

Programación Web .NET Core

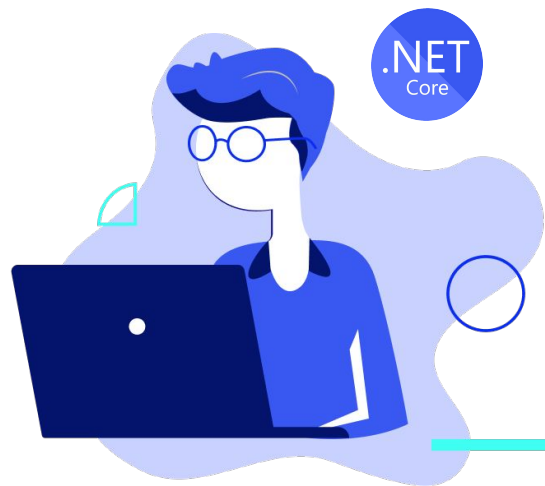
Módulo 2

Repaso de sobrecarga y sobrescritura

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 facilita la implementación de algoritmos que cumplen la misma función pero que poseen distintos 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**, que “**debe**” ser sobreescrito obligatoriamente.

Veamos un ejemplo
en la próxima pantalla.



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

```
abstract class Figura
{
    abstract public int Area();
}

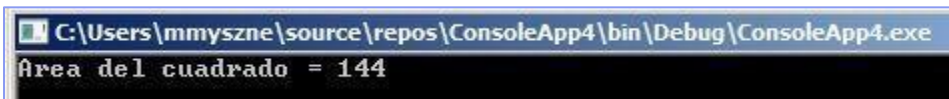
class Cuadrado : Figura
{
    int lado = 0;
    public Cuadrado(int n)
    {
        lado = n;
    }
}
```

```
...  
  
// Se requiere el método Area 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();
}
```



C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
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: En este caso, se 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**:

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.

```
classTestOverride
{
    publicclass Empleado
    {
        publicstring Nombre;
        // pagoBase está definido como Protegido, por lo tanto, solo
        // puede accederse por su clase y clases derivadas.
        protecteddecimalpagoBase;
```

```
...  
  
// 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 sobrescrito.  
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;  
}  
  
// Sobrescribe 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 + " ganó: " +
        empleado1.CalcularPago());
    Console.WriteLine("Empleado " + empleado2.Nombre + " ganó: " +
        empleado2.CalcularPago());
}
```

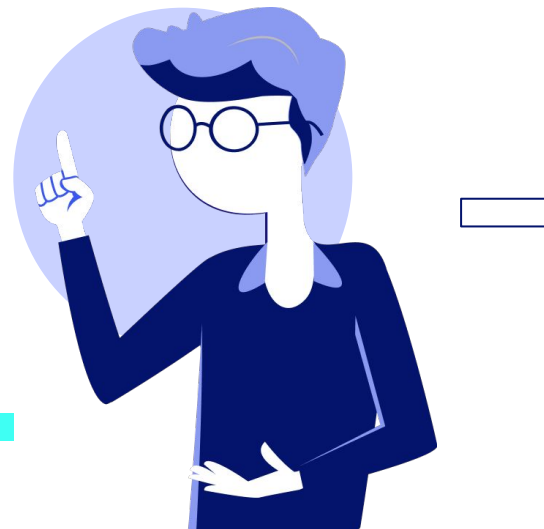


C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Empleado Alice gana: 1500
Empleado Bob gana: 1200

Introducción a relaciones entre objetos

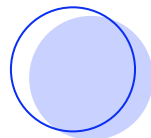
Introducción a relaciones entre objetos

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



Relación jerárquica (Herencia)

Cuando las **clases derivan de las clases más fundamentales**, se dice que tienen una **relación jerárquica**. Las jerarquías de clases son útiles para describir elementos que constituyen un sub-tipo de una clase más general. 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 **Ciente** contiene en forma lógica un objeto **Cuenta**. Observe que el objeto contenedor no contiene en forma física ningún otro objeto.



Uso de colecciones en relaciones de contención simples y múltiples

En las **relaciones de contención** pueden usarse colecciones para representar **relaciones entre objetos de distintas clases**.

Veamos el ejemplo de la derecha, que muestra una relación simple:

```
publicclass Persona
{
    publicstringDocumento;
    publicstringNombre;
    publicstringApellido;
}
publicclass Alumno
{
    public PersonaDatosPersonales;
    publicstringTitulo;
}
```



En otro ejemplo, en el cual un objeto de tipo alumno puede tener 1 ó más cursos, se puede representar una **relación múltiple** así:

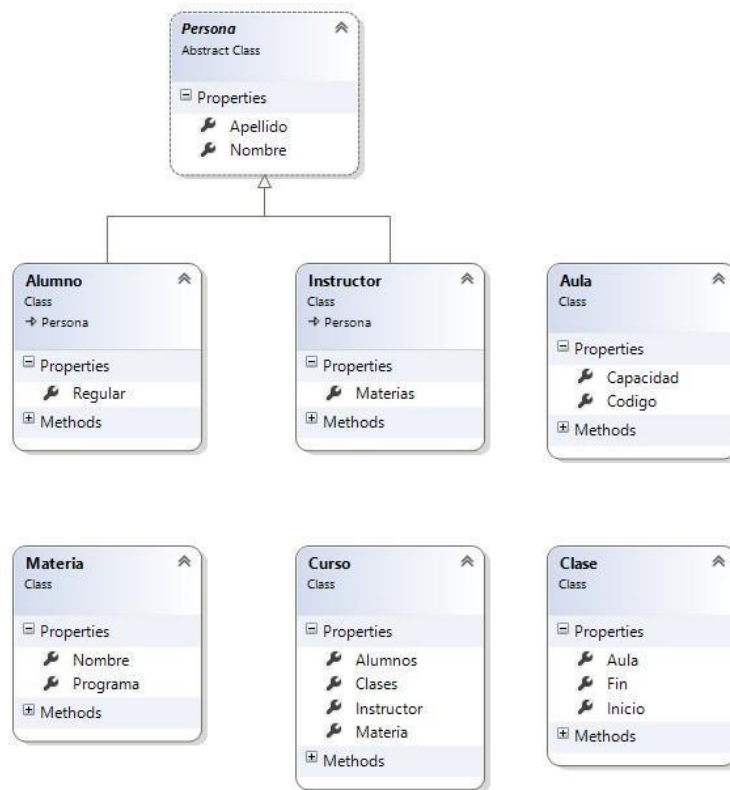
```
publicclass Curso
{
    publicstringNombre;
    publicstringModalidad;
    publicstringTurno;
}

publicclass Alumno
{
    public PersonaDatosPersonales;
    publicstringTitulo;
    publicList<Curso> Cursos;
}
```

En el diagrama de clases que se ve en la siguiente pantalla, se pueden apreciar ambos tipos de relaciones.

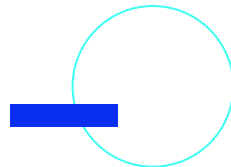


Diagrama de clases



Comencemos por la clase **Persona**: es una clase abstracta con 2 atributos simples. No hay ningún tipo de relación en esta clase.

```
public abstract class Persona
{
    public string Nombre {get; set;}
    public string Apellido {get; set;}
}
```



Alumno hereda de **Persona** y tiene un atributo *booleano*.
En este caso, tampoco hay relaciones.

```
public class Alumno:Persona
{
    public Alumno(string pNombre, string pApellido, bool pRegular)
    {
        this.Nombre = pNombre;
        this.Apellido = pApellido;
        this.Regular = pRegular;
    }
    public bool Regular { get; set; }
}
```



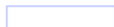
Sin embargo, **Instructor**, que también hereda de **Persona**, tiene un atributo **Materias** que está compuesto por una colección de objetos del tipo **Materia**. Esta es una relación del tipo uno a muchos.

```
public class Instructor:Persona
{
    public Instructor(string pNombre, string pApellido, List<Materia> pMaterias)
    {
        this.Nombre = pNombre;
        this.Apellido = pApellido;
        this.Materias = pMaterias;
    }
    public List<Materia> Materias { get; set; } //Relación muchos a muchos
}
```

Materia tiene 2 atributos simples: no existen relaciones con otros objetos, al igual que la clase **Aula**.

```
public class Materia
{
    public Materia(string pNombre, string pPrograma)
    {
        this.Nombre = pNombre;
        this.Programa = pPrograma;
    }
    public string Nombre { get; set; }
    public string Programa { get; set; }
}
```

```
public class Aula
{
    public Aula(string pCodigo, int pCapacidad)
    {
        this.Codigo = pCodigo;
        this.Capacidad = pCapacidad;
    }
    public string Codigo { get; set; }
    public int Capacidad { get; set; }
}
```



Sin embargo, la clase **Clase** tiene establece una relación uno a uno con la clase **Aula**.

```
public class Clase
{
    public Clase(DateTime pInicio, DateTime pFin, Aula pAula)
    {
        this.Inicio = pInicio;
        this.Fin = pFin;
        this.Aula = pAula;
    }
    public DateTime Inicio { get; set; }
    public DateTime Fin { get; set; }
    public Aula Aula { get; set; } //Relación 1 a muchos
}
```



La clase **Curso** posee las siguientes relaciones:

- **Alumnos** es una colección de **Alumno**.
(Relación 1 a muchos)
- **Clases** es una colección de **Clase**.
(Relación uno a muchos)
- **Instructor** es del tipo **Instructor**.
(Relación uno a uno)
- **Materia** es del tipo **Materia**.
(Relación uno a uno)

Veamos el detalle en
la próxima pantalla.



```
public class Curso
{
    public Curso(Instructor pInstructor, Materia pMateria,
                List<Alumno> pAlumnos, List<Clase> PClases)
    {
        this.Instructor = pInstructor;
        this.Materia = pMateria;
        this.Alumnos = pAlumnos;
        this.Clases = pClases;
    }
    public Instructor Instructor { get; set; } //Relación 1 a muchos
    public Materia Materia { get; set; } //Relación 1 a muchos
    public List<Alumno> Alumnos { get; set; } //Relación muchos a muchos
    public List<Clase> Clases { get; set; } //Relación muchos a muchos
}
```

**¡Sigamos
trabajando!**