Métodos y conceptos de programación

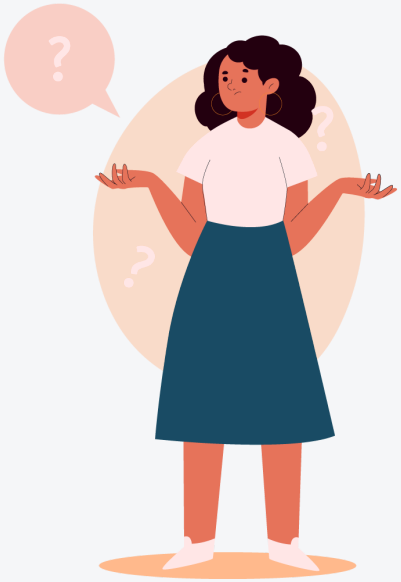
Sitio: Agencia de Habilidades para el Futuro
Curso: Desarrollo de Sistemas Orientado a Objetos 1° D
Libro: Métodos y conceptos de programación

Imprimido por: Eduardo Moreno
Día: domingo, 30 de marzo de 2025, 19:59

Tabla de contenidos

Preguntas orientadoras

- ¿Por qué necesitamos utilizar métodos en la programación orientada a objetos?
- Si tenemos dos métodos con la misma funcionalidad, ¿le podemos poner el mismo nombre?
- ¿Cómo podemos mostrar los datos de un objeto?
- ¿Puede un objeto estar compuesto por otros objetos? Por ejemplo, si miramos nuestro teclado (claramente es un objeto), está compuesto por teclas que también son objetos.



Introducción

ToString

Seguimos conceptualizando

El método ToString() es una función incorporada en el lenguaje C# que se utiliza para convertir un objeto en su representación de cadena de caracteres. Es una forma conveniente de obtener una descripción legible del objeto en forma de texto.

El método ToString() se define en la clase base o "padre" de todos los objetos en C#, llamada System.Object. Por lo tanto, está disponible para todos los objetos y se hereda en todas las clases. Sin embargo, en muchas clases, se anula sobreescribiendo al método ToString() habiendo un "override" para proporcionar una implementación específica y más útil para ese tipo de objeto.

Por defecto, la implementación de ToString() en la clase base System.Object devuelve el nombre completo del tipo del objeto. Por ejemplo, si tienes una instancia de la clase Persona, llamar al método ToString() devolverá algo como "MiNamespace.Persona".

No obstante, en la mayoría de los casos, es recomendable anular el método ToString() en tus propias clases para proporcionar una representación personalizada y más significativa de los objetos.

Al hacerlo, puedes especificar qué información deseas mostrar cuando se llama a ToString() en un objeto de esa clase.

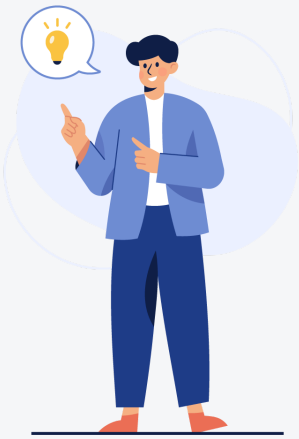
Para anular el método ToString(), debes agregar la siguiente firma en la definición de tu clase:

```
public override string ToString()
{
    // Implementación personalizada aquí
}
```

Dentro de la implementación, puedes construir una cadena de caracteres que contenga la información relevante del objeto. Puedes acceder a los campos, propiedades o cualquier otro miembro de la clase para incluirlos en la representación de cadena.

¡Veamos un ejemplo!

Aquí tienes un ejemplo sencillo que muestra cómo anular el método ToString() en una clase llamada Persona que contiene como atributos el nombre y la edad:



```
internal class Persona
{
    private string nombre;
    private int edad;
```

```
public override string ToString()
{
    return "Nombre: " + nombre + " edad: " + edad;
}
```

En este caso, al llamar al método ToString() en un objeto Persona, se devolverá una cadena de caracteres que muestra el nombre y la edad de la persona.

```
Persona unaPersona = new Persona("Jose", 40);
Console.WriteLine(unaPersona);
```

O bien podemos llamar al método mostrar que hicimos en Ventana

```
Ventana v = new Ventana();
v.Mostrar(unaPersona.ToString());
v.Mostrar(unaPersona.ToString(), 10, 5);
v.Mostrar(unaPersona.ToString(), 10, 10,ConsoleColor.Red);
v.Mostrar(unaPersona.ToString(), 10, 15, ConsoleColor.Red, ConsoleColor.Blue);
Console.ResetColor();
```

En resumen, el método ToString() en C# es utilizado para obtener una representación de cadena de caracteres de un objeto. Puedes anular este método en tus propias clases para proporcionar una representación personalizada y significativa de los objetos.

A continuación, definamos otro concepto esencial en la programación orientada a objetos: asociación simple.

Asociación simple

La asociación simple en programación orientada a objetos (POO) se refiere a una relación entre dos clases o entidades, donde una clase tiene una referencia o utiliza directamente a la otra. Esta relación se establece mediante la creación de una instancia de una clase dentro de otra.

En la asociación simple, una clase se asocia con otra clase para utilizar sus métodos y acceder a sus atributos. La clase que utiliza a la otra se denomina clase cliente o clase principal, mientras que la clase utilizada se conoce como clase proveedora o clase secundaria.

La asociación simple se basa en el principio de reutilización de código y promueve la modularidad y la flexibilidad en el diseño de software. Permite que las clases trabajen juntas de manera independiente, sin depender completamente unas de otras.

¡Relacionemos con un ejemplo! Un ejemplo sencillo de asociación simple podría ser una relación entre una clase Persona y una clase Auto. La clase Persona puede tener un atributo que sea una instancia de la clase Auto, lo que indica que cada persona está asociado con un auto en particular.

A través de esta asociación, la clase Persona puede acceder a los métodos y atributos públicos de la clase Auto. Por ejemplo, la clase Persona puede utilizar un método de la clase Auto para obtener información sobre los datos del auto.

Es importante tener en cuenta que en la asociación simple, las clases están acopladas de manera débil, lo que significa que pueden existir independientemente una de la otra. Si se modificara la implementación de una clase, no afectaría directamente a la otra clase.

La asociación simple en programación orientada a objetos se refiere a la relación entre dos clases donde una clase utiliza o tiene una referencia a la otra. Esta asociación promueve la reutilización de código y la modularidad en el diseño de software, permitiendo que las clases trabajen de forma independiente y flexible.

Ejemplo en C#

Veamos ahora un ejemplo de asociación simple en C#.
Creamos la clase Persona

```
public Persona(string nombre, int edad, Auto auto)
{
    this.nombre = nombre;
    this.edad = edad;
    this.auto = auto;
}

0 referencias
public string Nombre
{
    get { return nombre; }
    set { nombre = value; }
}

0 referencias
public void setNombre(string nombre)
{
    this.nombre = nombre;
}

0 referencias
public string getNombre()
{
    return nombre;
}


0 referencias
public override string ToString()
{
    return "Nombre: " + nombre + " edad: " + edad + " Caracteristicas del auto: " + auto;
}
```

Creamos la clase Auto

```
class Auto
{
    private int anio;
    private string patente, marca, modelo, color;

    1 referencia
    public Auto(int anio, string patente, string marca, string modelo, string color)
    {
        Anio = anio;
        Patente = patente;
        Marca = marca;
        Modelo = modelo;
        Color = color;
    }

    1 referencia
    public int Anio { get { return anio; } set { anio = value; } }
    1 referencia
    public string Patente { get { return patente; } set { patente = value; } }
    1 referencia
    public string Marca { get { return marca; } set { marca = value; } }
    1 referencia
    public string Modelo { get { return modelo; } set { modelo = value; } }
    1 referencia
    public string Color { get { return color; } set { color = value; } }
    0 referencias
    public override string ToString()
    {
        return "Anio: " + anio + " Marca: " + marca + " Modelo: " + modelo + " Color: " + color;
    }
}
```

Desde nuestro Program instanciamos a una Persona llamada Jose y le pasamos que tiene el auto Fiat 600 que instanciamos previamente.

Supongamos que Jose está casado con Andrea y tienen el mismo auto. Entonces a Andrea le pasamos como referencia el mismo auto creado.

```
static void Main(string[] args)
{
    Auto auto = new Auto(2020, "ABC123", "Fiat", "600", "Amarillo");
    Persona unaPersona = new Persona("Jose", 48,auto);
    Persona otraPersona = new Persona("Andrea", 40, auto);
    Console.WriteLine(unaPersona);
    Console.WriteLine(otraPersona);
}
```

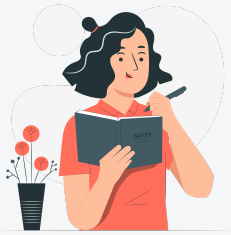
En este ejemplo, tenemos dos clases: Persona y Auto. La clase Persona tiene un atributo llamado auto que es del tipo Auto. Esto establece una asociación simple entre las dos clases.

En el método "Main", creamos una instancia de la clase Auto y la asignamos al atributo auto de la instancia de la clase Persona. Luego, podemos acceder a la información del auto desde el objeto Persona a través de la asociación.

En el método "ToString" de la clase Persona, mostramos el los datos de la persona y también la referencia al auto que dispone asociado.

La asociación simple en C# permite que las clases trabajen juntas y compartan información de manera flexible. Cada instancia de la clase Persona está asociada con una instancia de la clase Auto específica, lo que nos permite acceder a los datos relacionados entre ellas.

Sobrecarga de métodos



A lo largo de la unidad 1, te presentamos algunos conceptos que son esenciales para la programación orientada a objetos. En esta nueva unidad, incorporaremos algunos nuevos.

Sigamos conceptualizando

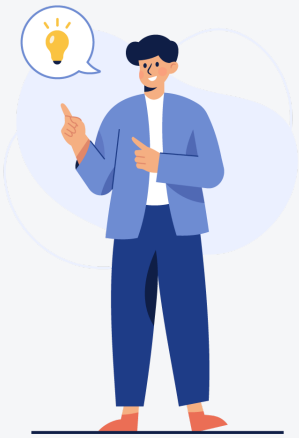
La sobrecarga de métodos es un concepto fundamental en la programación orientada a objetos que nos permite definir múltiples versiones de un método con el mismo nombre pero con diferentes parámetros. Esto nos brinda flexibilidad y nos permite adaptar la funcionalidad de un método según las necesidades específicas de cada situación.

La principal ventaja de la sobrecarga de métodos es la capacidad de reutilizar nombres de métodos para tareas similares pero con diferentes tipos de datos o parámetros. Esto nos evita tener que recordar múltiples nombres de métodos para hacer cosas ligeramente diferentes. En lugar de eso, podemos usar el mismo nombre, lo que hace que nuestro código sea más legible y fácil de mantener.

Cuando sobrecargamos un método, podemos definir diferentes versiones del mismo método con diferentes listas de parámetros.

La firma del método, que incluye el nombre del método y los tipos y orden de los parámetros, **debe ser única** para cada versión del método. De esta manera, el compilador distingue entre las diferentes versiones y llama a la que corresponda según los argumentos que se le pasen.

¡Relacionemos la idea con algo concreto!



Supongamos que estamos trabajando en una clase llamada Calculadora y queremos agregar un método llamado sumar. La sobrecarga de métodos nos permite definir diferentes versiones de este método para manejar diferentes tipos de datos. Podríamos tener una versión que sume dos enteros, otra que sume dos números reales y otra que sume una lista de enteros. Cada versión del método tendría una firma distinta, lo que nos permite usar el mismo nombre pero con diferentes parámetros.

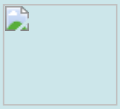
Más tips

Es importante tener en cuenta que la sobrecarga de métodos se basa en la lista de parámetros, no en el nombre de los parámetros ni en el tipo de retorno. Dos métodos no se pueden sobrecargar sólo en función de su tipo de retorno, ya que el compilador no puede distinguirlos por eso. Sin embargo, podemos sobrecargar un método cambiando la lista de parámetros, incluso si el tipo de retorno es el mismo.

Al utilizar la sobrecarga de métodos, debemos tener en cuenta la legibilidad y la claridad del código. Es recomendable utilizar nombres descriptivos para cada versión del método y evitar la sobrecarga excesiva, que puede llevar a confusiones y dificultades para mantener el código en el futuro.

La sobrecarga de métodos es un mecanismo poderoso en la programación orientada a objetos que nos permite definir múltiples versiones de un método con el mismo nombre pero con diferentes parámetros. Esto nos brinda flexibilidad y reutilización de código, lo que resulta en un código más legible y fácil de mantener. Al utilizar la sobrecarga de métodos de manera efectiva, podemos adaptar nuestros métodos a diferentes situaciones y tipos de datos, mejorando la modularidad y la eficiencia de nuestro código.

A continuación, veamos dos ejemplos de sobrecarga de métodos.



Ejemplo 1

Para sobrecargar un método en C#, simplemente necesitamos darle al método el mismo nombre, pero diferentes parámetros. Luego, el compilador elegirá el método correcto para llamar en función de los tipos de parámetros que se pasan al método.

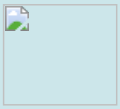
Ejemplo de cómo sobrecargar un método en C#:

public class Ejemplo

```
{
    public int add(int a, int b)
    {
        return a + b;
    }
    public double add (doble a, doble b)
    {
        return a + b;
    }
}
```

En este ejemplo, la clase Ejemplo tiene dos métodos llamados add. El primer método toma dos parámetros int y el segundo método toma dos parámetros double. Ambos métodos devuelven la suma de sus parámetros.

Cuando llama a un método que está sobrecargado, el compilador elegirá el método correcto para llamar en función de los tipos de parámetros que se pasan al método. Por ejemplo, si llama al método add con dos parámetros int, el compilador llamará al primer método add. Si llama al método add con dos parámetros double, el compilador llamará al segundo método add.



Ejemplo 2

En este ejemplo haremos un método para mostrar un texto por pantalla.

Realizaremos una clase llamada Ventana que defina cuatro métodos sobrecargados que muestren un mensaje en la consola.
El primero lo muestra donde se encuentra actualmente el cursor.
El segundo lo muestra en una determinada columna y fila.
El tercero lo muestra en una determinada columna, fila y con un color de letra.
Y por último similar al anterior y le agregamos un color de fondo.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApp7
{
    2 referencias
    class Ventana
    {
        1 referencia
        public void Mostrar(string mensaje)
        {
            Console.Write(mensaje);
        }
        2 referencias
        public void Mostrar(string mensaje, int columna, int fila)
        {
            Console.SetCursorPosition(columna, fila);
            Console.Write(mensaje);
        }

        2 referencias
        public void Mostrar(string mensaje, int columna, int fila, ConsoleColor colorletra)
        {
            Console.ForegroundColor = colorletra;
            Mostrar(mensaje, columna, fila);
        }
        1 referencia
        public void Mostrar(string mensaje, int columna, int fila, ConsoleColor colorletra, ConsoleColor colorfondo)
        {
            Console.BackgroundColor = colorfondo;
            Mostrar(mensaje, columna, fila, colorletra);
        }
    }
}
```

Como podemos observar, hemos definido cuatro métodos llamados Mostrar que difieren en la cantidad de parámetros.
El primero recibe un string y lo muestra en la consola:

```
      public void Mostrar(string mensaje)
       {
           Console.Write(mensaje);
       }
```

El segundo recibe tres parámetros con el mensaje, la columna y fila donde mostrarlo:

```
      public void Mostrar(string mensaje, int columna,int fila)
       {
           Console.SetCursorPosition(columna, fila);
           Console.Write(mensaje);
       }
```

Como vemos la clase Console tiene un método llamado SetCursorPosition que le pasamos la columna y fila donde queremos que se posicione el cursor previo a la salida de datos llamando al método Write.

El tercer método recibe cuatro parámetros y como podemos ver desde dentro de este método llamamos al método Mostrar que recibe tres parámetros:

```
      public void Mostrar(string mensaje, int columna, int fila,ConsoleColor colorletra)
       {
           Console.ForegroundColor = colorletra;
           Mostrar(mensaje, columna, fila);
       }
```

Para cambiar el color de la letra de la Console debemos inicializar la propiedad ForegroundColor.

El último método es similar al tercero pero con un quinto parámetro:

```
      public void Mostrar(string mensaje, int columna, int fila, ConsoleColor colorletra,ConsoleColor colorfondo)
       {
           Console.BackgroundColor = colorfondo;
           Mostrar(mensaje, columna, fila, colorletra);
       }
```

Ejemplo 3

Si tenemos una estructura Persona con los datos de nombre, apellido, edad, etc. y queremos agregar un método para asignar la edad al atributo edad, podríamos definirlo así:

```
using System;
using System.Collections.Generic;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApp8
{
    6 referencias
    internal class Persona
    {
        private string nombre;
        private string apellido;
        private int edad;
        1 referencia
        public Persona(string nombre, string apellido,int edad)
        {
            this.nombre = nombre;
            this.apellido = apellido;
            setEdad(edad);
        }
        1 referencia
        private void setEdad(int e)
        {
            edad = e;
        }
```

Sin embargo, podría interesarnos también que se pueda setear o darle un valor a la edad a partir de la fecha de nacimiento, es decir, que el método reciba la fecha de nacimiento, calcule la edad y la asigne al atributo edad. Para esto, es posible crear otro método con el mismo nombre pero que en vez de recibir un entero, reciba la fecha de nacimiento para calcular con ella la edad y asignarla al atributo de la estructura. Se dice entonces, que el método setEdad() está sobrecargado.

```
internal class Persona
{
    private string nombre;
    private string apellido;
    private int edad;
    1 referencia
    public Persona(string nombre, string apellido,int edad)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        setEdad(edad);
    }
    1 referencia
    public Persona(string nombre, string apellido, int dia, int mes, int anio)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        setEdad(dia,mes,anio);
    }
    1 referencia
    private void setEdad(int e)
    {
        edad = e;
    }
    1 referencia
    private void setEdad(int dia, int mes, int anio)
    {
        var hoy = DateTime.Now;
        edad = hoy.Year - anio;
    }
    0 referencias
    public override string ToString()
    {
        return apellido + ", " + nombre + " edad:" + edad;
    }
}
```

Cuando CSharp encuentra una llamada al método setEdad() analiza primero los parámetros de la llamada y luego llama al método que mejor coincide con esos parámetros. En el siguiente programa se muestra cómo es posible llamar al método setEdad() con distintos tipos de parámetros de forma transparente:

```
using System;

namespace ConsoleApp8
{
    0 referencias
    internal class Program
    {
        0 referencias
        static void Main(string[] args)
        {
            Persona p1 = new Persona("Lionel","Messi",30);
            Persona p2 = new Persona("Angel","Di Maria",14,02,1988);

            Console.WriteLine(p1);
            Console.WriteLine(p2);

        }
    }
}
```

En el ejemplo anterior, los 2 métodos tienen la misma cantidad de parámetros pero podríamos sobrecargar métodos con distinta cantidad de parámetros; incluso también pueden retornar valores de distinto tipo de dato.

En resumen

La sobrecarga de métodos nos permite tener varias versiones de un método con diferentes parámetros, el método toString nos proporciona una representación textual de un objeto y la asociación simple nos permite relacionar clases para lograr un comportamiento más completo y modular.

Estos conceptos son fundamentales en la programación orientada a objetos y nos ayudan a escribir programas más eficientes y mantenibles.