. Distintas estrategias de descuento. Obsérvese que debe
   //coincidir el tipo de dato devuelto, el número de parámetros
   //y el tipo.
   public static double DescuentoRegular(double precio) {
      return precio * 0.95; // 5%
   }
   public static double DescuentoPremium(double precio) {
      return precio * 0.85; // 15%
   }
   public static double SinDescuento(double precio) {
      return precio; // 0%
   }


   // 3. Método que aplica el descuento usando el delegado
   public static void AplicarDescuento(string tipoCliente, double precio,
      EstrategiaDescuento descuento) {
      double precioFinal = descuento(precio);
      Console.WriteLine($"Cliente: {tipoCliente} - Precio original:
{precio:C} - Precio con descuento: {precioFinal:C}");
   }

   static void Main() {
      double precioProducto = 100;

      // 4. Se usa el delegado para aplicar distintas estrategias
      AplicarDescuento("Regular", precioProducto, DescuentoRegular);
      AplicarDescuento("Premium", precioProducto, DescuentoPremium);
      AplicarDescuento("Invitado", precioProducto, SinDescuento);
   }
}
```

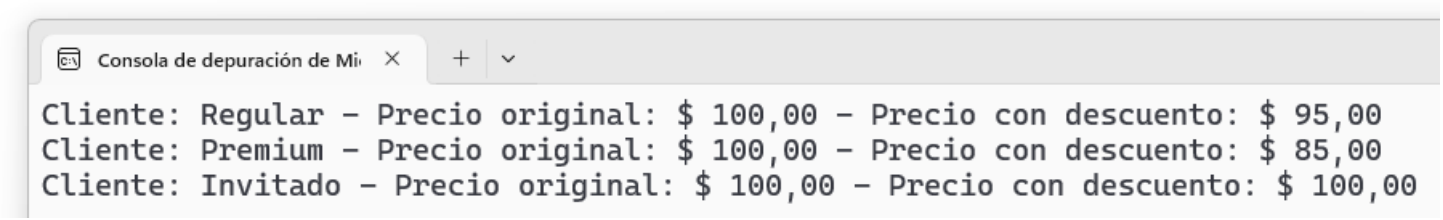

Ilustración 57: Uso de delegados

Segundo ejemplo

D/053.cs

```
namespace Ejemplo;

class Program {
   // 1. Se declara el delegado
   public delegate double Operacion(double valor);

   // 2. Métodos compatibles con el delegado
   public static double Doble(double x) => x * 2;
   public static double Cuadrado(double x) => x * x;
   public static double Raiz(double x) => Math.Sqrt(x);

   // 3. Método que recibe el delegado y aplica la operación
   public static void EjecutarOperacion(double numero, Operacion op) {
      double resultado = op(numero);
      Console.WriteLine($"Resultado: {resultado}");
   }

   static void Main() {
      Console.WriteLine("Ingrese un número:");
      double numero = Convert.ToDouble(Console.ReadLine());

      Console.WriteLine("Operación: 1 = Doble, 2 = Cuadrado, 3 = Raíz");
      string opcion = Console.ReadLine();

      Operacion operacionElegida;

      // 4. Se asigna el método al delegado según la opción
      switch (opcion) {
         case "1":
            operacionElegida = Doble;
            break;
         case "2":
            operacionElegida = Cuadrado;
            break;
         case "3":
            operacionElegida = Raiz;
            break;
         default:
            Console.WriteLine("Opción inválida.");
            return;
      }

      // 5. Ejecuta la operación
      EjecutarOperacion(numero, operacionElegida);
   }
```

```
}
```

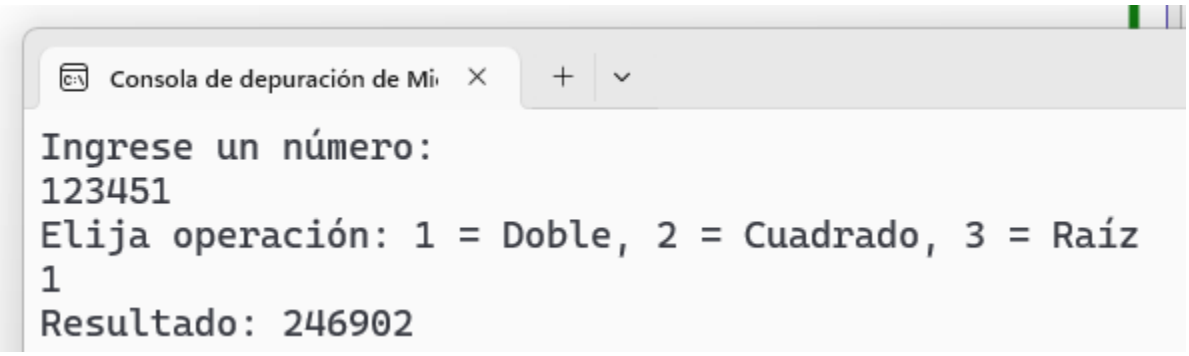

Ilustración 58: Uso de delegados

Almacenando objetos en medio persistente

Uso de JSON

Se tiene un objeto y se quiere almacenar en un medio persistente. Una forma de hacerlo es usando "serialización" y se guarda como una cadena JSON. La clase se marca con la etiqueta: [Serializable]

D/054.cs

```
using System.Text.Json;

namespace Ejemplo;

[Serializable]
public class Persona {
   public string Nombre { get; set; }
   public int Telefono { get; set; }
}

internal class Program {
   static void Main(string[] args) {
      var persona = new Persona { Nombre = "Rafael", Telefono = 3156789 };

      // Guardar
      var json = JsonSerializer.Serialize(persona);
      File.WriteAllText("persona.json", json);

      // Leer
      var jsonLeido = File.ReadAllText("persona.json");
      var personaLeida = JsonSerializer.Deserialize<Persona>(jsonLeido);

      //Imprime
      Console.WriteLine("Nombre: " + personaLeida.Nombre);
      Console.WriteLine("Teléfono: " + personaLeida.Telefono);
   }
}
```

Uso de XML

D/055.cs

```
using System.Xml.Serialization;

namespace Ejemplo;

[Serializable]
public class Persona {
   public string Nombre { get; set; }
   public int Telefono { get; set; }
}

internal class Program {
   static void Main(string[] args) {
      var persona = new Persona { Nombre = "Rafael", Telefono = 3124789 };

      var serializer = new XmlSerializer(typeof(Persona));
      using (var writer = new StreamWriter("persona.xml")) {
         serializer.Serialize(writer, persona);
      }

      // Leer
      using (var reader = new StreamReader("persona.xml")) {
         var personaLeida = (Persona)serializer.Deserialize(reader);

         //Imprime
         Console.WriteLine("Nombre: " + personaLeida.Nombre);
         Console.WriteLine("Teléfono: " + personaLeida.Telefono);
      }
   }
}
```

Genéricos

Los genéricos en C# son una característica del lenguaje que permite definir clases, interfaces, métodos y delegados con tipos parametrizados, es decir, sin especificar el tipo exacto hasta que se usa.

D/056.cs

```
namespace Ejemplo;

public class Caja<T> {
   public T Contenido { get; set; }
}

internal class Program {
   static void Main(string[] args) {
      var cajaInt = new Caja<int> { Contenido = 42 };
      var cajaTexto = new Caja<string> { Contenido = "Hola Rafael" };

      Console.WriteLine("Entero:" + cajaInt.Contenido);
      Console.WriteLine("Texto: " + cajaTexto.Contenido);
   }
}
```

D/057.cs

```
namespace Ejemplo;

//Compara dos valores si son iguales
public static class Utilidades {
   public static bool SonIguales<T>(T a, T b) {
      return EqualityComparer<T>.Default.Equals(a, b);
   }
}

internal class Program {
   static void Main(string[] args) {
      bool resultado1 = Utilidades.SonIguales(5, 5);          // true
      bool resultado2 = Utilidades.SonIguales("hola", "adiós"); // false
      bool resultado3 = Utilidades.SonIguales(3.14, 3.14);    // true

      Console.WriteLine(resultado1);
      Console.WriteLine(resultado2);
      Console.WriteLine(resultado3);
   }
}
```

¿Por qué es útil?

Funciona con cualquier tipo: int, string, double, DateTime, incluso tipos personalizados.

Usa EqualityComparer<T>.Default, que respeta la implementación de Equals() o IEquatable<T> si está definida.

Evita duplicar lógica para cada tipo.

Segundo ejemplo

D/058.cs

```
namespace Ejemplo
{
   //Compara dos valores si son iguales
   public static class Utilidades {
      public static void Intercambiar<T>(ref T a, ref T b) {
         T temp = a;
         a = b;
         b = temp;
      }
   }

   internal class Program
   {
      static void Main(string[] args)
      {
         int x = 10, y = 20;
         Utilidades.Intercambiar(ref x, ref y);
         // x = 20, y = 10

         string s1 = "uno", s2 = "dos";
         Utilidades.Intercambiar(ref s1, ref s2);
         // s1 = "dos", s2 = "uno"

         Console.WriteLine("x = " + x  + " y = " + y);
         Console.WriteLine("s1 = " + s1 + " s2 = " + s2);
      }
   }
}
```