

C#: PROGRAMACIÓN ORIENTADA A OBJETOS

Contenido

Otros libros del autor 3

Página web del autor y canal en Youtube 4

Sitio en GitHub 4

Licencia del software 4

Marcas registradas 4

Introducción..... 5

Iniciando con Programación Orientada a Objetos..... 6

Errores al tratar de acceder a atributos o métodos privados 8

Los atributos deben ser privados. Accediendo a ellos..... 9

 Forma reducida de los getters y setters..... 10

 Uso de los getters/setters 11

 Otro uso de los getters y setters 13

 Forma de inicializar los objetos llamando los setters..... 14

Dos variables refiriendo al mismo objeto..... 16

Polimorfismo 18

 Por número de parámetros 18

 Por tipo de parámetros..... 19

Constructores 20

 Constructor sin parámetros 20

 Constructor con parámetros 21

 Polimorfismo y constructores..... 22

 Usando el constructor para copiar objetos 24

 Un constructor puede llamar a otros métodos 26

Herencia 27

 Implementación en C# 27

 Clases abstractas y herencia 29

 Nivel de protección en los métodos y atributos 31

 Private 31

 Protected 33

 Public 35

 Herencia y métodos iguales en clase madre e hija 36

 Herencia y constructores 38

 Llamando a métodos de clases madres 40

 Evitar la herencia 42

Clases estáticas 43

 Métodos estáticos..... 44

 Constructor static..... 45

 Cuidado con el constructor static..... 47

Interface 49

 Interface múltiple..... 51

 Herencia e Interface 53

Enums..... 55

 Cambiando los valores de las constantes en enums..... 56

Structs 58

 Un struct se puede copiar fácilmente 59

 Métodos en un struct 60

 Structs y constructores..... 61

Clases parciales 62

Destrucción 65

Patrones de diseño 67

 Factory Method 67

 Abstract Factory 69

 Singleton 74

 Builder..... 75

 Adapter 80

Composite..... 83

Facade..... 85

Modelo Vista Controlador 87

Enlaces de interés sobre Programación Orientada a Objetos y C# 89

Otros libros del autor

Libro 15: "C#. Estructuras básicas de memoria.". En Colombia 2020. Págs. 60. Libro y código fuente descargable en: <https://github.com/ramsoftware/EstructuraBasicaMemoriaCSharp>

Libro 14: "Iniciando en C#". En Colombia 2020. Págs. 72. Libro y código fuente descargable en: <https://github.com/ramsoftware/C-Sharp-Iniciando>

Libro 13: "Algoritmos Genéticos". En Colombia 2020. Págs. 62. Libro y código fuente descargable en: <https://github.com/ramsoftware/LibroAlgoritmoGenetico2020>

Libro 12: "Redes Neuronales. Segunda Edición". En Colombia 2020. Págs. 108. Libro y código fuente descargable en: <https://github.com/ramsoftware/LibroRedNeuronal2020>

Libro 11: "Capacitándose en JavaScript". En Colombia 2020. Págs. 317. Libro y código fuente descargable en: <https://github.com/ramsoftware/JavaScript>

Libro 10: "Desarrollo de aplicaciones para Android usando MIT App Inventor 2". En Colombia 2016. Págs. 102. Ubicado en: <https://openlibra.com/es/book/desarrollo-de-aplicaciones-para-android-usando-mit-app-inventor-2>

Libro 9: "Redes Neuronales. Parte 1.". En Colombia 2016. Págs. 90. Libro descargable en: <https://openlibra.com/es/book/redes-neuronales-parte-1>

Libro 8: "Segunda parte de uso de algoritmos genéticos para la búsqueda de patrones". En Colombia 2015. Págs. 303. En publicación por la Universidad Libre – Cali.

Libro 7: "Desarrollo de un evaluador de expresiones algebraicas. **Versión 2.0.** C++, C#, Visual Basic .NET, Java, PHP, JavaScript y Object Pascal (Delphi)". En: Colombia 2013. Págs. 308. Ubicado en: <https://openlibra.com/es/book/evaluador-de-expresiones-algebraicas-ii>

Libro 6: "Un uso de algoritmos genéticos para la búsqueda de patrones". En Colombia 2013. En publicación por la Universidad Libre – Cali.

Libro 5: Desarrollo fácil y paso a paso de aplicaciones para Android usando MIT App Inventor. En Colombia 2013. Págs. 104. Estado: Obsoleto (No hay enlace).

Libro 4: "Desarrollo de un evaluador de expresiones algebraicas. C++, C#, Visual Basic .NET, Java, PHP, JavaScript y Object Pascal (Delphi)". En Colombia 2012. Págs. 308. Ubicado en: <https://openlibra.com/es/book/evaluador-de-expresiones-algebraicas>

Libro 3: "Simulación: Conceptos y Programación" En Colombia 2012. Págs. 81. Ubicado en: <https://openlibra.com/es/book/simulacion-conceptos-y-programacion>

Libro 2: "Desarrollo de videojuegos en 2D con Java y Microsoft XNA". En Colombia 2011. Págs. 260. Ubicado en: <https://openlibra.com/es/book/desarrollo-de-juegos-en-2d-usando-java-y-microsoft-xna> . ISBN: 978-958-8630-45-8

Libro 1: "Desarrollo de gráficos para PC, Web y dispositivos móviles" En Colombia 2009. ed.: Artes Gráficas Del Valle Editores Impresores Ltda. ISBN: 978-958-8308-95-1 v. 1 págs. 317

Artículo: "Programación Genética: La regresión simbólica". Entramado ISSN: 1900-3803 ed.: Universidad Libre Seccional Cali v.3 fasc.1 p.76 - 85, 2007

Página web del autor y canal en Youtube

Investigación sobre Vida Artificial: <http://darwin.50webs.com>

Canal en Youtube: <http://www.youtube.com/user/RafaelMorenoP> (dedicado principalmente al desarrollo en C#)

Sitio en GitHub

El código fuente se puede descargar en <https://github.com/ramsoftware/C-Sharp-POO>

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 2019 ® Enlace: <http://windows.microsoft.com/en-US/windows/home>

Introducción

El presente libro se aborda como el lenguaje de programación C# aborda el paradigma de Programación Orientada a Objetos, empezando por lo clásico como las clases, métodos, constructores, nivel de protección, instancias, herencia, polimorfismo y luego se aborda características propias del lenguaje como los structs (muy parecidos a las clases) y se finaliza con la implementación de algunos patrones de diseño.

El código fuente se puede descargar de GitHub en: <https://github.com/ramsoftware/C-Sharp-POO>

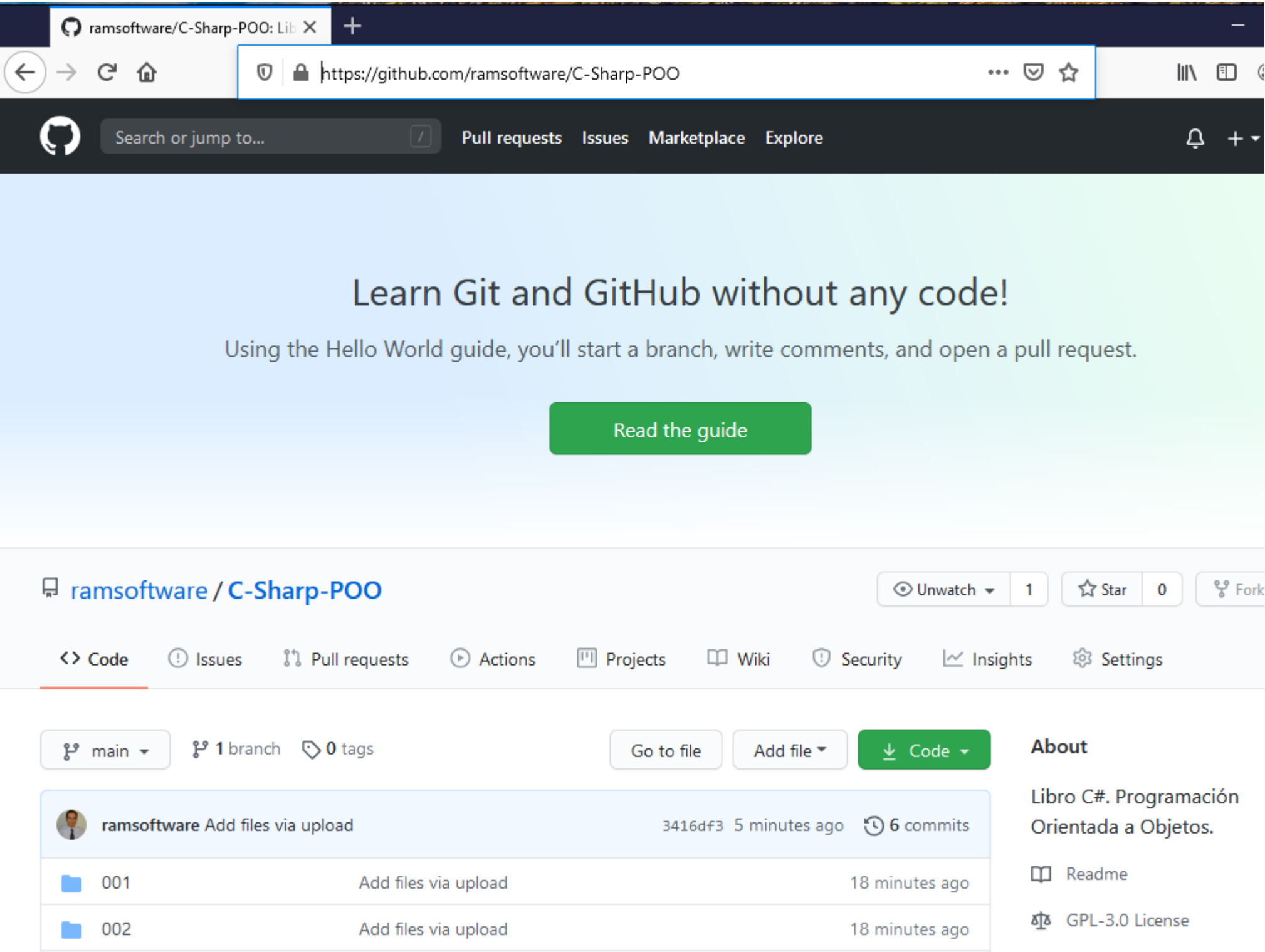


Ilustración 1: Sitio GitHub del libro y el código fuente: <https://github.com/ramsoftware/C-Sharp-POO>

Iniciando con Programación Orientada a Objetos

Al definir una clase en C#, puede 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.

Cada clase en su propio archivo con el mismo nombre de la clase.

Microsoft Visual Studio 2019 facilita el proceso. Primero es ir por la opción Proyecto -> Agregar clase...

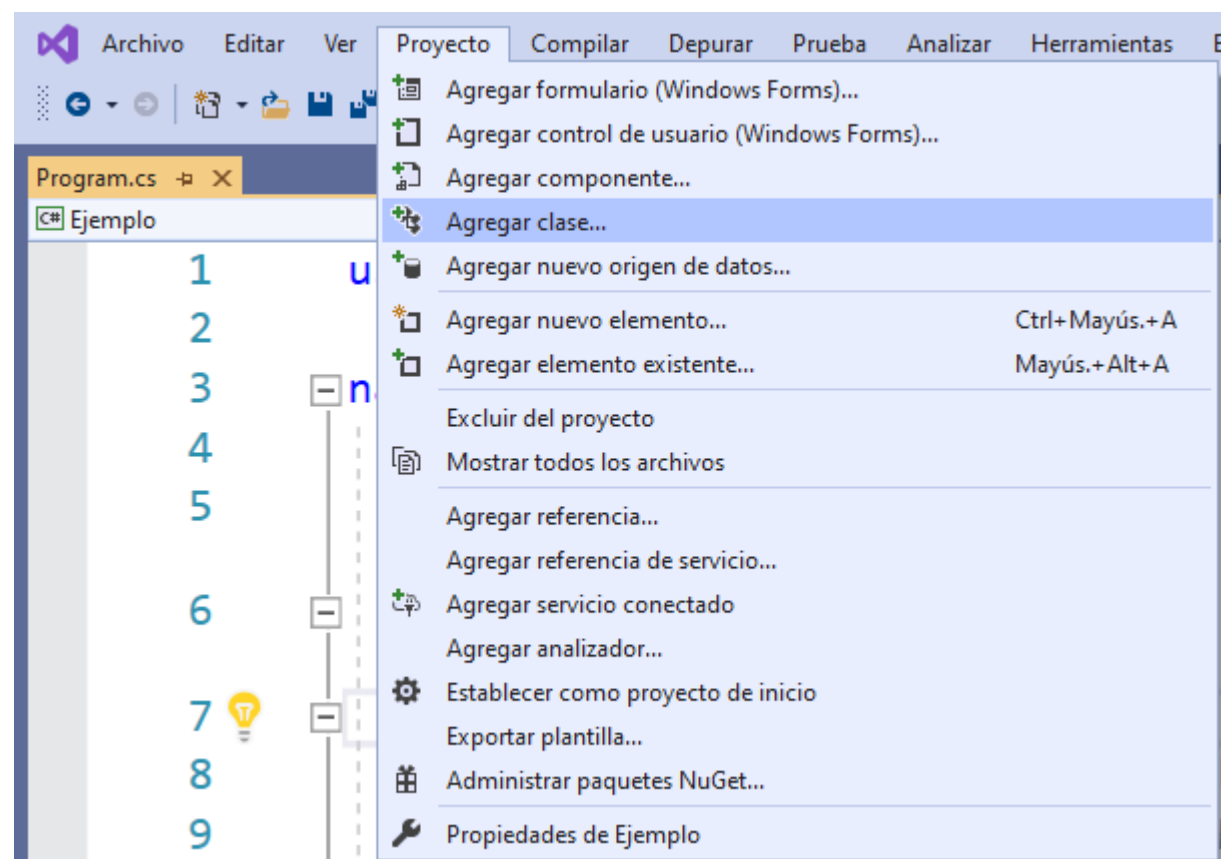


Ilustración 2: Crear una nueva clase

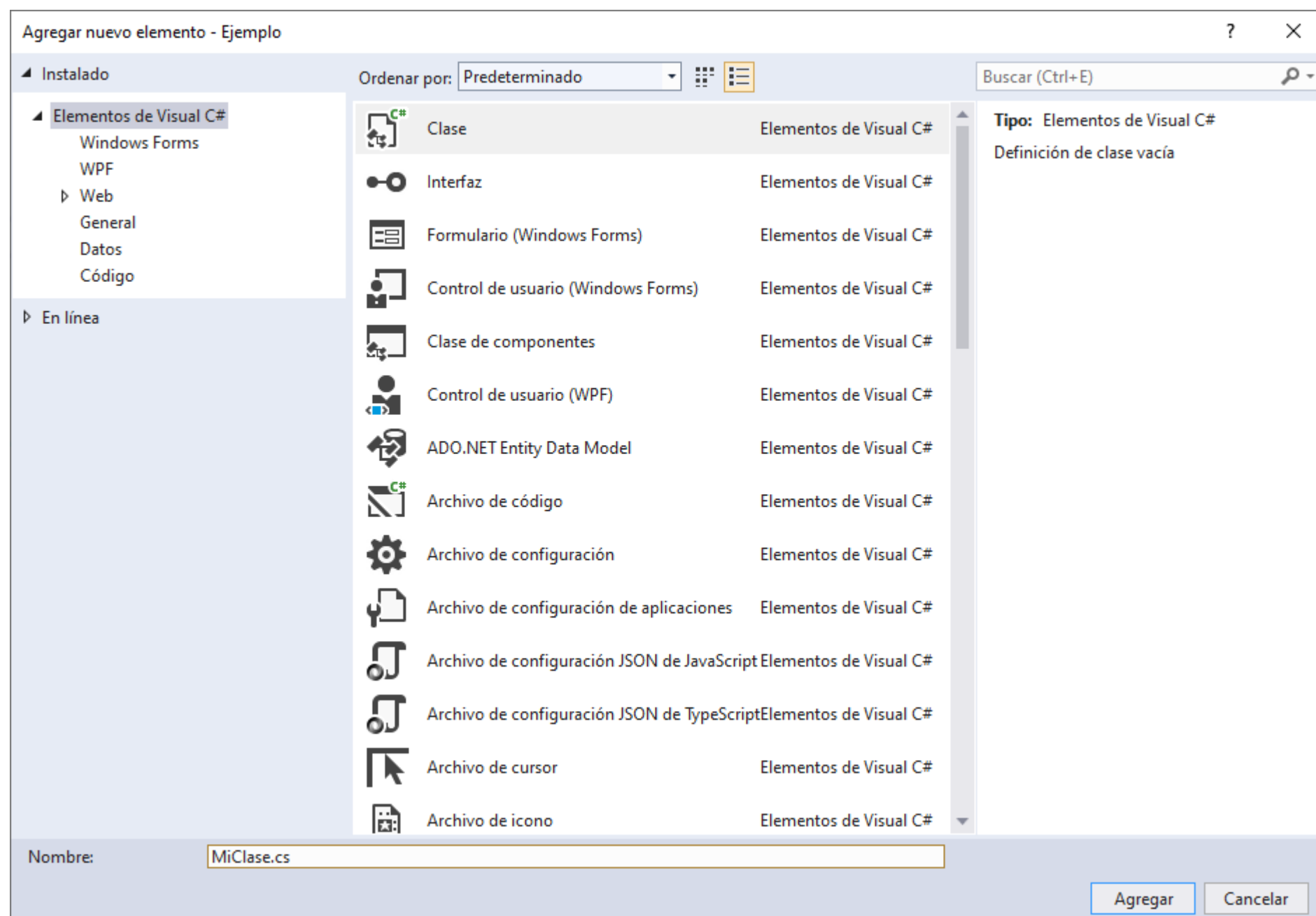


Ilustración 3: Selecciona Clase para crear una nueva y se presiona "Agregar"

Y se obtiene la plantilla de una clase.

```

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;

        //Atributos públicos (no recomendado)
        public int acumula;
        public char caracter;

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

        //Método público
        public double CalculaPromedio(double numA, double numB, double numC) {
            return (numA + numB + numC) / 3;
        }
    }
}

```

```

using System;

namespace Ejemplo {

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

            //Llama a un método público de MiClase
            double resultado = Objeto.CalculaPromedio(1, 7, 8);

            Console.WriteLine(resultado.ToString());
            Console.ReadKey();
        }
    }
}

```

Así se ve en el Explorador de Soluciones

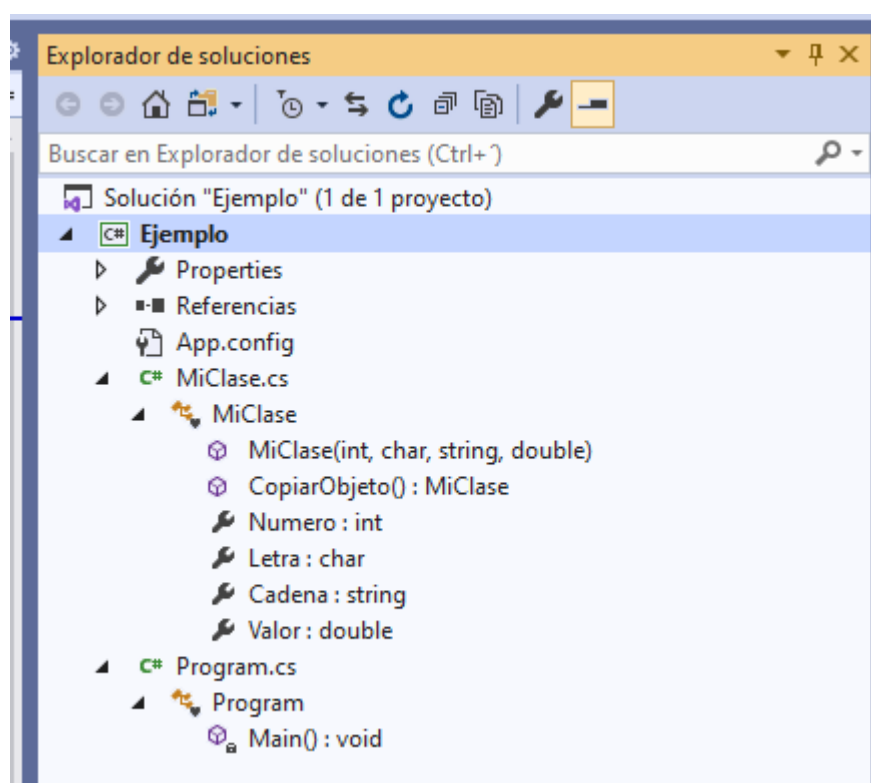


Ilustración 4: Explorador de soluciones

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.

```
//Intenta llamar a un método privado de MiClase
```

```
double resultado = Objeto.Maximo(1, 7, 8);
```



`double MiClase.Maximo(double numA, double numB, double numC)`

CS0122: 'MiClase.Maximo(double, double, double)' no es accesible debido a su nivel de protección

[Mostrar posibles correcciones](#) (Alt+Entrar o Ctrl+.)

```
//Intenta leer un atributo privado
```

```
string frase = Objeto.cadena;
```

Ilustración 5: Mensaje de error al intentar llamar a un método privado

```
//Intenta llamar a un método privado de MiClase
```

```
double resultado = Objeto.Maximo(1, 7, 8);
```

```
//Intenta leer un atributo privado
```

```
string frase = Objeto.cadena;
```

```
Console.WriteLine(resultado);
```

```
Console.WriteLine(frase.ToString());
```

```
Console.ReadKey();
```

`class System.String`

Representa texto como una secuencia de unidades de código UTF-16. Para examinar el código fuente de .NET Framework para este tipo, consulte el Reference Source.

CS0122: 'MiClase.cadena' no es accesible debido a su nivel de protección

Ilustración 6: Mensaje de error al intentar leer un atributo privado

Los atributos deben ser privados. Accediendo a ellos.

La recomendación y casi estándar internacional 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. En C# hay varias formas de hacer eso:

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

Carpeta 002. MiClase.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; }
    }
}
```

Y se usa de la siguiente forma para dar valores a ese atributo: Objeto.Metodo = valor;

Y para leer los valores a ese atributo: variable = Objeto.Metodo;

Carpeta 002. Program.cs

```
using System;

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

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

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

            Console.ReadKey();
        }
    }
}
```

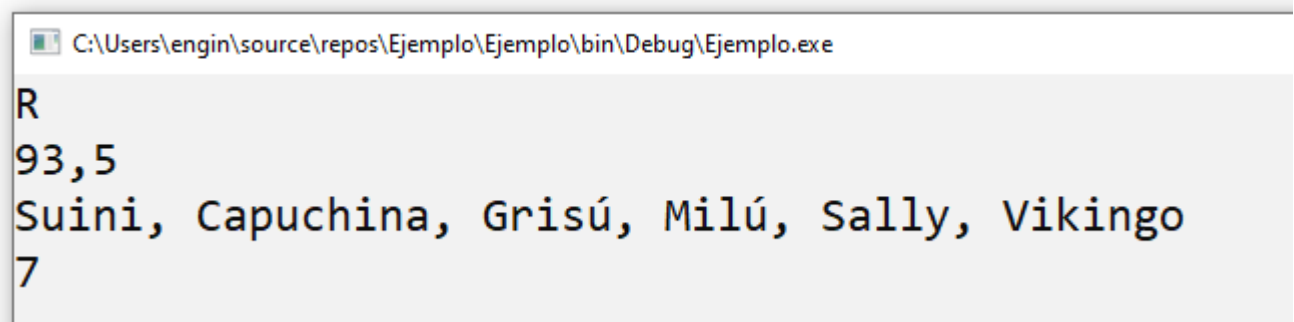


Ilustración 7: Uso de getters y setters para acceder a los atributos privados

Forma reducida de los getters y setters

Carpeta 003. MiClase.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; }  
    }  
}
```

Esto se debe tomar como una abreviación.

Carpeta 003. Program.cs

```
using System;  
  
namespace Ejemplo {  
    //Inicia la aplicación aquí  
    class Program {  
        static void Main() {  
            //Instancia o crea un objeto de MiClase  
            MiClase Objeto = new MiClase();  
  
            //Llama los setters  
            Objeto.Cadena = "Suini, Capuchina, Grisú, Milú, Sally, Vikingo";  
            Objeto.Numero = 7;  
            Objeto.Letra = 'R';  
            Objeto.Valor = 93.5;  
  
            //Usa los getters  
            Console.WriteLine(Objeto.Letra.ToString());  
            Console.WriteLine(Objeto.Valor.ToString());  
            Console.WriteLine(Objeto.Cadena);  
            Console.WriteLine(Objeto.Numero.ToString());  
  
            Console.ReadKey();  
        }  
    }  
}
```

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

Nota: En la documentación de C# los atributos son llamados también campos (fields). Y los getters/setters son llamados propiedades.

Uso de los getters/setters

Para hacer auditoría

Carpeta 004. MiClase.cs

```
using System;

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: " + DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt"));
                return numero;
            }
            set {
                Console.WriteLine("Cambia numero: " + DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt"));
                numero = value;
            }
        }

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

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

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

```

using System;

namespace Ejemplo {

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

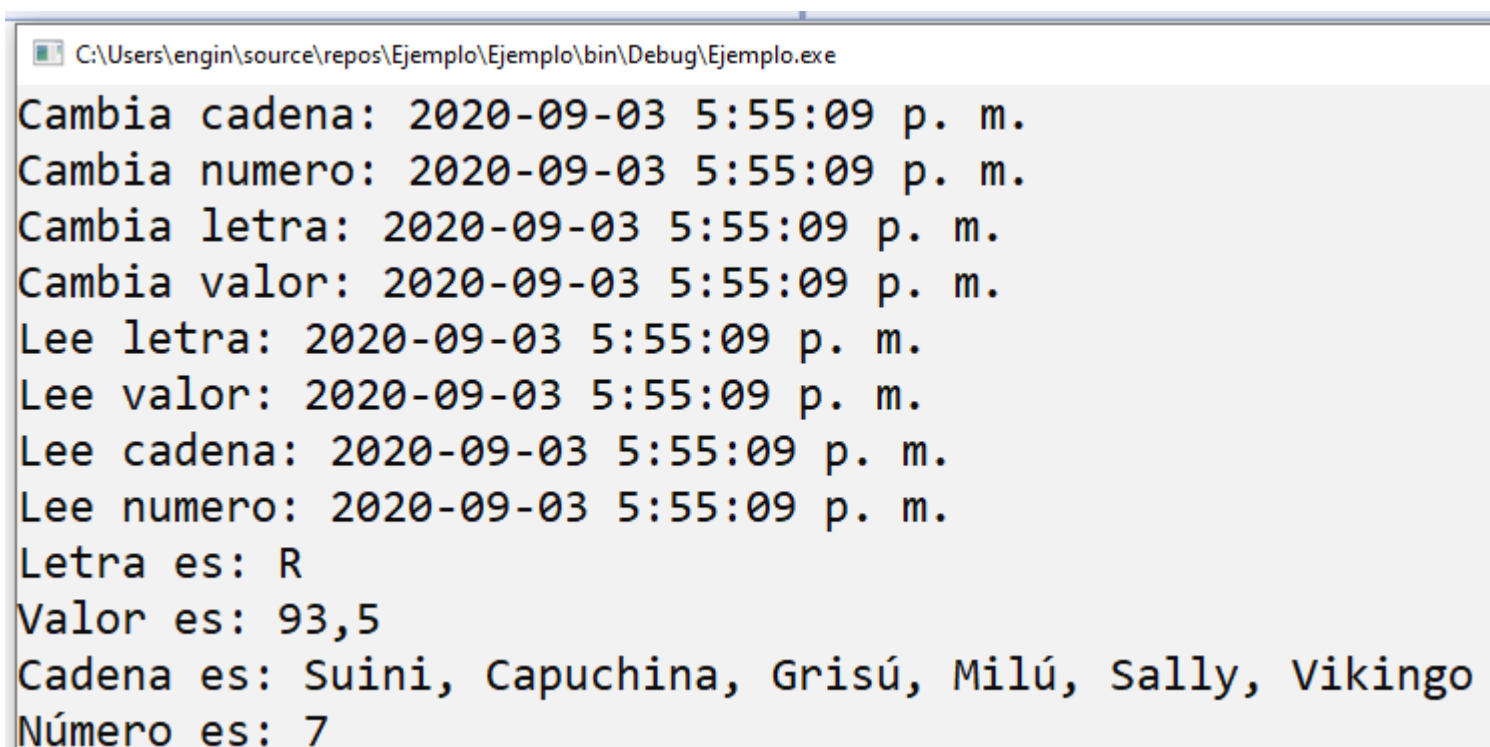
            //Llama los setters
            Objeto.Cadena = "Suini, Capuchina, Grisú, Milú, Sally, Vikingo";
            Objeto.Numero = 7;
            Objeto.Letra = 'R';
            Objeto.Valor = 93.5;

            //Usa los getters
            char una letra = Objeto.Letra;
            double unvalor = Objeto.Valor;
            string unacadena = Objeto.Cadena;
            int unnumero = Objeto.Numero;

            Console.WriteLine("Letra es: " + una letra.ToString());
            Console.WriteLine("Valor es: " + unvalor.ToString());
            Console.WriteLine("Cadena es: " + unacadena);
            Console.WriteLine("Número es: " + unnumero.ToString());

            Console.ReadKey();
        }
    }
}

```



C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe
 Cambia cadena: 2020-09-03 5:55:09 p. m.
 Cambia numero: 2020-09-03 5:55:09 p. m.
 Cambia letra: 2020-09-03 5:55:09 p. m.
 Cambia valor: 2020-09-03 5:55:09 p. m.
 Lee letra: 2020-09-03 5:55:09 p. m.
 Lee valor: 2020-09-03 5:55:09 p. m.
 Lee cadena: 2020-09-03 5:55:09 p. m.
 Lee numero: 2020-09-03 5:55:09 p. m.
 Letra es: R
 Valor es: 93,5
 Cadena es: Suini, Capuchina, Grisú, Milú, Sally, Vikingo
 Número es: 7

Ilustración 8: Un uso de los getters y setters para hacer auditoría

Otro uso de los getters y setters

Para validar el dato de entrada

Carpeta 005. MiClase.cs

```
using System;

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("La edad no debe ser negativa");
                else
                    edad = value;
            }
        }
    }
}
```

Carpeta 005. Program.cs

```
using System;

namespace Ejemplo {

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

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

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

            Console.ReadKey();
        }
    }
}
```

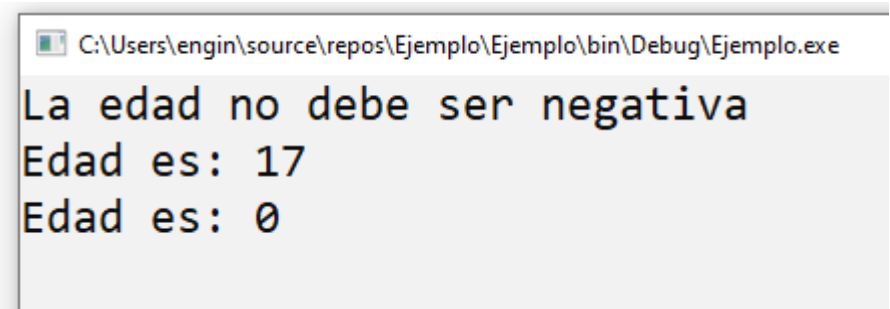


Ilustración 9: El setter valida el dato de entrada.

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.

```
using System;

namespace Ejemplo {
    //Esta es una clase propia con sus atributos y métodos (encapsulación)
    class MiClase {
        //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 de un atributo
        public int Numero {
            get {
                Console.WriteLine("Lee numero: " + DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt"));
                return numero;
            }
            set {
                Console.WriteLine("Cambia numero: " + DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt"));
                numero = value;
            }
        }

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

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

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

Y ahora una nueva forma de llamar a los setters.

```

using System;

namespace Ejemplo {

    //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

            };

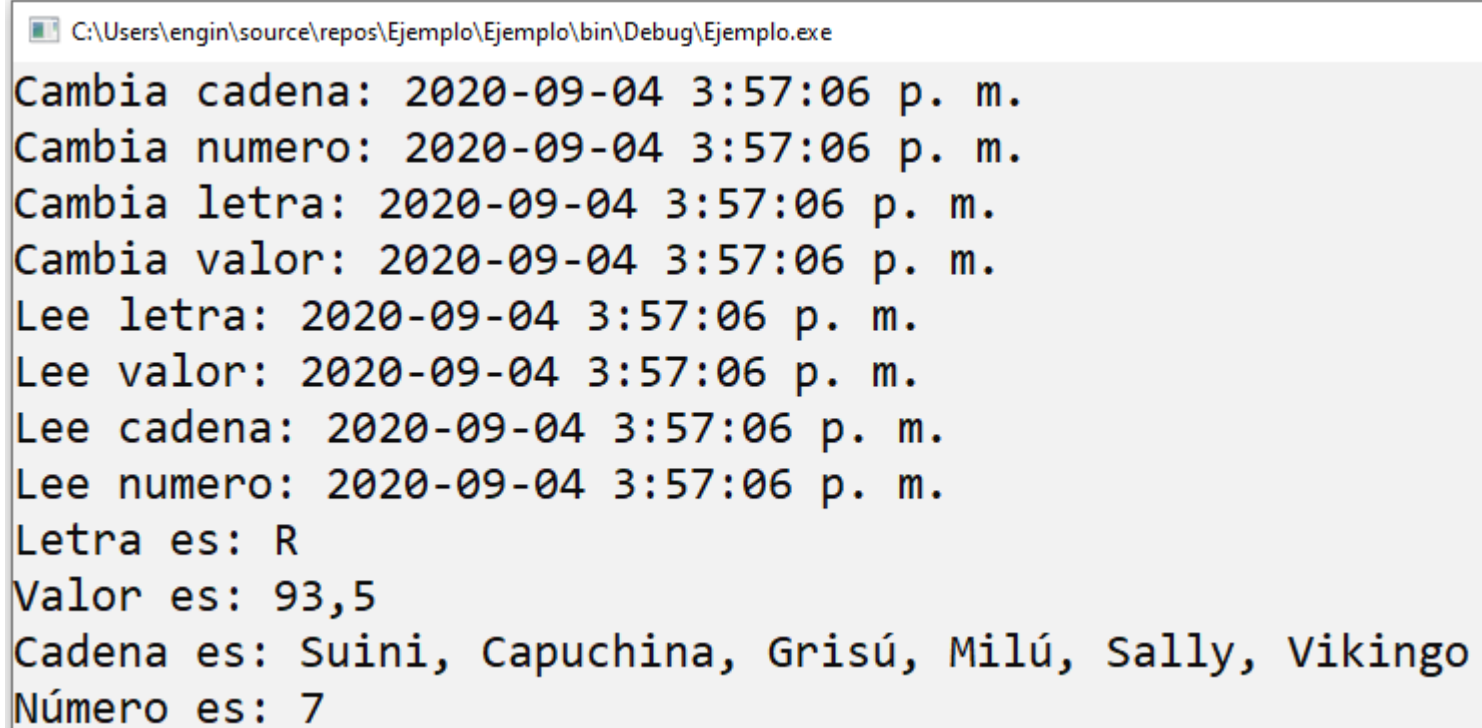
            //Usa los getters
            char unaletra = Objeto.Letra;
            double unvalor = Objeto.Valor;
            string unacadena = Objeto.Cadena;
            int unnumero = Objeto.Numero;

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

            Console.ReadKey();

        }
    }
}

```



```

C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe
Cambia cadena: 2020-09-04 3:57:06 p. m.
Cambia numero: 2020-09-04 3:57:06 p. m.
Cambia letra: 2020-09-04 3:57:06 p. m.
Cambia valor: 2020-09-04 3:57:06 p. m.
Lee letra: 2020-09-04 3:57:06 p. m.
Lee valor: 2020-09-04 3:57:06 p. m.
Lee cadena: 2020-09-04 3:57:06 p. m.
Lee numero: 2020-09-04 3:57:06 p. m.
Letra es: R
Valor es: 93,5
Cadena es: Suini, Capuchina, Grisú, Milú, Sally, Vikingo
Número es: 7

```

Ilustración 10: Otra forma de llamar a los setters

Dos variables refiriendo al mismo objeto

Carpeta 007. MiClase.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; }
    }
}
```

Ahora se instancia esa clase con la variable llamada *Mascotas* y se crea una variable de tipo *MiClase* pero que NO instancia llamada *otraVariable*, luego es una variable vacía.

Carpeta 007. Program.cs

```
using System;

namespace Ejemplo {

    //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, Sally, Vikingo",
                Numero = 7,
                Letra = 'R',
                Valor = 93.5
            };

            //Crea una variable de tipo MiClase
            MiClase otraVariable;

            //Asigna el primer objeto a esa variable
            otraVariable = 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 es: " + Mascotas.Letra.ToString());
            Console.WriteLine("Valor en Mascotas es: " + Mascotas.Valor.ToString());
            Console.WriteLine("Letra en otraVariable es: " + otraVariable.Letra.ToString());
            Console.WriteLine("Valor en otraVariable es: " + otraVariable.Valor.ToString());

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

            Console.ReadKey();
        }
    }
}
```

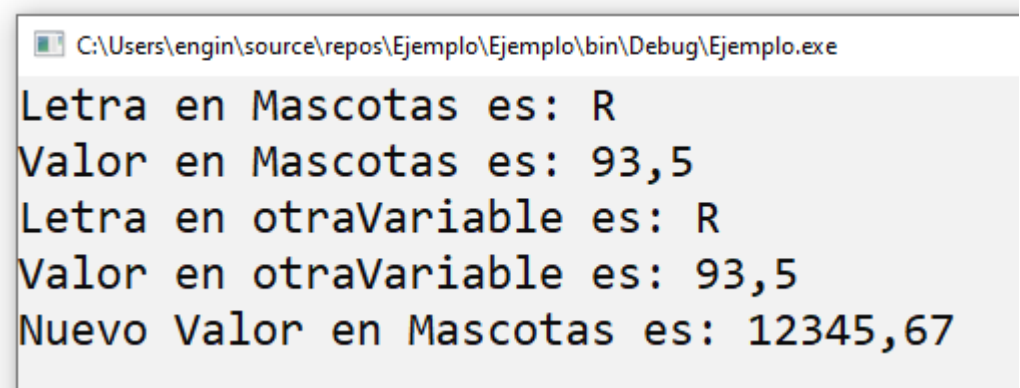


Ilustración 11: Mismo objeto referenciado por dos variables distintas

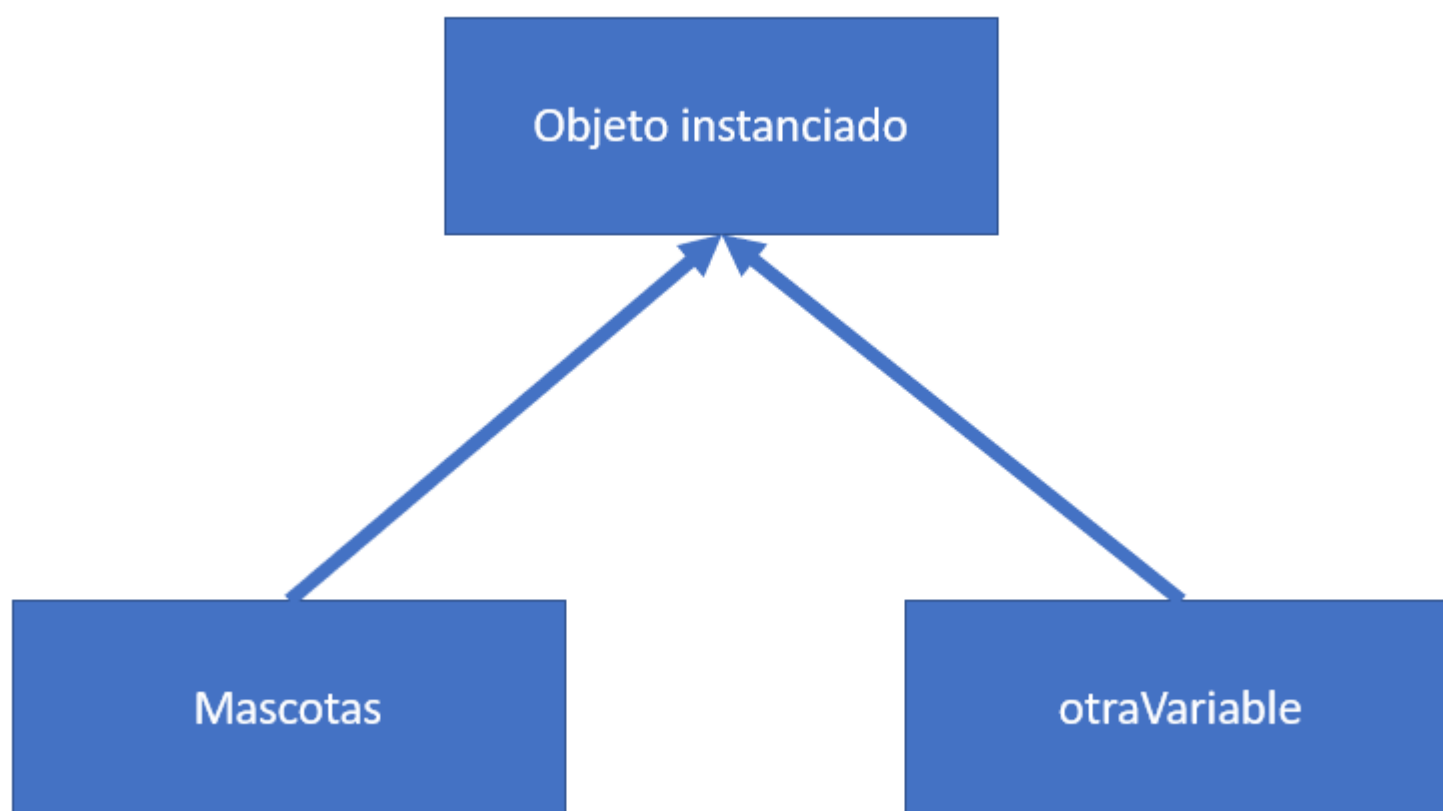


Ilustración 12: Ambas variables apuntan al mismo objeto instanciado

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

Polimorfismo

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

Carpeta 008. Geometria.cs

```
using System;

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));
        }
    }
}
```

Y así es su uso

Carpeta 008. Program.cs

```
using System;

namespace Ejemplo {

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

            //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.ToString());
            Console.WriteLine("Área del triángulo: " + areaTriangulo.ToString());
            Console.WriteLine("Área del rectángulo: " + areaRectangulo.ToString());

            Console.ReadKey();
        }
    }
}
```

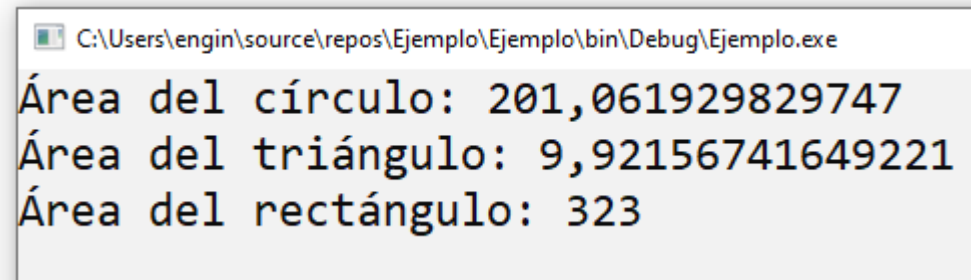


Ilustración 13: Polimorfismo

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.

Carpeta 009. MiClase.cs

```
using System;

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");
        }
    }
}
```

Carpeta 009. Program.cs

```
using System;

namespace Ejemplo {

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

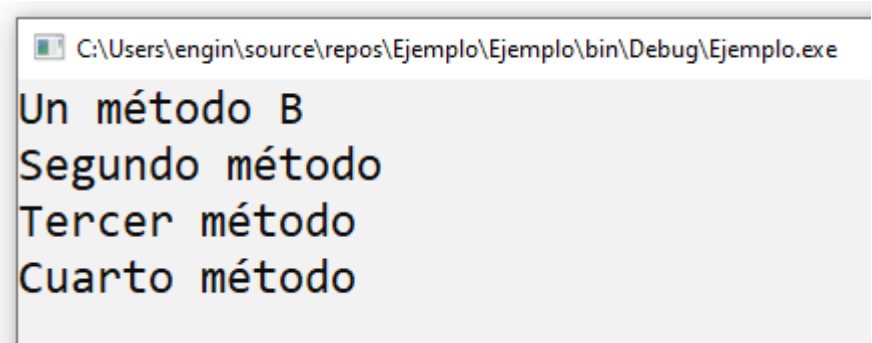


Ilustración 14: Polimorfismo dependiendo del tipo de parámetro

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

Carpeta 010. MiClase.cs

```
using System;

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

Carpeta 010. Program.cs

```
using System;

namespace Ejemplo {

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

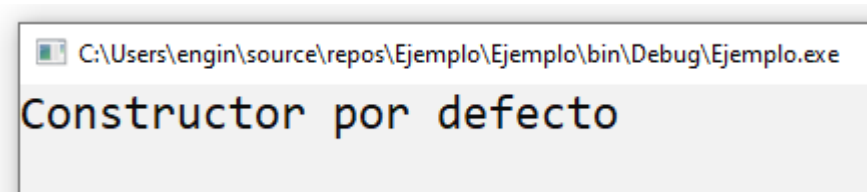


Ilustración 15: Constructor sin parámetros

Constructor con parámetros

Se envía los datos del objeto al instanciarlo

Carpeta 011. MiClase.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) {
            this.Numero = Numero; //Se asigna así this.atributo = valor parámetro
            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; }
    }
}
```

Carpeta 011. Program.cs

```
using System;

namespace Ejemplo {

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

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

            Console.ReadKey();
        }
    }
}
```

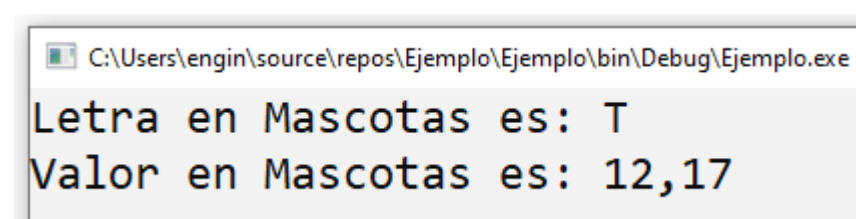


Ilustración 16: Los constructores

Polimorfismo y constructores

A continuación, una clase con varios constructores:

Carpeta 012. MiClase.cs

```
using System;

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");
        }
    }
}
```

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

Carpeta 012. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    class Program {
        static void Main() {
            //Instancia o crea un objeto
            MiClase objetoA = new MiClase();
            MiClase objetoB = new MiClase(48);
            MiClase objetoC = new MiClase(1972.06, 26);
            MiClase objetoD = new MiClase("Ramp", 48);
            MiClase objetoE = new MiClase("Moreno Parra", 1683.29, 29);
            Console.ReadKey();
        }
    }
}
```

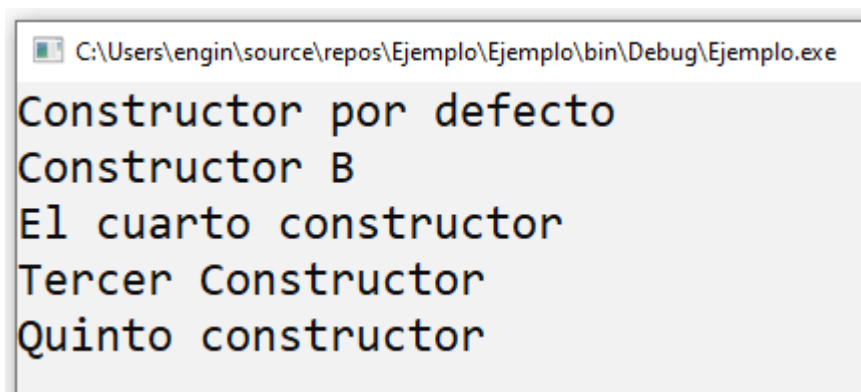


Ilustración 17: Constructores y polimorfismo

Usando el constructor para copiar objetos

Anteriormente se mostró que la asignación de una variable objeto a otro termina es 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-icloneable>). 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.

Carpeta 013. MiClase.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) {
            this.Numero = Numero; //Se asigna así this.atributo = valor parámetro
            this.Letra = Letra;
            this.Cadena = Cadena;
            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; }
    }
}
```

Y aquí se hace uso del método de copiar:

Carpeta 013. Program.cs

```
using System;

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

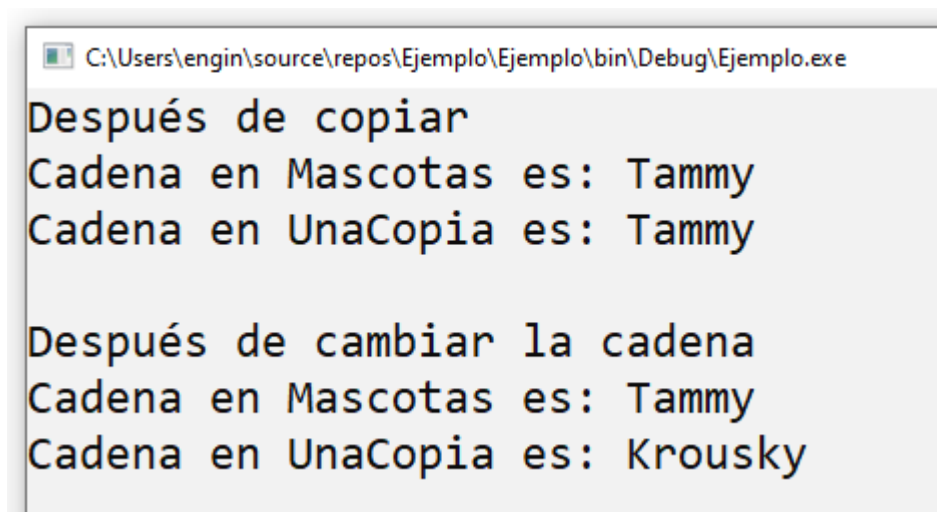
            //Hace una copia de ese objeto
            MiClase UnaCopia = Mascotas.CopiarObjeto();

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

            Console.ReadKey();
        }
    }
}
```



```
C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe
Después de copiar
Cadena en Mascotas es: Tammy
Cadena en UnaCopia es: Tammy

Después de cambiar la cadena
Cadena en Mascotas es: Tammy
Cadena en UnaCopia es: Krousky
```

Ilustración 18: Copiar un objeto en otro

Un constructor puede llamar a otros métodos

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

Carpeta 014. MiClase.cs

```
using System;

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

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

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

Carpeta 014. Program.cs

```
using System;

namespace Ejemplo {

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

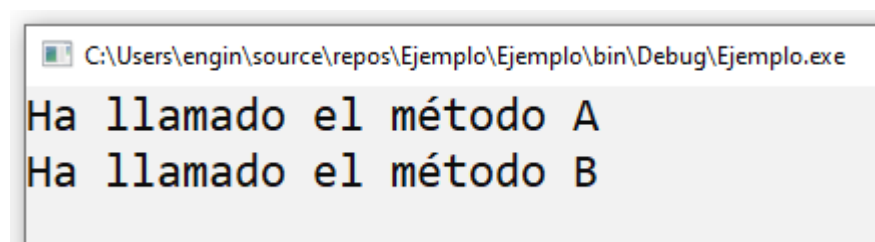


Ilustración 19: El constructor llama a otros métodos

Herencia

Implementación en C#

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

Carpeta 015. Mascota.cs

```
using System;

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
        public string Propietario { get; set; } //Nombre del propietario
        public string Telefono { get; set; } //Teléfono del propietario
        public string Correo { get; set; } //Correo electrónico del propietario
        public double Peso { get; set; }
        public int ColorOjos { get; set; } //0. Azul, 1. Verde, 2. Dorado, 3. Dispar
        public int EsperanzaVidaMinimo { get; set; }
        public int EsperanzaVidaMaximo { get; set; }
        public int NecesidadAtencion { get; set; } //0. Baja, 1. Media, 2. Alta
        public string Raza { get; set; }
    }
}
```

Carpeta 015. Gato.cs

```
namespace Ejemplo {
    //Gato hereda de Mascota
    class Gato:Mascota {
        public string PatronPelo { get; set; }
        public int TendenciaPerderPelo { get; set; }

        public string ReconocimientoCFA { get; set; } //Asociación de Criadores de Gatos
        public string ReconocimientoACFA { get; set; } //Asociación Americana de Criadores de Gatos
        public string ReconocimientoFIFE { get; set; } //Fédération Internationale Féline
        public string ReconocimientoTICA { get; set; } //Asociación Internacional de Gatos
    }
}
```

Carpeta 015. Perro.cs

```
namespace Ejemplo {
    //Perro hereda de Mascota
    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; }
        public int TendenciaBabear { get; set; } //0. Ninguna, 1. Baja, 2. Moderada
        public int TendenciaRoncar { get; set; }
        public int TendenciaLadrar { get; set; }
        public int TendenciaExcavar { get; set; }
    }
}
```

```
using System;

namespace Ejemplo {

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

            //Da valores a la instancia de mascota
            objMascota.Correo = "enginelif@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;

            Console.ReadKey();
        }
    }
}
```

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.

Carpeta 016. Mascota.cs

```
using System;

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; }
        public char Sexo { get; set; } //Macho, Hembra
        public string Propietario { get; set; } //Nombre del propietario
        public string Telefono { get; set; } //Teléfono del propietario
        public string Correo { get; set; } //Correo electrónico del propietario
        public double Peso { get; set; }
        public int ColorOjos { get; set; } //0. Azul, 1. Verde, 2. Dorado, 3. Dispar
        public int EsperanzaVidaMinimo { get; set; }
        public int EsperanzaVidaMaximo { get; set; }
        public int NecesidadAtencion { get; set; } //0. Baja, 1. Media, 2. Alta
        public string Raza { get; set; }
    }
}
```

Esta clase abstracta puede heredar

Carpeta 016. Gato.cs

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

        public string ReconocimientoCFA { get; set; } //Asociación de Criadores de Gatos
        public string ReconocimientoACFA { get; set; } //Asociación Americana de Criadores de Gatos
        public string ReconocimientoFIFe { get; set; } //Fédération Internationale Féline
        public string ReconocimientoTICA { get; set; } //Asociación Internacional de Gatos
    }
}
```

Carpeta 016. Perro.cs

```
namespace Ejemplo {
    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; }
        public int TendenciaBabear { get; set; } //0. Ninguna, 1. Baja, 2. Moderada
        public int TendenciaRoncar { get; set; }
        public int TendenciaLadrar { get; set; }
        public int TendenciaExcavar { get; set; }
    }
}
```

Al intentar instanciar una clase abstracta se obtiene el siguiente mensaje de error:

```
using System;

namespace Ejemplo {

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


            //Da valores a la instancia de mascota
            objMascota.Correo = "enginelifemail.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;

            Console.ReadKey();
        }
    }
}
```

```
static void Main() {
    Mascota objMascota = new Mascota();
    Gato objGato = new Gato();
    Perro objPerro = new Perro();
}
```

 class Ejemplo.Mascota

CS0144: No se puede crear una instancia de la clase o interfaz abstracta 'Mascota'

Ilustración 20: Error al tratar de instanciar una clase abstracta

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.

Carpeta 017. Mascota.cs

```
namespace Ejemplo {
    //Clase madre con atributos privados
    class Mascota {
        private string Nombre { get; set; }
        private char Sexo { get; set; } //Macho, Hembra
        private string Propietario { get; set; } //Nombre del propietario
    }
}
```

En este ejemplo la clase hija tratará de acceder a métodos privados de la clase madre. Generará un mensaje de error.

Carpeta 017. Gato.cs

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

En este ejemplo la clase hija tratará de acceder a métodos privados de la clase madre. Generará un mensaje de error.

Carpeta 017. Perro.cs

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

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

Carpeta 017. Program.cs

```
using System;

namespace Ejemplo {

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


            //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");

            Console.ReadKey();
        }
    }
}
```


Si se intenta usar un atributo o método privado de la clase madre al interior de la clase hija, obtenemos un mensaje de error en tiempo de compilación.

```
this.Nombre = Nombre;  
this.Se  
this.Pr
```

 **class System.String**

Representa texto como una secuencia de unidades de código UTF-16. Para examinar el código fuente de .NET Framework para este tipo, consulte el Reference Source.

CS0122: 'Mascota.Nombre' no es accesible debido a su nivel de protección

Ilustración 21: Mensaje de error al tratar de acceder a un método privado de la clase madre

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.

Carpeta 018. Mascota.cs

```
namespace Ejemplo {
    //Clase madre con atributos privados
    class Mascota {
        protected string Nombre { get; set; }
        protected char Sexo { get; set; } //Macho, Hembra
        protected string Propietario { get; set; } //Nombre del propietario
    }
}
```

Carpeta 018. Gato.cs

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

Carpeta 018. Perro.cs

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

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

Carpeta 018. Program.cs

```
using System;

namespace Ejemplo {

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

            //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";

            Console.ReadKey();
        }
    }
}
```

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

```
Console.ReadKey();
```

 **class** System.String

Representa texto como una secuencia de unidades de código UTF-16. Para obtener más información sobre este tipo, consulte el Reference Source.

CS0122: 'Mascota.Nombre' no es accesible debido a su nivel de protección

Ilustración 22: Las instancias no pueden acceder a los métodos protected

Public

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

Carpeta 019. Mascota.cs

```
namespace Ejemplo {
    //Clase madre con atributos privados
    class Mascota {
        public string Nombre { get; set; }
        public char Sexo { get; set; } //Macho, Hembra
        public string Propietario { get; set; } //Nombre del propietario
    }
}
```

Carpeta 019. Gato.cs

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

Carpeta 019. Perro.cs

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

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

Carpeta 019. Program.cs

```
using System;

namespace Ejemplo {

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

            //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";

            Console.ReadKey();
        }
    }
}
```

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?

Carpeta 020. Madre.cs

```
using System;

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

Carpeta 020. Hija.cs

```
using System;

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

Carpeta 020. Program.cs

```
using System;

namespace Ejemplo {

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

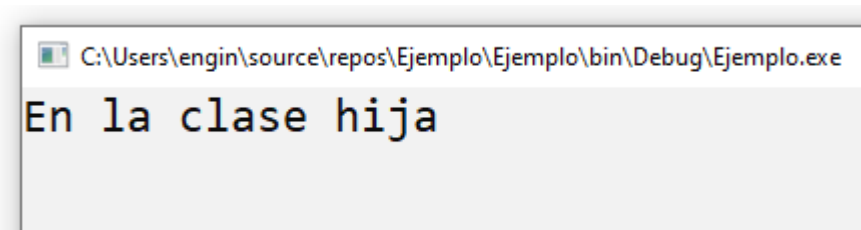


Ilustración 23: Ejecuta el procedimiento de la clase hija

Se ejecuta el método de la clase hija. El compilador lo advierte.

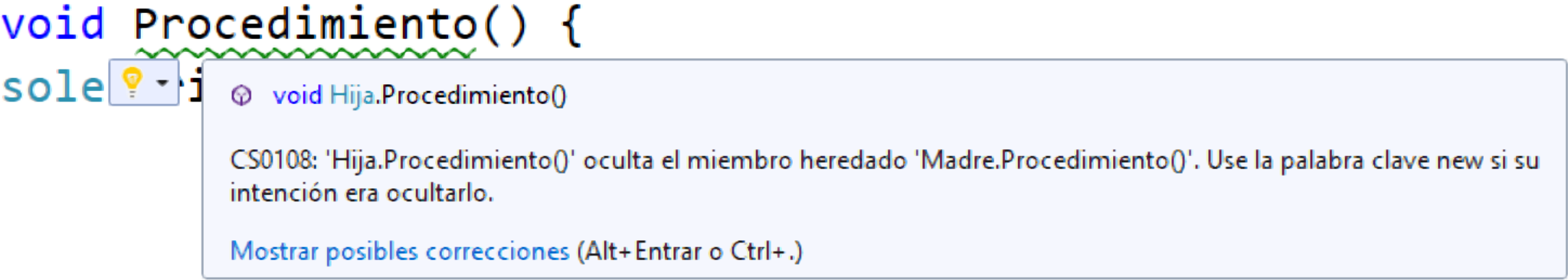


Ilustración 24: Compilador advierte que el método de la clase Hija oculta el método de la clase Madre

Se hace explicito ese mismo comportamiento con la palabra "new". Así no hay advertencia.

```
using System;

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

Herencia y constructores

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

Carpeta 021. Abuela.cs

```
using System;

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

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

Carpeta 021. Madre.cs

```
using System;

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

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

Carpeta 021. Hija.cs

```
using System;

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

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

```

using System;

namespace Ejemplo {

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

            //Ejecuta método
            objHija.Mostrar();

            Console.ReadKey();
        }
    }
}

```

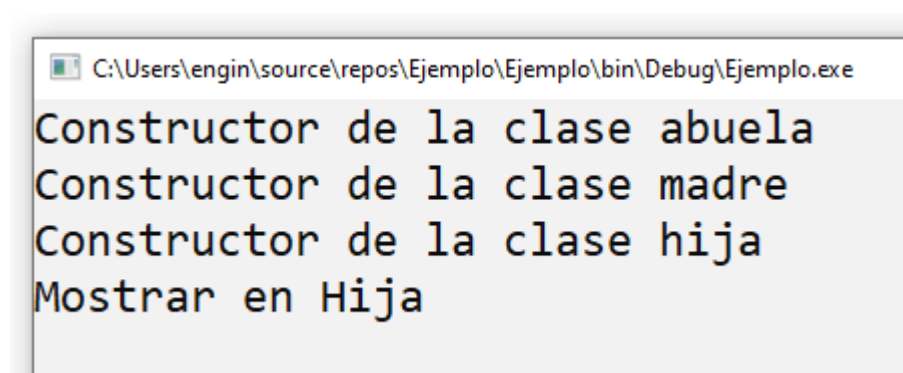


Ilustración 25: Ejecuta los constructores de la clase abuela, madre y luego hija.

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.

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

Carpeta 022. Abuela.cs

```
using System;

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

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

Carpeta 022. Madre.cs

```
using System;

namespace Ejemplo {
    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");
        }
    }
}
```

Carpeta 022. Hija.cs

```
using System;

namespace Ejemplo {
    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");
        }
    }
}
```

Carpeta 022. Program.cs

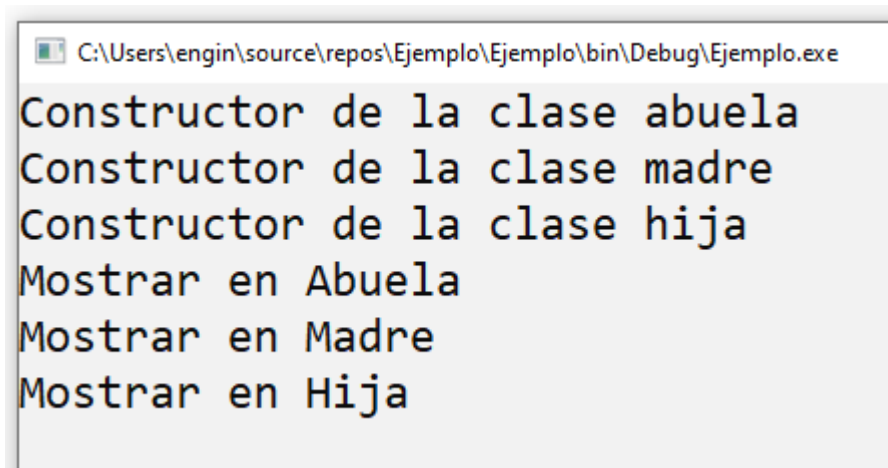
```
using System;

namespace Ejemplo {

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

            //Ejecuta método
            objHija.Mostrar();

            Console.ReadKey();
        }
    }
}
```



```
C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe
Constructor de la clase abuela
Constructor de la clase madre
Constructor de la clase hija
Mostrar en Abuela
Mostrar en Madre
Mostrar en Hija
```

Ilustración 26: Usando "base" para llamar los métodos de la clase madre

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.

Carpeta 023. Madre.cs

```
using System;

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

Carpeta 023. Hija.cs

```
using System;

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

Carpeta 023. Program.cs

```
using System;

namespace Ejemplo {

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

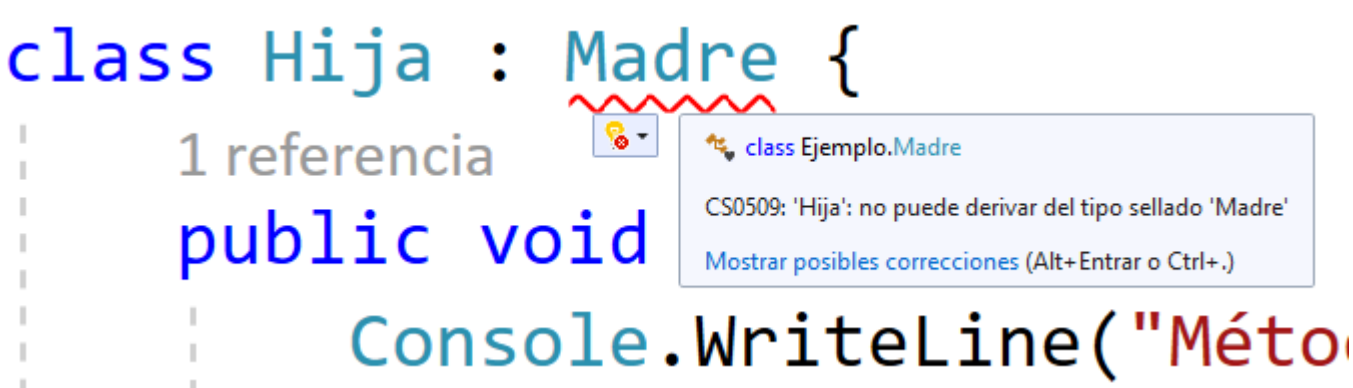


Ilustración 27: Con "sealed" la clase no permite heredar

Clases estáticas

Una clase estática no requiere instanciarse para ser usada.

Carpeta 024. Geometria.cs

```
using System;

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;
        }
    }
}
```

Carpeta 024. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    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.ToString());

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

            Console.ReadKey();
        }
    }
}
```

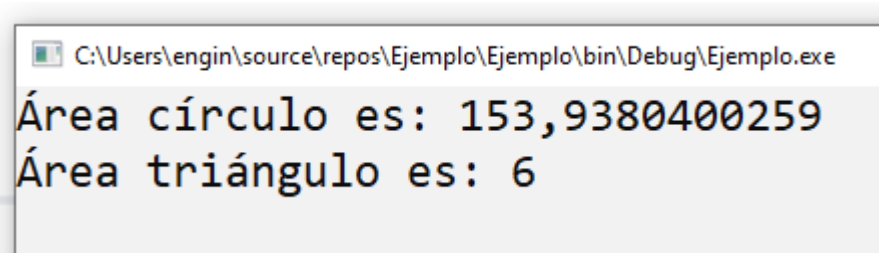


Ilustración 28: Clases estáticas

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.

Carpeta 025. Geometria.cs

```
using System;

namespace Ejemplo {
    //Clase con métodos estáticos
    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);
        }
    }
}
```

Carpeta 025. Program.cs

```
using System;

namespace Ejemplo {

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

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

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

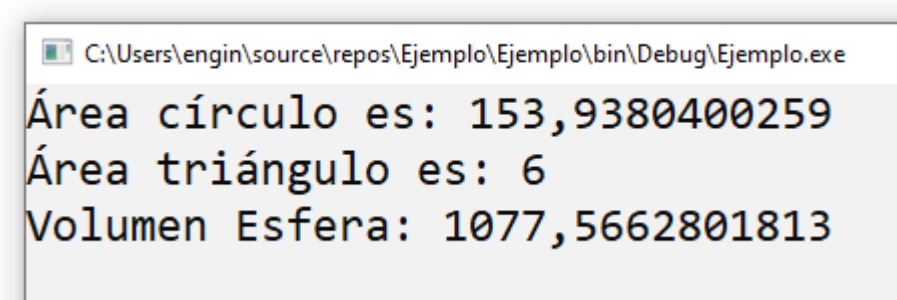


Ilustración 29: Métodos estáticos

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.

Carpeta 026. Geometria.cs

```
using System;

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);
        }
    }
}
```

```

using System;

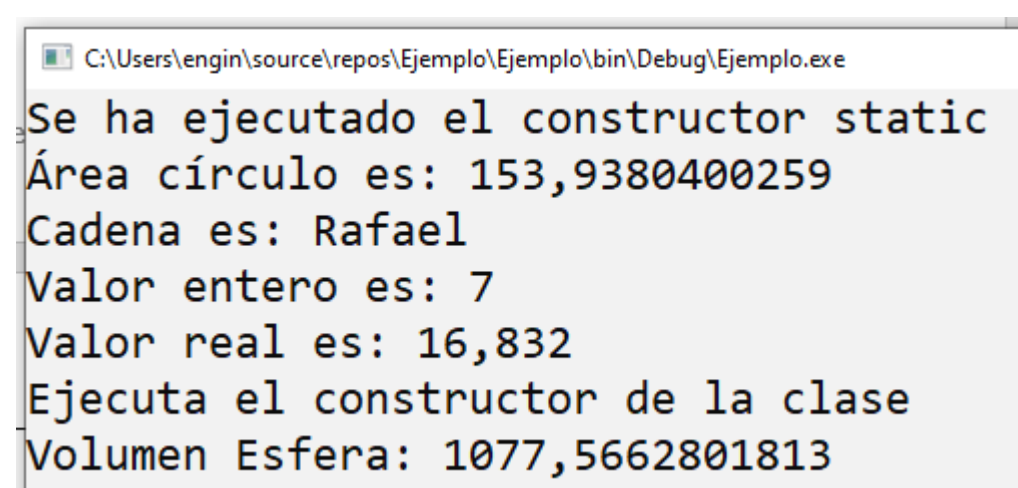
namespace Ejemplo {

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

            //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.ToString());
            Console.ReadKey();
        }
    }
}

```



```

C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe
Se ha ejecutado el constructor static
Área círculo es: 153,9380400259
Cadena es: Rafael
Valor entero es: 7
Valor real es: 16,832
Ejecuta el constructor de la clase
Volumen Esfera: 1077,5662801813

```

Ilustración 30: 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.

Carpeta 027. Geometria.cs

```
using System;

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);
        }
    }
}
```



```

using System;

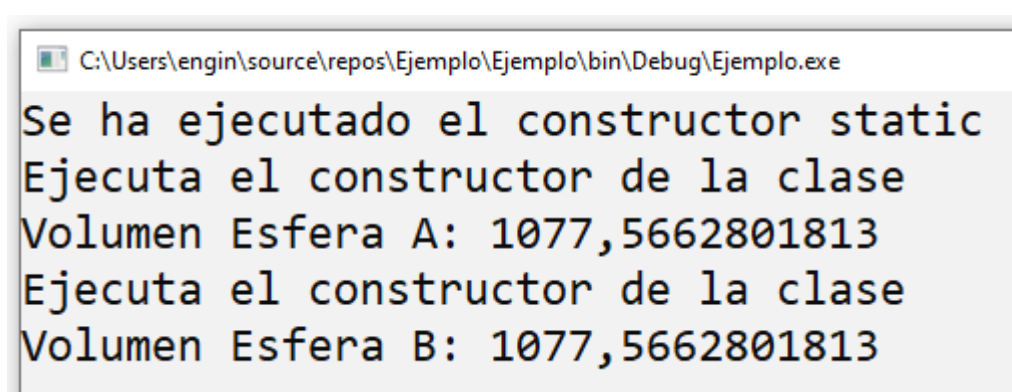
namespace Ejemplo {

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

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

            Console.ReadKey();
        }
    }
}

```



```

C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe
Se ha ejecutado el constructor static
Ejecuta el constructor de la clase
Volumen Esfera A: 1077,5662801813
Ejecuta el constructor de la clase
Volumen Esfera B: 1077,5662801813

```

Ilustración 31: El constructor static sólo se ejecuta una vez, en cambio, constructor normal se ejecuta cada vez que se instancia.

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

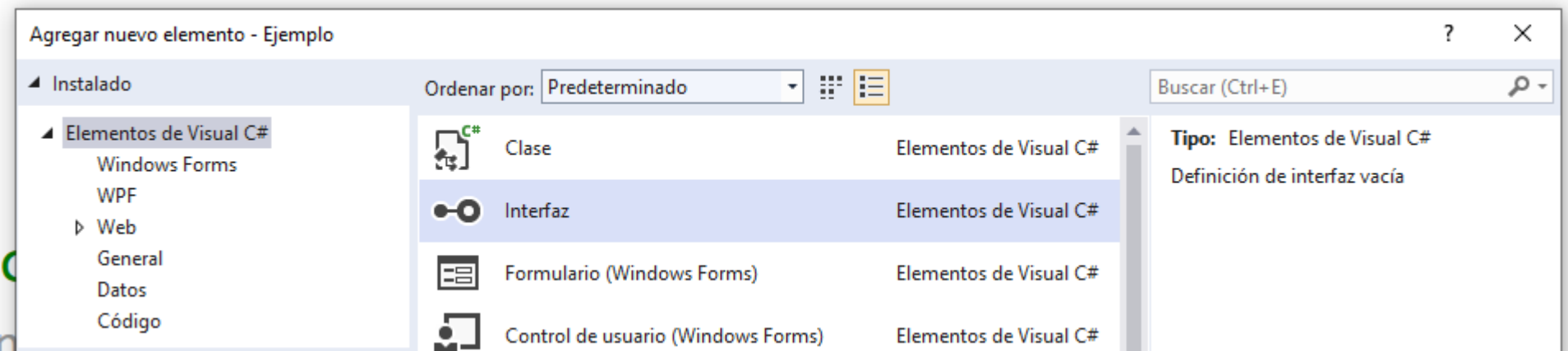


Ilustración 32: En el IDE, se agrega con "Interfaz"

Carpeta 028. IMetodosRequeridos.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 AreaFigura();
        double PerimetroFigura();
    }
}
```

Carpeta 028. Circulo.cs

```
using System;

namespace Ejemplo {
    //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 AreaFigura() {
            return Math.PI * Radio * Radio;
        }

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

```

namespace Ejemplo {
    //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 AreaFigura() {
            return Lado * Lado;
        }

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

```

```

using System;

namespace Ejemplo {

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

            //Imprime los valores
            Console.WriteLine("Área del círculo: " + objCirculo.AreaFigura().ToString());
            Console.WriteLine("Área del cuadrado: " + objCuadrado.AreaFigura().ToString());
            Console.WriteLine("Perímetro del círculo: " + objCirculo.PerimetroFigura().ToString());
            Console.WriteLine("Perímetro del cuadrado: " + objCuadrado.PerimetroFigura().ToString());

            Console.ReadKey();
        }
    }
}

```

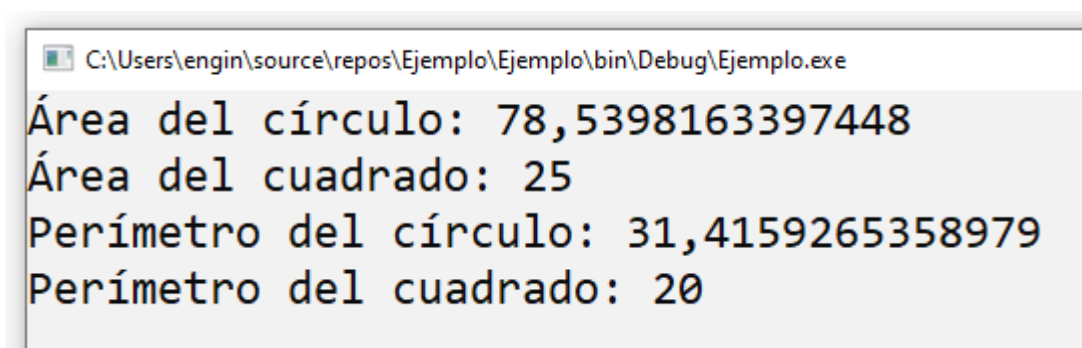


Ilustración 33: Interface

Interface múltiple

Si se puede “implementar” de múltiples “interface”. Ejemplo:

Carpeta 029. ICalculos.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 AreaFigura();
        double PerimetroFigura();
    }
}
```

Carpeta 029. IMuestra.cs

```
namespace Ejemplo {
    //Otra interface para obligar a mostrar los resultados
    interface IMuestra {
        void ImprimeArea();
        void ImprimePerimetro();
    }
}
```

Carpeta 029. Circulo.cs

```
using System;

namespace Ejemplo {
    //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 AreaFigura() {
            return Math.PI * Radio * Radio;
        }

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

        public void ImprimeArea() {
            Console.WriteLine("Área del círculo: " + AreaFigura().ToString());
        }

        public void ImprimePerimetro() {
            Console.WriteLine("Perímetro del círculo: " + PerimetroFigura().ToString());
        }
    }
}
```

```
using System;

namespace Ejemplo {
    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 AreaFigura() {
            return Lado * Lado;
        }

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

        public void ImprimeArea() {
            Console.WriteLine("Área del cuadrado: " + AreaFigura().ToString());
        }

        public void ImprimePerimetro() {
            Console.WriteLine("Perímetro del cuadrado: " + PerimetroFigura().ToString());
        }
    }
}
```

```
using System;

namespace Ejemplo {

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

            //Imprime los valores
            objCuadrado.ImprimeArea();
            objCuadrado.ImprimePerimetro();

            objCirculo.ImprimeArea();
            objCirculo.ImprimePerimetro();

            Console.ReadKey();
        }
    }
}
```

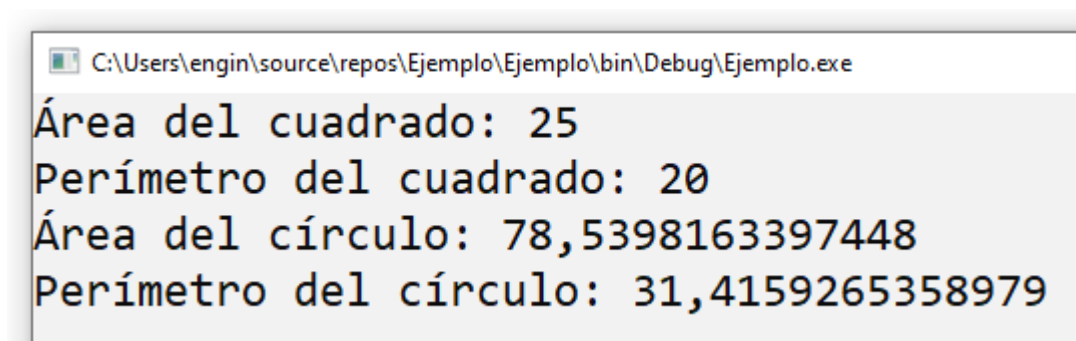


Ilustración 34: Interface múltiple

Herencia e Interface

Se puede heredar de una clase e implementar de varios Interface

Carpeta 030. Madre.cs

```
using System;

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

Carpeta 030. IMetodos.cs

```
namespace Ejemplo {
    interface IMetodos {
        void MetodoA();
        void MetodoB();
    }
}
```

Carpeta 030. IProcedimientos.cs

```
namespace Ejemplo {
    interface IProcedimientos {
        void ProcedimientoA();
        void ProcedimientoB();
    }
}
```

Carpeta 030. Hija.cs

```
using System;

namespace Ejemplo {
    //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");
        }
    }
}
```

```
using System;

namespace Ejemplo {

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

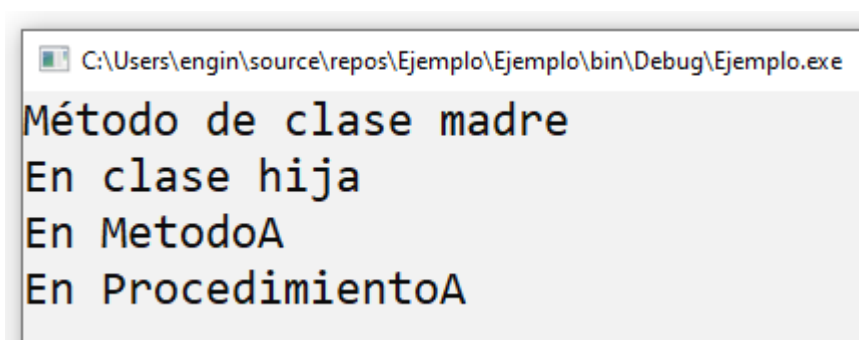


Ilustración 35: Hereda de madre, implementa las "interface"

Enums

Es una “clase especial” para guardar constantes

Carpeta 031. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    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);
            Console.ReadKey();
        }
    }
}
```

Al ejecutar se obtiene esto:

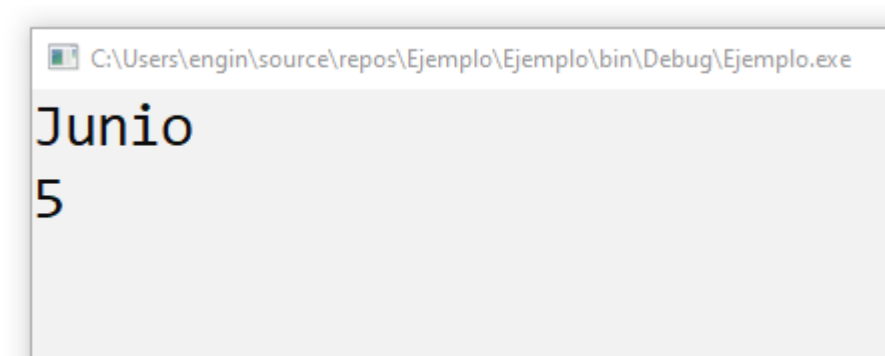


Ilustración 36: Las constantes de un enum comienzan a enumerarse desde cero

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

Cambiando los valores de las constantes en enums

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

Carpeta 032. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    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);
            Console.ReadKey();
        }
    }
}
```

Al ejecutarlo se obtiene

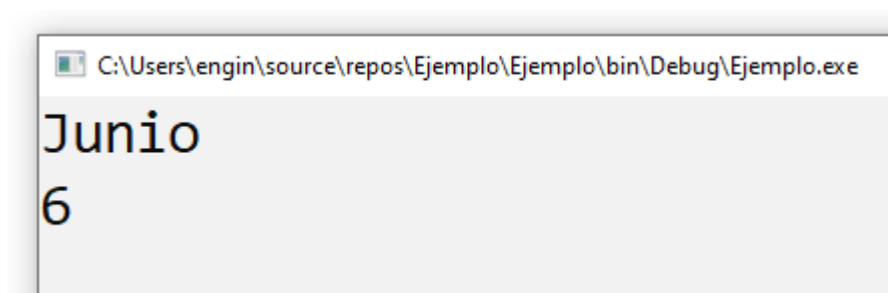


Ilustración 37: Las constantes tienen ahora unos determinados valores

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    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);
            Console.ReadKey();
        }
    }
}
```

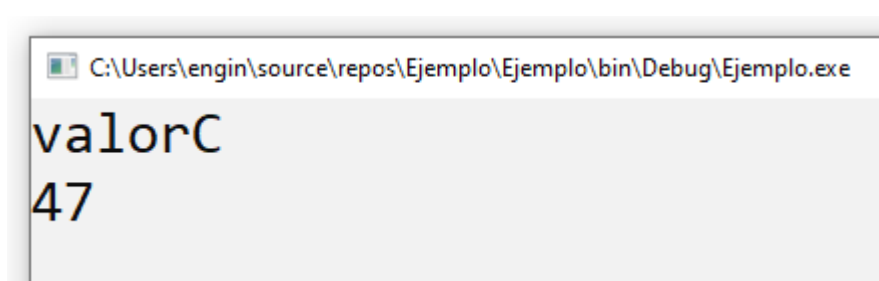


Ilustración 38: Una constante con su valor

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

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 tan 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.

Un ejemplo de su uso:

Carpeta 034. Program.cs

```
using System;

namespace Ejemplo {

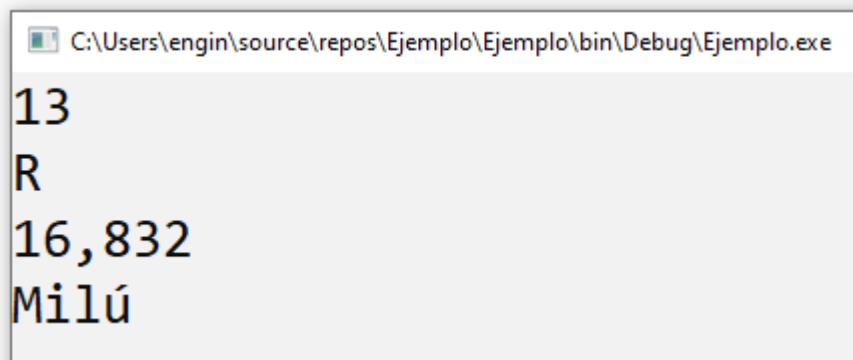
    //Inicia la aplicación aquí
    class Program {

        //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);

            Console.ReadKey();
        }
    }
}
```



C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Debug\Ejemplo.exe

13
R
16,832
Milú

Ilustración 39: Uso de 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.

Carpeta 035. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    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("\nValores después de modificar el original");
            Console.WriteLine(otro.valorA);

            Console.ReadKey();
        }
    }
}
```

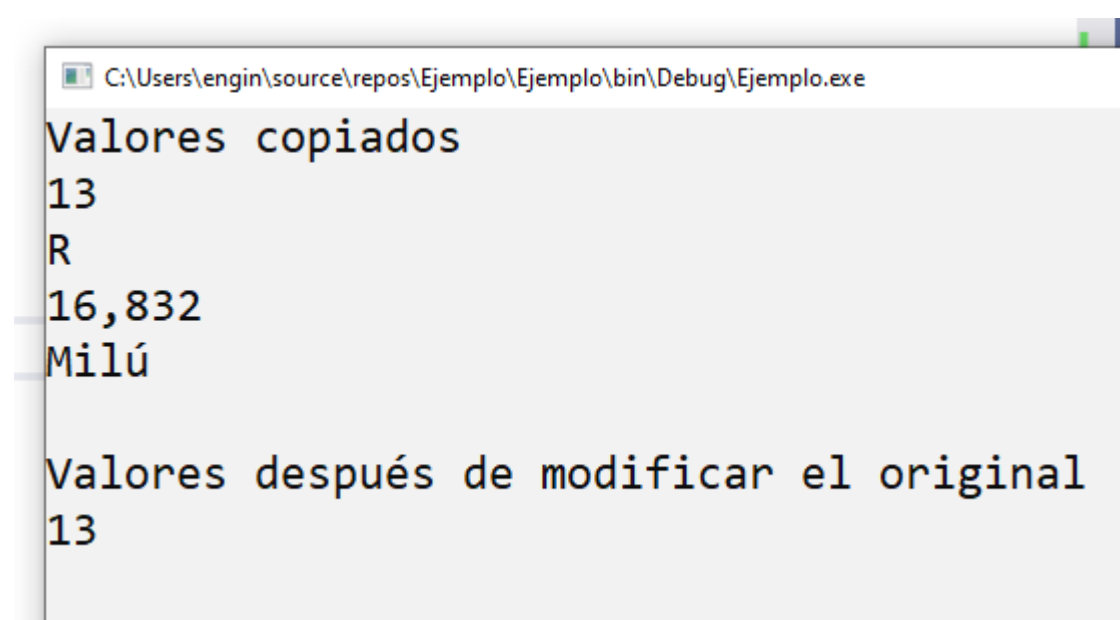


Ilustración 40: Con struct se copian los valores

Métodos en un struct

Un struct puede tener métodos.

Carpeta 036. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    class Program {

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

            public void DaValores(int valorA, char valorB, double valorC, string valorD) {
                this.valorA = valorA;
                this.valorB = valorB;
                this.valorC = valorC;
                this.valorD = valorD;
            }

            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.DaValores(13, 'R', 16.832, "Milú");
            unosValores.ImprimeValores();

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

            Console.ReadKey();
        }
    }
}
```

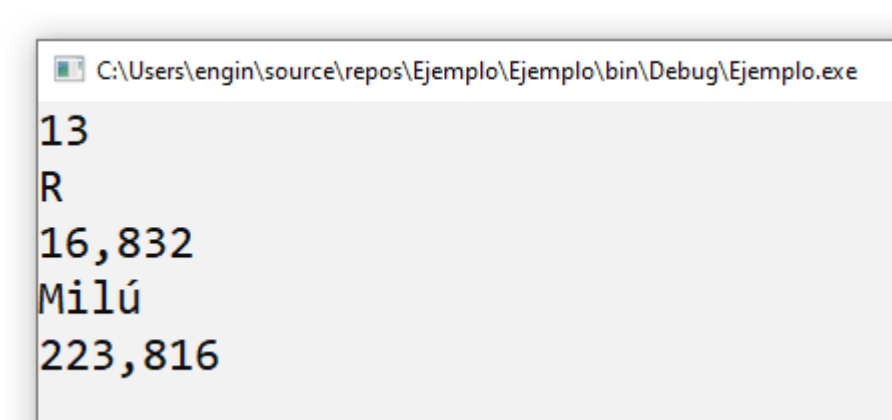


Ilustración 41: Un struct con métodos

Structs y constructores

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

Carpeta 037. Program.cs

```
using System;

namespace Ejemplo {

    //Inicia la aplicación aquí
    class Program {

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

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

            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();

            Console.ReadKey();
        }
    }
}
```

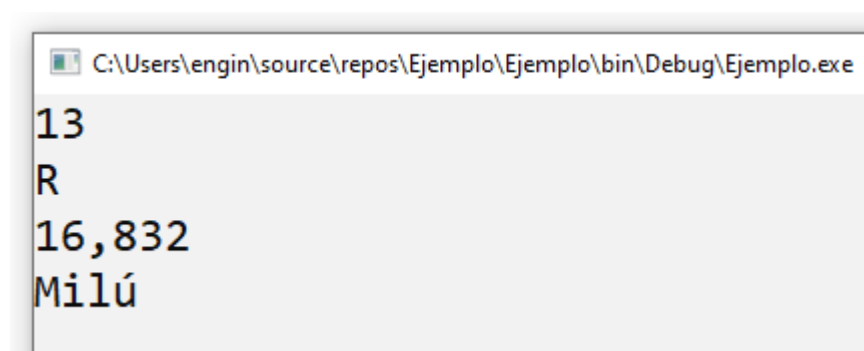


Ilustración 42: structs y un constructor

El constructor se ejecuta cuando se declara la variable con la palabra reservada “new”.

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 uso intensivo cuando por un lado uno como desarrollador de software trabaja en los eventos de un botón o control gráfico y el diseñador del Visual Studio trabaja con la misma clase en otro archivo en las características visuales de ese botón como tamaño, posición, etiqueta, etc.

Carpeta 038. MiClase.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; }
    }
}
```

Carpeta 038. Program.cs

```
using System;

namespace Ejemplo {
    partial class MiClase {
        public MiClase(int valorA, double valorB, char valorC, string valorD) {
            ValorA = valorA;
            ValorB = valorB;
            ValorC = valorC;
            ValorD = valorD;
        }

        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();
            Console.ReadKey();
        }
    }
}
```

En cada archivo hay un pedazo de la clase MiClase (se requiere usar la palabra reservada “partial”).

En una aplicación gráfica de escritorio, el desarrollador trabaja en una clase parcial poniendo su propio código:

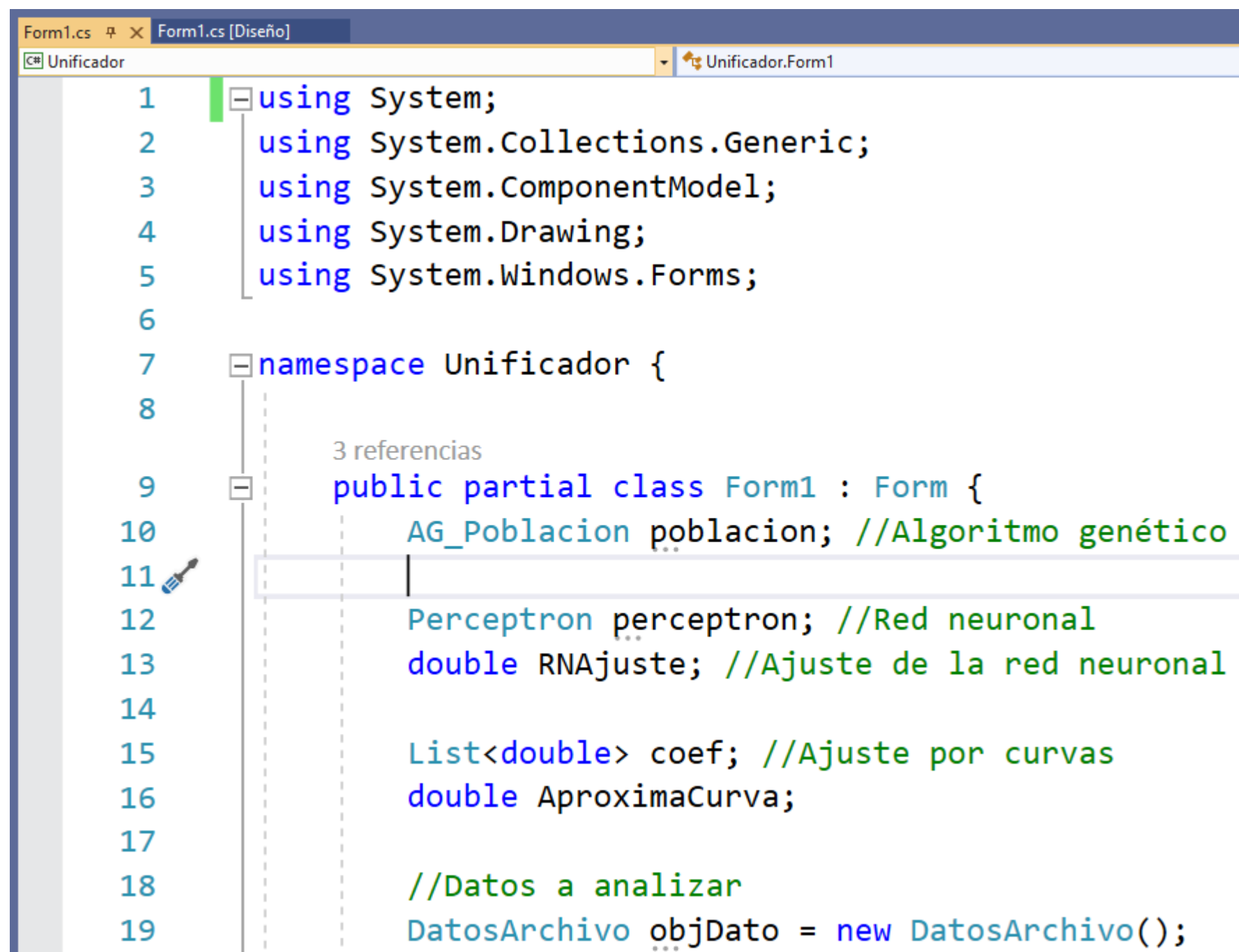


Ilustración 43: Clase parcial, código propio del desarrollador

Y el diseño gráfico se encarga el entorno de desarrollo (Visual Studio):

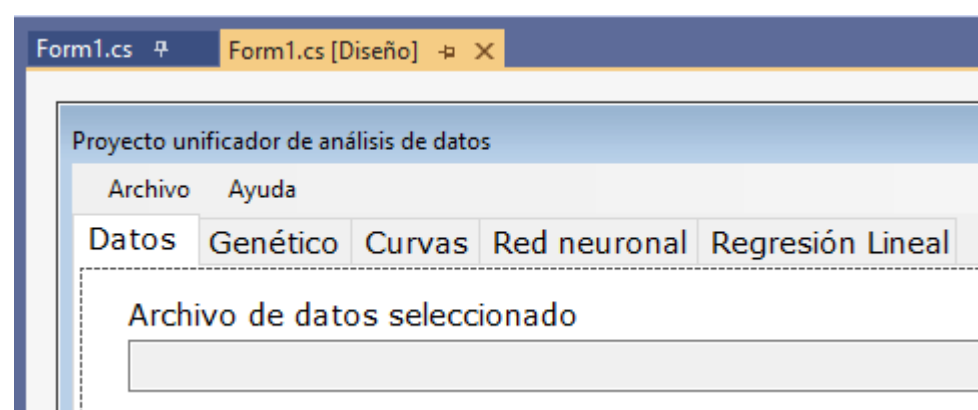


Ilustración 44: Diseño visual de la pantalla con Visual Studio

Son dos archivos generados:

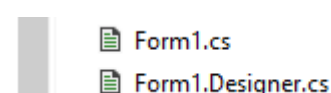
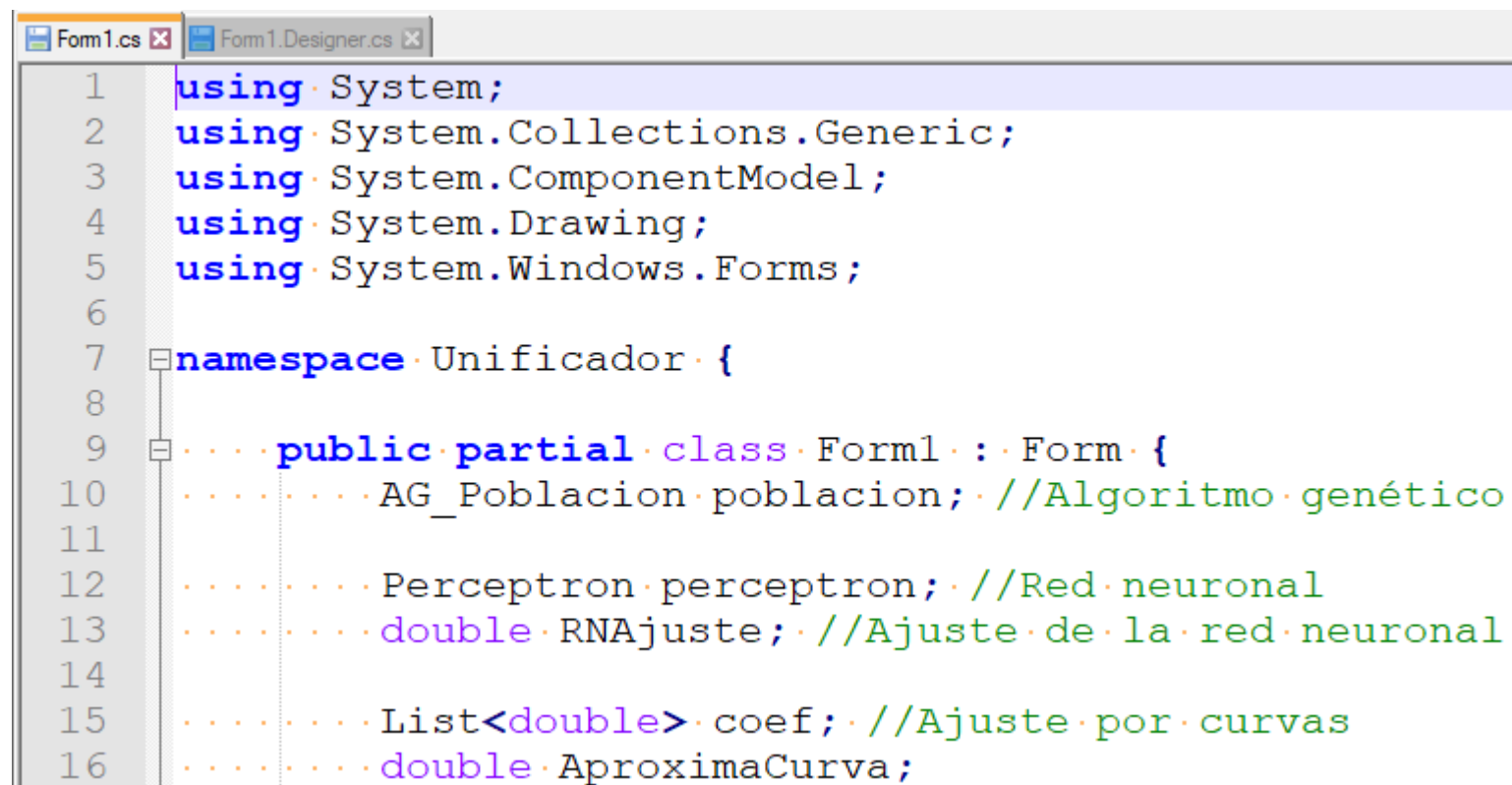


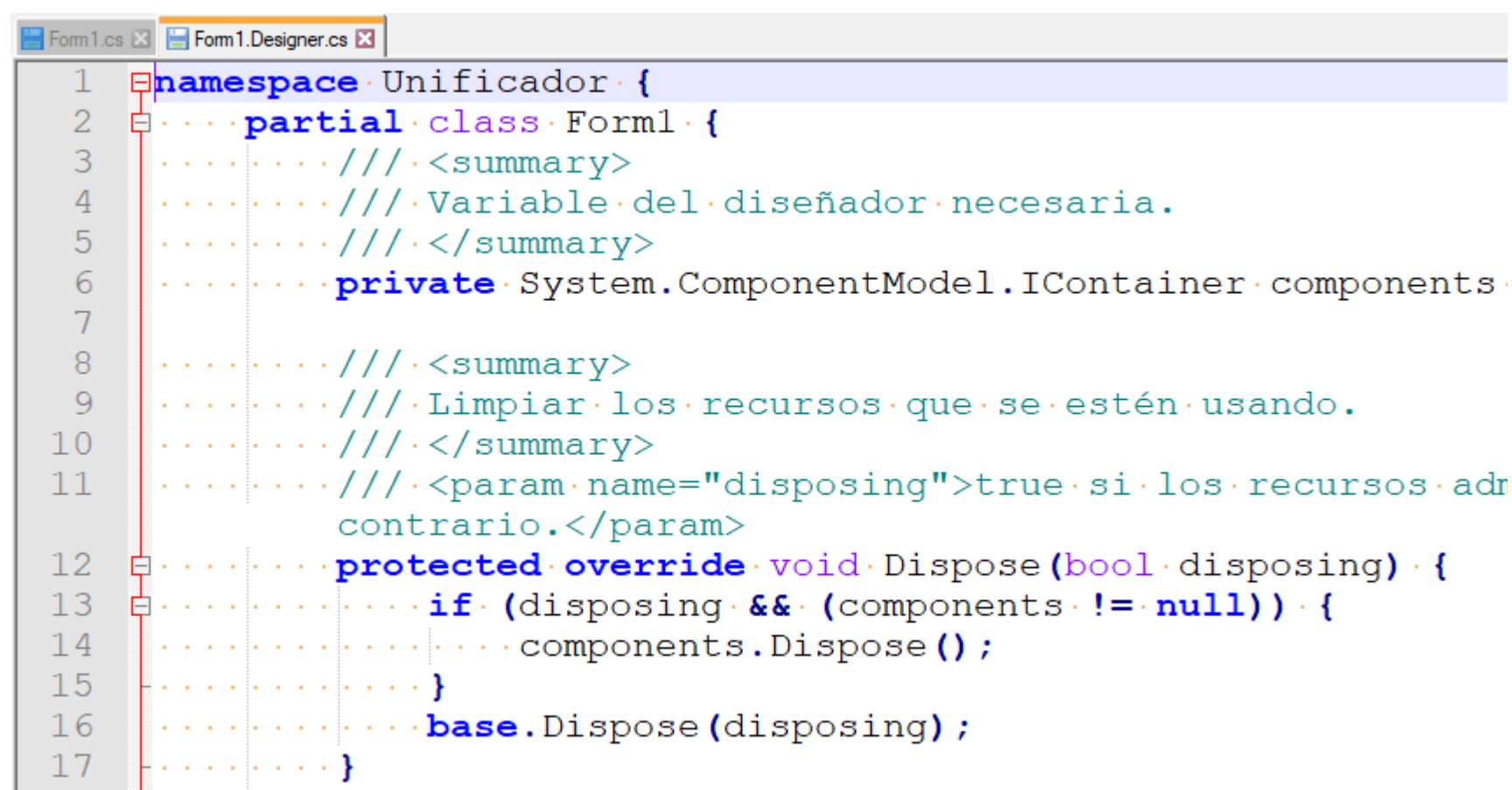
Ilustración 45: Se generan dos archivos de C# para ese formulario

El contenido de los archivos muestra que uno es del programador y el otro del entorno de desarrollo



```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 namespace Unificador {
8
9     public partial class Form1 : Form {
10         AG_Poblacion poblacion; //Algoritmo genético
11
12         Perceptron perceptron; //Red neuronal
13         double RNAjuste; //Ajuste de la red neuronal
14
15         List<double> coef; //Ajuste por curvas
16         double AproximaCurva;
```

Ilustración 46: Archivo con el código del desarrollador



```
1 namespace Unificador {
2     partial class Form1 {
3         /// <summary>
4         /// Variable del diseñador necesaria.
5         /// </summary>
6         private System.ComponentModel.IContainer components
7
8         /// <summary>
9         /// Limpiar los recursos que se estén usando.
10        /// </summary>
11        /// <param name="disposing">true si los recursos administrados
12        ///     contrario.</param>
13        protected override void Dispose (bool disposing) {
14            if (disposing && (components != null)) {
15                components.Dispose ();
16            }
17            base.Dispose (disposing);
18        }
19    }
20 }
```

Ilustración 47: Archivo con el código generado por Visual Studio

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.

Carpeta 039. MiClase.cs

```
using System;

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 valorA, double valorB, char valorC, string valorD) {
            ValorA = valorA;
            ValorB = valorB;
            ValorC = valorC;
            ValorD = valorD;
        }

        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");
        }
    }
}
```

Y la clase que llama

Carpeta 039. Program.cs

```
using System;

namespace Ejemplo {

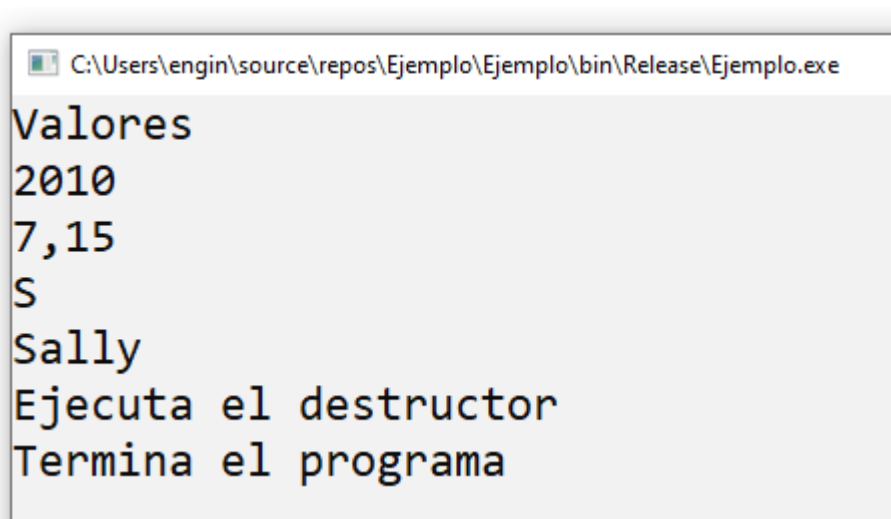
    //Inicia la aplicación aquí
    class Program {
        public static void Main() {
            Procedimiento();

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

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

        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
        }
    }
}
```



A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\engin\source\repos\Ejemplo\Ejemplo\bin\Release\Ejemplo.exe. The output of the program is displayed as follows:

```
Valores
2010
7,15
S
Sally
Ejecuta el destructor
Termina el programa
```

Ilustración 48: Se ejecuta explícitamente el Garbage Collector para obligar a ejecutar el destructor de la clase cuando queramos

Nota: Por lo leído 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.

Patrones de diseño

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

Factory Method

Se crea una interface de figuras

Carpeta 040. IFigura.cs

```
//Patrón: Factory Method
namespace PatronDiseno {

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

Luego las clases que implementan esa interface

Carpeta 040. Circulo.cs

```
//Patrón: Factory Method
using System;

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

Carpeta 040. Rectangulo.cs

```
//Patrón: Factory Method
using System;

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

Carpeta 040. Triangulo.cs

```
//Patrón: Factory Method
using System;

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

Luego se hace la fábrica de figuras

Carpeta 040. FabricaFiguras.cs

```
//Patrón: Factory Method

namespace PatronDiseno {
    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;
        }
    }
}
```

Y ahora se prueba

Carpeta 040. Program.cs

```
//Patrón: Factory Method
using System;

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

            //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();

            Console.ReadKey();
        }
    }
}
```

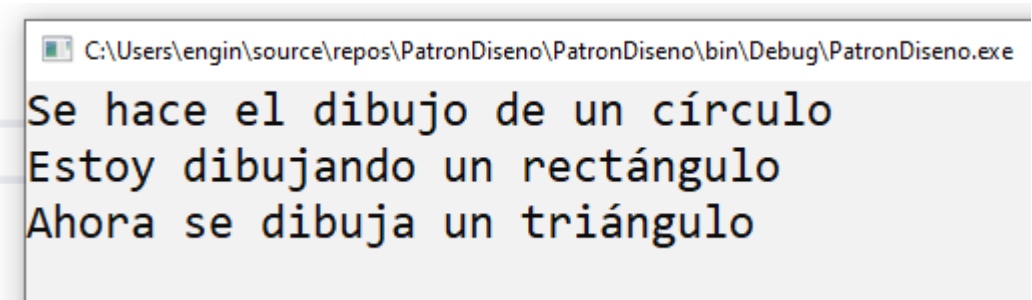


Ilustración 49: Patrón de diseño: Factory Method

Abstract Factory

Se crea una interface de figuras

Carpeta 041. IFigura.cs

```
//Patrón: Abstract Factory
namespace PatronDiseno {
    //Interface que obliga a definir el método dibujar
    public interface IFigura {
        void Dibujar();
    }
}
```

Luego las clases que implementan esa interface

Carpeta 041. Rectangulo.cs

```
//Patrón: Abstract Factory
using System;

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

Carpeta 041. Triangulo.cs

```
//Patrón: Abstract Factory
using System;

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

Carpeta 041. Circulo.cs

```
//Patrón: Abstract Factory
using System;

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

```
//Patrón: Abstract Factory
namespace PatronDiseno {
    public interface IColor {
        void Rellenar();
    }
}
```

```
//Patrón: Abstract Factory
using System;

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

```
//Patrón: Abstract Factory
using System;

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

```
//Patrón: Abstract Factory
using System;

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

Se crea una fábrica abstracta para poder generar las dos fábricas de figuras y colores

Carpeta 041. FabricaAbstracta.cs

```
//Patrón: Abstract Factory
namespace PatronDiseno {
    public abstract class FabricaAbstracta {
        public abstract IFigura GetFigura(string TipoFigura);
        public abstract IColor GetColor(string color);
    }
}
```

Ahora se crea la fábrica de figuras

Carpeta 041. FabricaFiguras.cs

```
//Patrón: Abstract Factory
namespace PatronDiseno {
    public class FabricaFiguras : FabricaAbstracta {
        //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;
        }
    }
}
```

Y luego se crea la fábrica de colores

Carpeta 041. FabricaColores.cs

```
//Patrón: Abstract Factory
namespace PatronDiseno {
    class FabricaColores : FabricaAbstracta {
        //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;
        }
    }
}
```



```
//Patrón: Abstract Factory
namespace PatronDiseno {
    class ProductorDeFabricas {
        public static FabricaAbstracta GetFabrica(string seleccion) {
            if (seleccion.Equals("FIGURA")) return new FabricaFiguras();
            if (seleccion.Equals("COLOR")) return new FabricaColores();
            return null;
        }
    }
}
```

```
//Patrón: Abstract Factory
using System;

namespace PatronDiseno {
    class Program {
        static void Main() {
            //Trae una determinada fábrica en este caso de FIGURA
            FabricaAbstracta fabricaFiguras = ProductorDeFabricas.GetFabrica("FIGURA");

            //Obtenida la fábrica, se solicita un tipo de objeto de esa fábrica
            IFigura figura1 = fabricaFiguras.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 = fabricaFiguras.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 = fabricaFiguras.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
            FabricaAbstracta FabricaColores = ProductorDeFabricas.GetFabrica("COLOR");

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

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

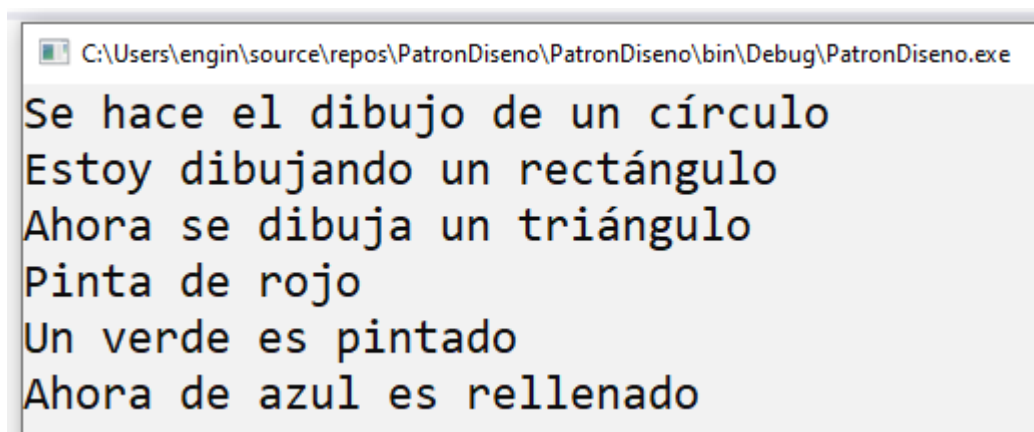
            //Obtenida la fábrica, se solicita un tipo de objeto de esa fábrica
            IColor color2 = FabricaColores.GetColor("VERDE");

            //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 = FabricaColores.GetColor("AZUL");

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

            Console.ReadKey();
        }
    }
}
```



```
C:\Users\engin\source\repos\PatronDiseno\PatronDiseno\bin\Debug\PatronDiseno.exe
Se hace el dibujo de un círculo
Estoy dibujando un rectángulo
Ahora se dibuja un triángulo
Pinta de rojo
Un verde es pintado
Ahora de azul es relleñado
```

Ilustración 50: Abstract Factory

```
//Patrón: Singleton
using System;

namespace PatronDiseno {
    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");
        }
    }
}
```

```
//Patrón: Singleton
using System;

namespace PatronDiseno {
    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();

            Console.ReadKey();
        }
    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    public interface IEmpacado {
        string Empaque();
    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    //Todo producto en la comida tendrá estos ítems: Nombre, como se empaca, precio
    public interface Item {
        string Nombre();

        IEmpacado EmpacandoProducto();

        float Precio();
    }
}
```

Clases concretas que hacen uso de la “interface” de empackado

```
//Patrón: Builder
namespace PatronDiseno {
    class Envoltura : IEmpacado {
        public string Empaque() {
            return "Empaque Ecológico";
        }
    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    public class Botella : IEmpacado {
        public string Empaque() {
            return "Botella biodegradable";
        }
    }
}
```

Clases concretas que hacen uso de la “interface” de Ítem

```
//Patrón: Builder
namespace PatronDiseno {
    public abstract class BebidaFria : Item {
        public IEmpacado EmpacandoProducto() {
            return new Botella();
        }

        public abstract float Precio();

        public abstract string Nombre();
    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    public abstract class Hamburguesa : Item {

        public IEmpacado EmpacandoProducto() {
            return new Envoltura();
        }

        public abstract float Precio();

        public abstract string Nombre();

    }
}
```

Ahora los tipos de hamburguesas en particular

```
//Patrón: Builder
namespace PatronDiseno {
    public class HamburguesaPollo : Hamburguesa {
        public override float Precio() {
            return 7000;
        }

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

    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    class HamburguesaVegetariana : Hamburguesa {
        public override float Precio() {
            return 5000;
        }

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

    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    class Malteada : BebidaFria {
        public override float Precio() {
            return 4700;
        }

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

```
//Patrón: Builder
namespace PatronDiseno {
    class CaFeFrio : BebidaFria {
        public override float Precio() {
            return 4000;
        }

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

```
//Patrón: Builder
using System;
using System.Collections.Generic;

namespace PatronDiseno {
    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.WriteLine("Item: " + item.Nombre());
                Console.WriteLine(", Empaque: " + item.EmpacandoProducto().Empaque());
                Console.WriteLine(", Precio: " + item.Precio());
            }
        }
    }
}
```

```
//Patrón: Builder
namespace PatronDiseno {
    //Prepara la comida dependiendo si es vegetariana o no
    class FabricaComida {
        public Comida PrepararComidaVegetariana() {
            Comida miComida = new Comida();
            miComida.AddItem(new HamburguesaVegetariana());
            miComida.AddItem(new CaFeFrio());
            return miComida;
        }

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

```
//Patrón: Builder
using System;

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

            Comida vegetariano = miComida.PrepararComidaVegetariana();
            Console.WriteLine("Comida vegetariana");
            vegetariano.MostrarItems();
            Console.WriteLine("Costo total: " + vegetariano.GetCosto());

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

            Console.ReadKey();
        }
    }
}
```

```
C:\Users\engin\source\repos\PatronDiseno\PatronDiseno\bin\Debug\PatronDiseno.exe

Comida vegetariana
Item: Hamburguesa vegetariana
, Empaque: Empaque Ecológico
, Precio: 5000
Item: Café frío
, Empaque: Botella biodegradable
, Precio: 4000
Costo total: 9000

Comida No vegetariana
Item: Hamburguesa de pollo
, Empaque: Empaque Ecológico
, Precio: 7000
Item: Malteada
, Empaque: Botella biodegradable
, Precio: 4700
Costo total: 11700
```

Ilustración 51: Patrón de diseño: Builder

Adapter

Las “interfaces” no compatibles

Carpeta 044. IEjecutorMultimedia.cs

```
//Patrón de diseño: Adapter
namespace PatronDiseno {
    public interface IEjecutorMultimedia {
        void Ejecutar(string TipoAudio, string NombreArchivo);
    }
}
```

Carpeta 044. IEjecutorAvanzadoArchivosMultimedia.cs

```
//Patrón de diseño: Adapter
namespace PatronDiseno {
    public interface IEjecutorAvanzadoArchivosMultimedia {
        void EjecutaVLC(string NombreArchivo);

        void EjecutaMP4(string NombreArchivo);
    }
}
```

Clases que usan la segunda “interface”

Carpeta 044. EjecutorMP4.cs

```
//Patrón de diseño: Adapter
using System;

namespace PatronDiseno {
    class EjecutorMP4 : IEjecutorAvanzadoArchivosMultimedia {

        public void EjecutaVLC(string NombreArchivo) {
        }

        public void EjecutaMP4(string NombreArchivo) {
            Console.WriteLine("Ejecutando un archivo MP4. Nombre: " + NombreArchivo);
        }
    }
}
```

Carpeta 044. EjecutorVLC.cs

```
//Patrón de diseño: Adapter
using System;

namespace PatronDiseno {
    class EjecutorVLC : IEjecutorAvanzadoArchivosMultimedia {

        public void EjecutaVLC(string NombreArchivo) {
            Console.WriteLine("Ejecutando un archivo VLC. Nombre: " + NombreArchivo);
        }

        public void EjecutaMP4(string NombreArchivo) {
        }
    }
}
```

```
//Patrón de diseño: Adapter
namespace PatronDiseno {
    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);
            }
        }
    }
}
```

La clase que une la funcionalidad de ambas interfaces ya implementadas

```
//Patrón de diseño: Adapter
using System;

namespace PatronDiseno {
    class EjecutorAudio : IEjecutorMultimedia {

        AdaptadorMultimedia adaptadorMultimedia;

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

Y se prueba:

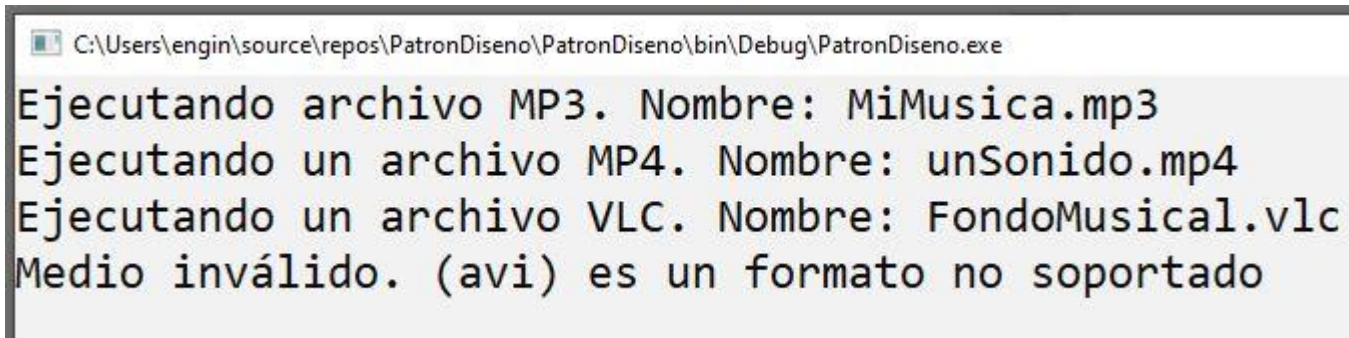
Carpeta 044. Program.cs

```
//Patrón de diseño: Adapter
using System;

namespace PatronDiseno {
    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");

            Console.ReadKey();
        }
    }
}
```



```
C:\Users\engin\source\repos\PatronDiseno\PatronDiseno\bin\Debug\PatronDiseno.exe
Ejecutando archivo MP3. Nombre: MiMusica.mp3
Ejecutando un archivo MP4. Nombre: unSonido.mp4
Ejecutando un archivo VLC. Nombre: FondoMusical.vlc
Medio inválido. (avi) es un formato no soportado
```

Ilustración 52: Patrón de diseño Adapter

```
//Patrón de diseño: Composite
using System.Collections.Generic;

namespace PatronDiseno {
    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() {
            return "Empleado => Nombre: " + nombre + ", departamento: " + departamento + ", salario: " +
            salario.ToString();
        }
    }
}
```

```
//Patrón de diseño: Composite
using System;

namespace PatronDiseno {
    class Program {
        static void Main() {
            Empleado Gerente = new Empleado("Laura", "Gerente", 5000000);
            Empleado jefeVentas = new Empleado("Patricia", "Jefa de Ventas", 3000000);
            Empleado jefeMercadeo = new Empleado("Adriana", "Jefa de Mercadeo", 3000000);
            Empleado disenador1 = new Empleado("Sandra", "Marketing", 2000000);
            Empleado disenador2 = new Empleado("Alejandra", "Marketing", 2000000);
            Empleado vendedor1 = new Empleado("Francisca", "Ventas", 200000);
            Empleado vendedor2 = new Empleado("Flor", "Ventas", 200000);

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

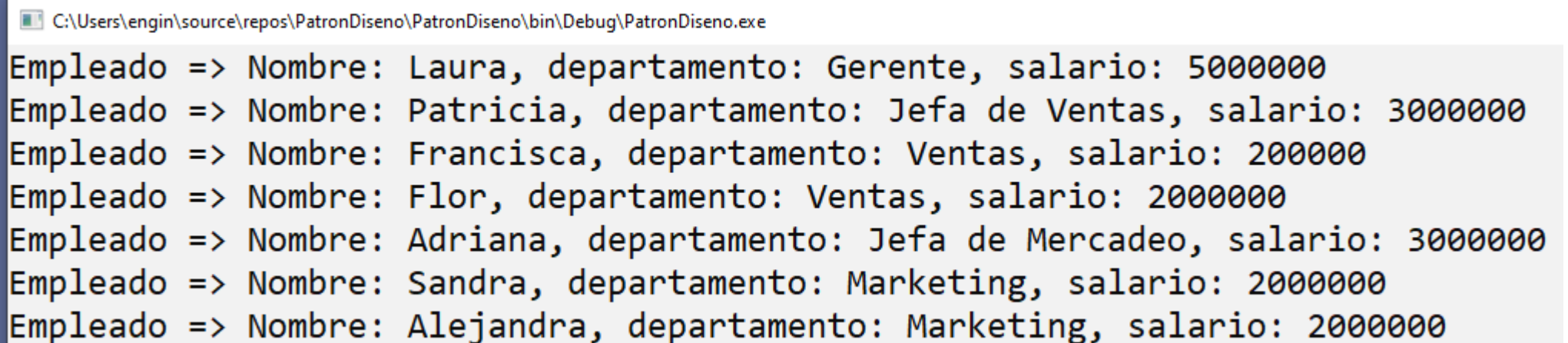
            jefeVentas.Adicionar(vendedor1);
            jefeVentas.Adicionar(vendedor2);

            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());
                }
            }
            Console.ReadKey();
        }
    }
}
```



```
C:\Users\engin\source\repos\PatronDiseno\PatronDiseno\bin\Debug\PatronDiseno.exe

Empleado => Nombre: Laura, departamento: Gerente, salario: 5000000
Empleado => Nombre: Patricia, departamento: Jefa de Ventas, salario: 3000000
Empleado => Nombre: Francisca, departamento: Ventas, salario: 200000
Empleado => Nombre: Flor, departamento: Ventas, salario: 2000000
Empleado => Nombre: Adriana, departamento: Jefa de Mercadeo, salario: 3000000
Empleado => Nombre: Sandra, departamento: Marketing, salario: 2000000
Empleado => Nombre: Alejandra, departamento: Marketing, salario: 2000000
```

Ilustración 53: Patrón de diseño: Composite

```
//Patrón de diseño: Facade
namespace PatronDiseno {
    interface IFigura {
        void Dibujar();
    }
}
```

```
//Patrón de diseño: Facade
using System;

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

```
//Patrón de diseño: Facade
using System;

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

```
//Patrón de diseño: Facade
using System;

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

```
//Patrón de diseño: Facade
namespace PatronDiseno {
    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();
        }
    }
}
```

```
//Patrón de diseño: Facade
using System;

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

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

            Console.ReadKey();
        }
    }
}
```

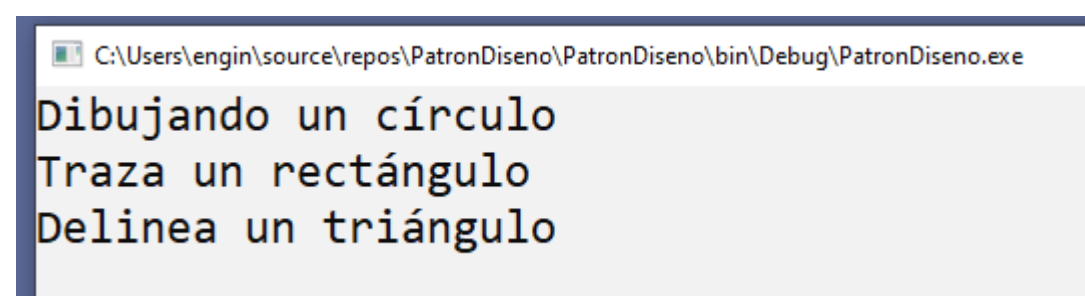


Ilustración 54: Patrón de diseño: Facade

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

```
//Patrón de diseño: Modelo Vista Controlador
using System;

namespace PatronDiseno {
    class VisorEstudiante {
        public void ImprimeDetallesEstudiante(string NombreEstudiante, string CodigoEstudiante) {
            Console.WriteLine("Estudiante: ");
            Console.WriteLine("Nombre: " + NombreEstudiante);
            Console.WriteLine("Código: " + CodigoEstudiante);
        }
    }
}
```

```
//Patrón de diseño: Modelo Vista Controlador
namespace PatronDiseno {
    class ControladorEstudiante {
        private Estudiante modelo;
        private VisorEstudiante vista;

        public ControladorEstudiante(Estudiante modelo, VisorEstudiante vista) {
            this.modelo = modelo;
            this.vista = vista;
        }

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

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

        public void setCodigoEstudiante(string codigo) {
            modelo.Codigo = codigo;
        }

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

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



```
//Patrón de diseño: Modelo Vista Controlador
using System;

namespace PatronDiseno {
    class Program {
        static void Main() {
            Estudiante modelo = TraeEstudianteBaseDatos();
            VisorEstudiante vista = new VisorEstudiante();
            ControladorEstudiante controlador = new ControladorEstudiante(modelo, vista);

            controlador.ActualizarVista();
            controlador.setNombreEstudiante("Laura");
            controlador.ActualizarVista();

            Console.ReadKey();
        }

        private static Estudiante TraeEstudianteBaseDatos() {
            Estudiante estudiante = new Estudiante();
            estudiante.Nombre = "Johanna";
            estudiante.Codigo = "17123456";
            return estudiante;
        }
    }
}
```

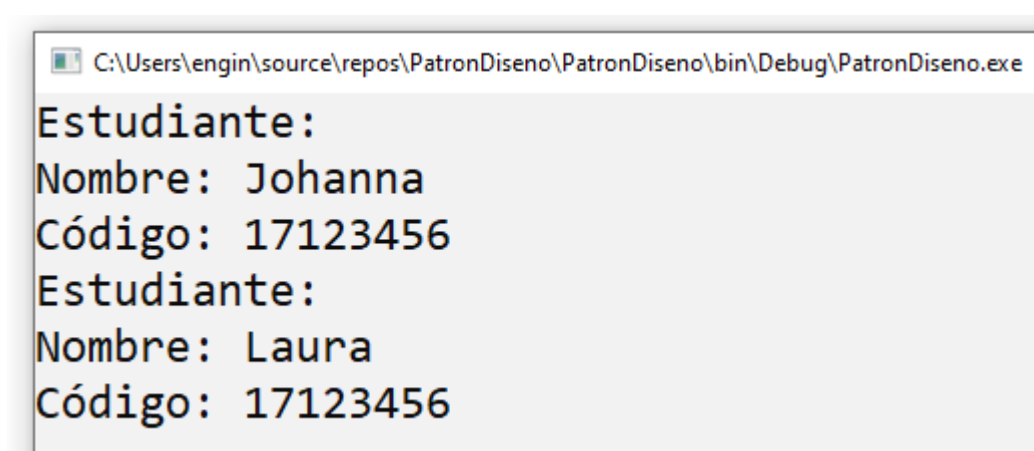


Ilustración 55: Patrón de diseño: Modelo Vista Controlador

Enlaces de interés sobre Programación Orientada a Objetos y C#

https://www.tutorialspoint.com/design_pattern/index.htm

<https://reactiveprogramming.io/blog/es/patrones-de-diseno/factory-method>

<https://refactoring.guru/es/design-patterns>

<https://www.c-sharpcorner.com/>

<https://docs.microsoft.com/en-us/dotnet/csharp/>

<https://docs.microsoft.com/es-es/visualstudio/get-started/csharp/?view=vs-2019>

<https://visualstudio.microsoft.com/es/vs/>

<https://sourceforge.net/projects/sharpdevelop/>

<https://openlibra.com/es/book/c-notes-for-professionals>