

# REPASO DE SOBRECARGA Y SOBREESCRITURA

## Sobrecarga - OverLoad

La sobrecarga de métodos permite definir dos o más métodos con el mismo nombre, pero que difieren en cantidad o tipo de parámetros.

Esta característica del lenguaje nos facilita la implementación de algoritmos que cumplen la misma función pero que difieren en los parámetros.

## Sobreescritura - Override

El modificador override es necesario para ampliar o modificar la implementación abstracta o virtual de un método, propiedad, indexador o evento heredado.

- **Abstract** indica que puede ser sobreescrito o no.
- **Virtual** indica que “**debe**” ser sobreescrito de manera obligatoria.

## Ejemplo

En este ejemplo, la clase **Cuadrado**, debe proporcionar una implementación de invalidación (Override) de **Área** porque ésta se hereda de la clase abstracta **Figura**:

```
abstract class Figura
{
    abstract public int Area();
}
class Cuadrado : Figura
{
    int lado = 0;

    public Cuadrado(int n)
    {
        lado = n;
    }
    // El método Área es requerido para evitar un error en tiempo de compilación
```

```


public override int Area()
{
    return lado * lado;
}

static void Main()
{
    Cuadrado cd = new Cuadrado(12);
    Console.WriteLine("Area del cuadrado = {0}", cd.Area());
}

interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
}

```



A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe. The command prompt displays the text "Area del cuadrado = 144".

Un método con **override** proporciona una nueva implementación de un miembro que se hereda de una clase base. El método invalidado por una declaración **override** se conoce como método base invalidado. El método base reemplazado debe tener la misma firma que el método **override**.

No se puede reemplazar un método estático o no virtual. El método base reemplazado debe ser **virtual**, **abstract** u **override**.

Una declaración **override** no puede cambiar la accesibilidad del método virtual. El método **override** y el método **virtual** deben tener el mismo modificador de nivel de acceso. No se pueden usar los modificadores **new**, **static** o **virtual** para modificar un método **override**. Una declaración de propiedad de invalidación debe especificar exactamente el mismo modificador de acceso, tipo y nombre que la propiedad heredada, y la propiedad invalidada debe ser **virtual**, **abstract** u **override**.

## Ejemplo

Este ejemplo define una clase base denominada **Empleado** y una clase derivada denominada **EmpleadoDeVentas** . La clase **EmpleadoDeVentas** incluye una propiedad adicional, **bonoDeVentas**, e invalida el método **CalcularPago** para tenerlo en cuenta.

```
class TestOverride
{
    public class Empleado
    {
        public string Nombre;

        // pagoBase está definido como Protegido, por lo tanto solo
        // puede accederse por su clase y clases derivadas.
        protected decimal pagoBase;

        // Constructor para setear los valores de Nombre y pagoBase.
        public Empleado(string paramNombre, decimal paramPagoBase)
        {
            this.Nombre = paramNombre;
            this.pagoBase = paramPagoBase;
        }
        // Declarado Virtual para que pueda ser sobre escrito.
        public virtual decimal CalcularPago()
        {
            return pagoBase;
        }
    }

    // Derivar una nueva clase de empleado.
    public class EmpleadoDeVentas: Empleado
    {
        // nuevo campo que afectará el pagoBase.
        private decimal bonoDeVentas;

        // El constructor llama a la versión de la clase base
        // e inicializa luego el campo bonoDeVentas.
        public EmpleadoDeVentas(string paramNombre, decimal paramPagoBase,
```

```

        decimal paramBonoDeVentas) : base( paramNombre, paramPagoBase)
    {
        this.bonoDeVentas = paramBonoDeVentas;
    }

    // Sobreescribe el método CalcularPago para tener en cuenta el bono
    public override decimal CalcularPago()
    {
        return pagoBase + bonoDeVentas;
    }
}

static void Main()
{
    // Creamos algunos empleados
    EmpleadoDeVentas empleado1 = new EmpleadoDeVentas("Alice",
        1000, 500);
    Empleado empleado2 = new Empleado("Bob", 1200);

    Console.WriteLine("Empleado " + empleado1.Nombre +
        " gana: " + empleado1.CalcularPago());
    Console.WriteLine("Empleado " + empleado2.Nombre +
        " gana: " + empleado2.CalcularPago());
}
}

```



```

C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Empleado Alice gana: 1500
Empleado Bob gana: 1200

```

**Nota:** Recuerde que en las buenas prácticas se aconseja colocar una clase por archivo de código fuente (archivo .cs). Aquí se colocan una debajo de la otra para que se pueda comprender mejor el ejemplo.

## INTRODUCCIÓN A RELACIONES ENTRE OBJETOS

Los objetos pueden relacionarse entre sí de varias maneras. Los tipos principales de relación son **jerárquica** y de **contención**.

## Relación jerárquica

Cuando las clases derivan de las clases más fundamentales, se dice que tienen una relación jerárquica. Las jerarquías de clases son útiles cuando se describen elementos que constituyen un subtipo de una clase más general. Por ejemplo, en el espacio de nombres `System.Web.UI`, las clases `Label` y `TextBox` derivan las dos de la clase `WebControl`. Las clases derivadas heredan miembros de la clase en la que se basan, lo que permite agregar complejidad a medida que se progresa en una jerarquía de clases.

## Relación de contención

Otra manera en que se pueden relacionar objetos es una **relación de contención**. Los objetos contenedores encapsulan lógicamente otros objetos. Por ejemplo, el objeto **OperatingSystem** contiene lógicamente un objeto **Version** que vuelve a través de su propiedad **Version**. Observe que el objeto contenedor no contiene físicamente ningún otro objeto. Otro ejemplo puede ser la propiedad **Colors** que dentro almacena una lista de objetos del tipo **Color**, que poseen valor y nombre del color.

## Colecciones

Un tipo de contención de objetos particular lo representan las colecciones. Las colecciones son grupos de objetos similares que se pueden enumerar. La instrucción **Foreach** permite recorrer en iteración los elementos de una colección. Además, las colecciones suelen permitir el uso de **Item** (propiedad del objeto **Collection**) para recuperar elementos mediante su índice o asociándolos con una key que es un nombre único. Las colecciones pueden ser más fáciles de utilizar que los arreglos (`Arrays`) puesto que permiten agregar o quitar elementos sin utilizar índices. Debido a su facilidad de uso, las colecciones se utilizan frecuentemente para almacenar formularios y controles.

# CLASES ESTÁTICAS

Una clase estática es básicamente igual que una clase no estática, pero existe una diferencia: no se pueden crear instancias de una clase estática. En otras palabras, no puede utilizar la palabra clave **new** para crear una variable de instancia de una clase estática. Dado que no hay ninguna variable de instancia, el acceso a los miembros de una clase estática se realiza mediante el propio **nombre de clase**. Por ejemplo, si tiene una clase estática llamada **Utiles** e incluye un método público llamado **Agregar**, la llamada al método se realiza tal como se muestra en el ejemplo siguiente:

```
Utiles.Agregar();
```

Una clase estática se puede utilizar como un contenedor de conjuntos de métodos que solo funcionan con parámetros de entrada y no tienen que obtener ni establecer campos internos de instancia. Por ejemplo, en la biblioteca de clases .NET Framework, la clase estática **Math** contiene varios métodos que realizan operaciones matemáticas, sin ningún requisito para almacenar o recuperar datos únicos de una instancia determinada de la clase **Math**. Es decir, los miembros de la clase se aplican indicando el nombre de la clase y del método, como se muestra en el ejemplo siguiente.

```
double dub = -3.14;  
Console.WriteLine(Math.Abs(dub));  
Console.WriteLine(Math.Floor(dub));  
Console.WriteLine(Math.Round(Math.Abs(dub)));
```



Como sucede con todos los tipos de clase, Common Language Runtime (CLR) de .NET Framework carga la información de tipo de una clase estática cuando se carga el programa que hace referencia a la clase. El programa no puede especificar exactamente cuándo se carga la clase. Sin embargo, se garantiza que se ha cargado, que sus campos se han inicializado y que se ha llamado a su constructor estático antes de que se haga referencia a la clase por primera vez en el programa. Solo se llama una vez a un constructor estático y una clase estática permanece en memoria durante el período de duración de la aplicación que la usa.

En la siguiente lista se indican las características principales de una clase estática:

- Sólo contiene miembros estáticos.
- No se pueden crear instancias de ella (no se puede usar **new**).
- Es de tipo **sealed**.
- No puede contener constructores de instancia.

Resumiendo crear una clase estática es, básicamente igual que crear una clase que sólo tiene **miembros estáticos** y un **constructor privado**. *Un constructor privado evita que se creen instancias de la clase*. La ventaja de utilizar una clase estática es que el compilador puede comprobar que no se agregue accidentalmente ningún miembro de instancia. El compilador garantizará que no se puedan crear instancias de esta clase.

Las clases estáticas al ser de tipo **sealed** (clases selladas) no pueden heredarse. No pueden heredar de ninguna clase, excepto de la clase **Object**. Las clases estáticas no pueden tener un constructor de instancia; sin embargo, pueden tener **un constructor estático**. Las clases no estáticas también deben definir un constructor estático si contienen miembros estáticos que requieren una inicialización no trivial.

## Ejemplo

Este es un ejemplo de una clase estática que contiene dos métodos que convierten la temperatura de grados Celsius a Fahrenheit y viceversa:

```
public static class ConvertidorTemperatura
{
    public static double CelsiusAFahrenheit(string temperaturaCelsius)
    {
        // Convierte el argumento a tipo double para hacer los cálculos.
        double celsius = Double.Parse(temperaturaCelsius);

        // Convierte de Celsius a Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }
    public static double FahrenheitACelsius(string temperaturaFahrenheit)
    {

```

```

        // Convierte el argumento a tipo double para hacer los cálculos.
double fahrenheit = Double.Parse(temperaturaFahrenheit);

        // Convierte Fahrenheit a Celsius.
double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class PruebaConvertidorTemperatura
{
    static void Main()
    {
        Console.WriteLine("Indique el tipo de conversión a realizar:");
        Console.WriteLine("1. De Celsius a Fahrenheit.");
        Console.WriteLine("2. De Fahrenheit a Celsius.");
        Console.Write(":.");

        string seleccion = Console.ReadLine();
        double F, C = 0;

        switch (seleccion)
        {
            case "1":
                Console.Write("Ingrese la temperatura en grados Celsius: ");
                F = ConvertidorTemperatura.CelsiusAFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperatura en Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Ingrese la temperatura en grados Fahrenheit: ");
                C = ConvertidorTemperatura.FahrenheitACelsius(Console.ReadLine());
                Console.WriteLine("Temperatura en Celsius: {0:F2}", C);
                break;

            default:

```



```

        Console.WriteLine("Por favor seleccione el convertidor a realizar.");
        break;
    }
    // Mantiene la consola abierta para ver el resultado.
    Console.WriteLine("Pulse una tecla para terminar el programa.");
    Console.ReadKey();
}
}

```

```

C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Indique el tipo de conversión a realizar:
1. De Celsius a Fahrenheit.
2. De Fahrenheit a Celsius.
:2
Ingrese la temperatura en grados Fahrenheit: 20
Temperatura en Celsius: -6.67
Pulse una tecla para terminar el programa.

```

## Miembros estáticos

Una clase no estática puede contener métodos, campos, propiedades o eventos estáticos. El miembro estático es invocable en una clase incluso si no se ha creado ninguna instancia de esa clase. *El nombre de clase, y no el nombre de instancia, es el que tiene acceso al miembro estático.* Solo existe una copia de un miembro estático, independientemente del número de instancias que se creen de la clase. Los métodos y propiedades estáticos no pueden tener acceso a los campos y eventos no estáticos de su tipo contenedor y no pueden tener acceso a una variable de instancia de ningún objeto a menos que se pase explícitamente en un parámetro de método.

*Es más habitual declarar una clase no estática con algunos miembros estáticos que declarar toda una clase como estática.* Dos usos comunes de los campos estáticos son mantener un recuento del número de objetos de los que se han creado instancias y almacenar un valor que se debe compartir entre todas las instancias de una clase.

Los métodos estáticos se pueden sobrecargar pero no invalidar, porque pertenecen a la clase y no a una instancia de la clase.

Aunque un campo no se puede declarar como **static const**, un campo **const** es esencialmente estático en su comportamiento. Pertenece al tipo, no a las instancias del tipo.

Por lo tanto, se puede tener acceso a los campos de constante mediante la misma notación **NombreClase.NombreMiembroDeClase** que se utiliza para los campos estáticos. No se requiere ninguna instancia de objeto.

C# no admite el uso de variables locales estáticas (variables que se declaran en el ámbito del método).

Los miembros estáticos de la clase se declaran mediante la incorporación de la palabra clave **static** antes del tipo de valor devuelto del miembro, tal como se muestra en el ejemplo siguiente:

```
public class Automovil
{
    public static int CantidadDeRuedas = 4;
    public static int TamanoTanqueCombustible
    {
        get {return 15;}
    }
    public static void Conducir() {}
    public static event EventType FaltCombustible;

    // Otros campos y propiedades no estáticos...
}
```

Los miembros estáticos se inicializan antes de obtener acceso al miembro estático por primera vez y antes de llamar al constructor estático, si éste existe. Para tener acceso a un miembro estático de la clase, utilice el nombre de la clase en lugar de un nombre de variable para indicar la ubicación del miembro, tal como se muestra en el siguiente ejemplo:

```
static void Main()
{
    Automovil.Conducir();
    int i = Automovil.CantidadDeRuedas;

}
```

Si la clase contiene campos estáticos, proporcione un constructor estático que los inicialice cuando se cargue la clase.

Una llamada a un método estático genera una instrucción de llamada en el lenguaje intermedio de Microsoft (MSIL), mientras que una llamada a un método de instancia genera

una instrucción **callvirt**, que también comprueba las referencias nulas de un objeto. Sin embargo, la mayoría de las veces la diferencia de rendimiento entre ambas no es significativa.

## Clases Abstractas y clases selladas

La palabra clave **abstract** permite crear clases y miembros de clase que están incompletos (como si creara un esqueleto o definición para una clase base), y se deben implementar en una clase derivada.

En otras palabras el código de estas clases debe ser codificado en las clases que hereden dicha clase.

La palabra clave **sealed** permite impedir la herencia de una clase o de ciertos miembros de clase marcados previamente como virtuales.

## Clases y miembros de clase abstractos

Las clases se pueden declarar como abstractas si se incluye la palabra clave **abstract** antes de la definición de clase. Por ejemplo:

```
public abstract class A
{
    // Aquí van los miembros de la clase.
}
```

No se pueden crear instancias de una clase abstracta. El propósito de una clase abstracta es proporcionar una definición común de una clase base que múltiples clases derivadas pueden compartir. Por ejemplo, una biblioteca de clases puede definir una clase abstracta que se utiliza como parámetro para muchas de sus funciones y solicitar a los programadores que utilizan esa biblioteca que proporcionen su propia implementación de la clase mediante la creación de una clase derivada de esta clase abstracta.

Las clases abstractas también pueden definir métodos abstractos. Esto se consigue agregando la palabra clave **abstract** antes del tipo de valor que devuelve el método. Por ejemplo:

```
public abstract class A
{
    public abstract void HacerAlgo(int i); }
```

Los métodos abstractos no tienen ninguna implementación, de modo que la definición de método va seguida por un punto y coma en lugar de un bloque de método normal. Las clases derivadas de la clase abstracta deben implementar todos los métodos abstractos. Cuando una clase abstracta hereda un método virtual de una clase base, la clase abstracta puede reemplazar el método virtual con un método abstracto. Por ejemplo:

```
public class D
{
    public virtual void HacerAlgo(int i)
    {
        // Implementación Original.
    }
}

public abstract class E : D
{
    // Override sin implementación porque es Abstracta.
    public abstract override void HacerAlgo(int i);
}

public class F : E
{
    public override void HacerAlgo(int i)
    {
        // nueva implementación
    }
}
```

Si un método virtual se declara como abstract, sigue siendo virtual para cualquier clase que herede de la clase abstracta. Una clase que hereda un método abstracto no puede tener acceso a la implementación original del método; en el ejemplo anterior, HacerAlgo de la clase F no puede llamar a HacerAlgo de la clase D. De esta forma, una clase abstracta puede obligar a las clases derivadas a proporcionar nuevas implementaciones de método para los métodos virtuales.

# Clases y miembros de clase sellados

Las clases se pueden declarar como selladas si se incluye la palabra clave sealed antes de la definición de clase. Por ejemplo:

```
public sealed class D
{
    // Aquí los miembros de la clase
}
```

Una clase sellada no se puede utilizar como clase base. Por esta razón, tampoco puede ser una clase abstracta. Las clases selladas evitan la derivación. Puesto que nunca se pueden utilizar como clase base, algunas optimizaciones en tiempo de ejecución pueden hacer que sea un poco más rápido llamar a miembros de clase sellada.

Un método, indizador, propiedad o evento de una clase derivada que reemplaza a un miembro virtual de la clase base puede declarar ese miembro como sellado. Esto niega el aspecto virtual del miembro para cualquier clase derivada adicional. Esto se logra colocando la palabra clave sealed antes de la palabra clave override en la declaración del miembro de clase. Por ejemplo:

```
public class D : C
{
    public sealed override void HacerAlgo() { }
}
```