UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

INTRODUCCION A LA PROGRAMACION C#

3 - MÓDULO 2 : FUNDAMENTOS DE LA PROGRAMACION ORIENTADA A OBJETOS (PARTE 2)

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

UNIDAD: 3

MODULO: 2

PRESENTACIÓN: En esta unidad se continuará explicando los fundamentos de la programación orientada a objetos.

OBJETIVOS

Que los participantes logren: Comprender los pilares de la POO y su aplicación en C#

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

TEMARIO

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

Sobrecarga de métodos

La sobrecarga de métodos consiste en poner varios métodos con el mismo nombre en la misma clase, pero siempre que su lista de argumentos sea distinta.

Es decir, no puede haber dos métodos que se llamen igual con la misma lista de argumentos, aunque devuelvan datos de distinto tipo. El compilador sabría a cuál de todas las sobrecargas nos referimos por los argumentos que se le pasen en la llamada, pero no sería capaz de determinar cuál de ellas debe ejecutar si tienen la misma lista de argumentos.

Por ejemplo, dada la siguiente clase,

```
using System;


namespace OOP_EjemplosClase7
{
   public class CuentaCorriente
   {
      protected double saldo=0;

      public double Saldo
      {
         get{ return saldo; }
      }

      public bool Extraccion(double cantidad)
      {
         if (cantidad<=0) return false;

         this.saldo -= cantidad;
         return true;
      }

      public bool Deposito(double cantidad)
      {
         Console.WriteLine("Ejecutando Deposito con un double como argumento");
         if (cantidad <=0) return false;

         this.saldo += cantidad;
```

```
        return true;
    }
            public bool Deposito(float cantidad)
    {
        Console.WriteLine("sobrecarga con float !");
        return true;
    }

    public int Deposito(double cantidad, double argumento2)
    {
            Console.WriteLine("sobrecarga con double, double !");

            return 0;
    }

    public int Deposito(float cantidad, double argumento2)
    {
            Console.WriteLine("sobrecarga con float, double !");

            return 0;
    }


  }
}
```

no podríamos sobrecargar el método Deposito de este modo:

```
    public int Deposito(double cantidad)
    {
        if (cantidad <=0) return 1;
        this.saldo += cantidad;
        return 0;
    }
```

A pesar de devolver un valor int en lugar de un bool, su lista de argumentos es idéntica, por lo que el compilador avisará de un error. Sin embargo, sí podríamos sobrecargalo de estos modos:

```
public bool Deposito(float cantidad)){...}
public int Deposito(double cantidad, double argumento2){...}
public int Deposito(float cantidad, double argumento2){...}
```

Lo que diferencia las listas de argumentos de las diferentes sobrecargas no es el nombre de las variables, sino el tipo de cada una de ellas. Por ejemplo, la siguiente sobrecarga tampoco sería válida:

```
public bool Deposito(double num) //Error. No se puede sobrecargar así
{...}
```

A pesar de que el argumento tiene un nombre distinto (num en lugar de cantidad), es del mismo tipo que el del método del ejemplo, por lo que el compilador tampoco sabría cuál de las dos sobrecargas ejecutar.

Bueno, supongo que ahora vendrá la pregunta: ¿Cuál de todas las sobrecargas válidas ejecutará si efectúo la siguiente llamada?

```
cuenta.Deposito(20.53);
```

Efectivamente, aquí podría haber dudas, ya que el número 200.53 puede ser tanto double, como float. Para números decimales, el compilador ejecutará la sobrecarga con el argumento de tipo double. En el caso de números enteros, el compilador ejecutará la sobrecarga cuyo argumento mejor se adapte con el menor consumo de recursos (int, uint, long y unlong, por este orden). Y ahora vendrá la otra pregunta: ¿y si yo quiero que, a pesar de todo, se ejecute la sobrecarga con el argumento de tipo float? Bien, en ese caso tendríamos que añadir un sufijo al número para indicarle al compilador cuál es el tipo de dato que debe aplicar para el argumento:

```
cuenta.Deposito(20.53F);
```

Sufijos para literales de los distintos tipos de datos numéricos

Son los siguientes:

- L (mayúscula o minúscula): long ó ulong, por este orden.
- U (mayúscula o minúscula): int ó uint, por este orden.
- UL ó LU (independientemente de que esté en mayúsculas o minúsculas): ulong.
- F (mayúscula o minúscula): float.
- D (mayúscula o minúscula): doublé.
- M (mayúscula o minúscula): decimal;

Métodos static

Los métodos *static*, son aquellos que se pueden ejecutar sin necesidad de instanciar la clase donde está escrito.

"Un método estático es un método que existe en una clase como un todo más que en una instancia específica de la clase".

Por lo tanto, el hecho de que el método *Main* tenga que ser *static* no es un capricho, ya que, de lo contrario, el CLR (O Core CLR en el caso de .Net Core) no sería capaz de encontrarlo pues antes de que se ejecute la aplicación, lógicamente, no puede haber instancias de ninguna de las clases que la componen.

Estos métodos suelen usarse para hacer una serie de operaciones globales que tienen mucho más que ver con la clase como tal que con una instancia específica de la misma.

Ejemplos de uso:

- Si tenemos una clase Coche y queremos listar todas las marcas de coches de que disponemos, lo más propio es un método *static*.
- Si representamos la fabricación de Calculadoras, y necesitamos llevar una numeración serializada consecutiva. El próximo número de serie disponible es más propio de la clase en si que de las instancias de la clase Calculadora.

Por lo tanto, los métodos *static* no aparecen como miembros de las instancias de una clase, sino como parte integrante de la propia clase.

Vamos a poner un pequeño programa completo que ilustra el uso de los métodos *static*:

```
namespace OOP_EjemplosClase7
{
    public class VisaDebito
    {
        public static ushort LimiteDiario()
        {
            return 300;
        }

        // otros miembros de clase...
    }
}
```

Clases abstractas

Una clase abstracta es aquella que forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma o acceder a sus miembros estáticos.
Para definir una clase abstracta se antepone *abstract* a su definición, como se muestra en el siguiente ejemplo:

```
public abstract class A
{
   public abstract void F();
}

abstract public class B: A
{
   public void G()
   {
   }
}

class C: B
{
   public override void F()
   {

   }
}
```

Las clases A y B del ejemplo son abstractas, y como puede verse es posible combinar en cualquier orden el modificador *abstract* con modificadores de acceso.

La utilidad de las clases abstractas es que pueden contener métodos para los que no se dé directamente una implementación sino que se deje en manos de sus clases hijas darla.

No es obligatorio que las clases abstractas contengan métodos de este tipo, pero sí lo es marcar como abstracta a toda la que tenga alguno. Estos métodos se definen precediendo su definición del modificador *abstract* y sustituyendo su código por un punto y coma (;), como se muestra en el método F() de la clase A del ejemplo (nótese que B también ha de definirse como abstracta porque tampoco implementa el método F() que hereda de A).

Obviamente, como un método abstracto no tiene código no es posible llamarlo.

Véase que todo método definido como abstracto es implícitamente *virtual*, pues si no sería imposible redefinirlo para darle una implementación en las clases hijas de la clase abstracta donde esté definido. Por ello es necesario incluir el modificador *override* a la hora de darle implementación y es redundante marcar un método como *abstract* y *virtual* a la vez (de hecho, hacerlo provoca un error al compilar)

Ejemplo de aplicación

Supongamos que tenemos que modelar un escenario en donde una empresa tiene dos tipos de empleados: Los fulltime y los freelancers.
Todos los empleados tienen DNI, nombre, apellido y un sueldo.
Los fulltime tienen un salario fijo mensual y los freelancers cobran un valor hora, con lo cual su paga mensual depende de la cantidad de horas trabajadas en ese periodo.

Definamos la clase Empleado

```
namespace empleados
{
   public abstract class Empleado
   {
      public int DNI{get;}
      public string Nombre{get;}
      public string Apellido{get;}

      protected Empleado(int DNI, string nombre, string apellido){
         this.DNI =DNI;
         Nombre =nombre;
         Apellido = apellido;

      }

      public abstract double SueldoMensual();
   }

}
```

Definamos la clase EmpleadoFulltime

```
namespace empleados
{
   public class EmpleadoFulltime : Empleado
   {
      public double Sueldo;

      public EmpleadoFulltime(int DNI, string nombre, string apellido, double sueldo)
               : base(DNI, nombre, apellido){

         Sueldo =sueldo;

      }

      public override double SueldoMensual()
      {
         return Sueldo;
      }
   }

}
```

Definamos la clase Freelancer

```
namespace empleados
{
   public class Freelancer : Empleado
   {
      public double ValorHora;
      public int HorasMensuales;

      public Freelancer(int DNI, string nombre, string apellido, double valorHora, int horasMensuales)
               : base(DNI, nombre, apellido)
      {
         ValorHora =valorHora;
         HorasMensuales =horasMensuales;
      }

      public override double SueldoMensual()
      {
         return ValorHora * HorasMensuales;
      }
   }

}
```

Ahora, hagamos uso de este modelo

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

namespace empleados
{
   class Program
   {
      static void Main(string[] args)
      {
         EmpleadoFulltime empleado1 =new EmpleadoFulltime(123456, "Osvaldo", "Olmos", 100.34);
         EmpleadoFulltime empleado2 =new EmpleadoFulltime(999999, "Mario", "Mandzucick", 100000.50);
         Freelancer empleado3 = new Freelancer(1111111, "Luka", "Modric", 20.50, 100);

         List<Empleado> empleados =new List<Empleado>();
         empleados.Add(empleado1);
         empleados.Add(empleado2);
         empleados.Add(empleado3);

         foreach (Empleado empleado in empleados)
         {
            Console.WriteLine($"El empleado: {empleado.Apellido} cobra: {empleado.SueldoMensual()}");
         }
      }
   }
}
```

La salida por consola de la ejecución seria la siguiente:

```
Osvaldos-MacBook-Air:OOP-EjemplosClase7-1 osvaldoolmos$ dotnet run
El empleado: Olmos cobra: 100.34
El empleado: Mandzucick cobra: 100000.5
El empleado: Modric cobra: 2050
Osvaldos-MacBook-Air:OOP-EjemplosClase7-1 osvaldoolmos$ dotnet run
El empleado: Olmos cobra: 100.34
El empleado: Mandzucick cobra: 100000.5
El empleado: Modric cobra: 2050
```

Conclusiones:

Usamos una clase abstracta cuando queremos mantener en un solo lugar el estado y comportamiento de dos o más clases que están relacionadas y cuando la clase base representa en nuestro modelo algo que no puede ser instanciado.

La clase primigenia: System.Object

En .Net Core todos los tipos que se definan heredan implícitamente de la clase System.Object predefinida en la Librería de Clases Base, por lo que dispondrán de todos los miembros de ésta. Por esta razón se dice que System.Object es la raíz de la jerarquía de objetos de .Net Core.

A continuación vamos a explicar cuáles son estos métodos comunes a todos los objetos:

public virtual bool Equals(object o) :

Se usa para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro. Devuelve true si ambos objetos son iguales y false en caso contrario.
La implementación que por defecto se ha dado a este método consiste en usar igualdad por referencia para los tipos por referencia e igualdad por valor para los tipos por valor. Es decir, si los objetos a comparar son de tipos por referencia sólo se devuelve true si ambos objetos apuntan a la misma referencia en memoria dinámica, y si los tipos a comparar son tipos por valor sólo se devuelve true si todos los bits de ambos objetos son iguales, aunque se almacenen en posiciones diferentes de memoria.

Como se ve, el método ha sido definido como *virtual* , lo que permite que los programadores puedan redefinirlo para indicar cuándo ha de considerarse que son iguales dos objetos de tipos definidos por ellos. De hecho, muchos de los tipos incluidos en la Biblioteca de clases base, cuentan con redefiniciones de este tipo, como es el caso de *string* , quien aun siendo un tipo por referencia, sus objetos se consideran iguales si apuntan a cadenas que sean iguales carácter a carácter (aunque referencien a distintas direcciones de memoria dinámica)

El siguiente ejemplo muestra cómo hacer una redefinición de *Equals*() de manera que aunque los objetos Empleado sean de tipos por referencia, se considere que dos Empleados son iguales si tienen el mismo DNI:

```
public override bool Equals(object obj)
{
    //
    // See the full list of guidelines at
    //   http://go.microsoft.com/fwlink/?LinkID=85237
    // and also the guidance for operator== at
    //   http://go.microsoft.com/fwlink/?LinkId=85238
    //

    if (obj == null || GetType() != obj.GetType())
    {
        return false;
    }

    // TODO: write your implementation of Equals() here
    return (obj is Empleado) && this.DNI == (obj as Empleado).DNI;
}
```

Hay que tener en cuenta que es conveniente que toda redefinición del método *Equals*() que hagamos cumpla con una serie de propiedades que muchos de los métodos incluidos en las distintas clases de la librería de clases base esperan que se cumplan.

Estas propiedades son:

Reflexividad: Todo objeto ha de ser igual a sí mismo. Es decir, x.Equals(x) siempre ha de devolver true.

Simetría: Ha de dar igual el orden en que se haga la comparación. Es decir, x.Equals(y) ha de devolver lo mismo que y.Equals(x) .

Transitividad: Si dos objetos son iguales y uno de ellos es igual a otro, entonces el primero también ha de ser igual a ese otro objeto. Es decir, si x.Equals(y) e y.Equals(z) entonces x.Equals(z) .

Consistencia: Siempre que el método se aplique sobre los mismos objetos ha de devolver el mismo resultado.

Tratamiento de objetos nulos: Si uno de los objetos comparados es nulo (null) , sólo se ha de devolver true si el otro también lo es.

Hay que recalcar que el hecho de que redefinir Equals() no implica que el operador de igualdad (==) quede también redefinido.

public virtual int GetHashCode() :

Devuelve un código de hash que representa de forma numérica al objeto sobre el que el método es aplicado.

Siempre que se redefina Equals(), hay que redefinir GetHashCode(). De hecho, si no se hace el compilador informa de la situación con un mensaje de aviso.

Las regla a seguir es:

- Si dos objetos son iguales, deben tener el mismo hashcode.
- El hashcode debe estar basado en un campo inmutable del objeto. Es decir, que no cambie durante todo su ciclo de vida.

En el caso de la clase Empleado:

```
// override object.GetHashCode
public override int GetHashCode()
{
   return this.DNI;
}
```

public virtual string ToString() :

Devuelve una representación en forma de cadena del objeto sobre el que se el método es aplicado, lo que es muy útil para depurar aplicaciones ya que permite mostrar con facilidad el estado de los objetos.
La implementación por defecto de este método simplemente devuelve una cadena de texto con el nombre de la clase a la que pertenece el objeto sobre el que es aplicado. Sin embargo, como lo habitual suele ser implementar *ToString*() en cada nueva clase que es defina, a continuación mostraremos un ejemplo de cómo redefinirlo en la clase Empleado:

```
public override string ToString()
{
   return $"{DNI} - {Apellido}, {Nombre}";
}
```

public static bool ReferenceEquals(object objeto1, object objeto2):

Indica si los dos objetos que se le pasan como parámetro se almacenan en la misma posición de memoria dinámica. A través de este método, aunque se hayan redefinido *Equals*() y el operador de igualdad (==) para un cierto tipo por referencia, se podrán seguir realizando comparaciones por referencia.entre objetos de ese tipo en tanto que redefinir de *Equals*() no afecta a este método. Por ejemplo, dada la anterior redefinición de *Equals*() para objetos Empleado :

```
static void Main(string[] args)
    {
        EmpleadoFulltime empleado1 =new EmpleadoFulltime(123456, "Shubert", "Lemos",
                                                100.34);
        EmpleadoFulltime empleado2 =new EmpleadoFulltime(123456, "Mario", "Mandzucick",
                                                100000.50);


        Console.WriteLine($"{empleado1.Equals(empleado2)}");
        Console.WriteLine($"{Object.ReferenceEquals(empleado1, empleado2)}");
    }
```

La salida por consola es la siguiente:

```
Osvaldos-MacBook-Air:OOP-EjemplosClase7-1 osvaldoolmos$ dotnet run
True
False
```

Determinación de tipo. Operador is

Dentro de una rutina polimórifica que, como la del ejemplo anterior, admita parámetros que puedan ser de cualquier tipo, muchas veces es conveniente poder consultar en el código de la misma cuál es el tipo en concreto del parámetro que se haya pasado al método en cada llamada al mismo. Para ello C# ofrece el operador *is* , cuya forma sintaxis de uso es:

<expresión> is <nombreTipo>

Este operador devuelve true en caso de que el resultado de evaluar <expresión> sea del tipo cuyo nombre es <nombreTipo> y false en caso contrario.

El operador as

El operador *as* se usa así:

<expresión> as <tipoDestino>

Lo que hace es devolver el resultado de convertir el resultado de evaluar <expresión> al tipo indicado en <tipoDestino> Por ejemplo, para almacenar en una variable p el resultado de convertir un objeto t a el tipo Empleado se haría:

p = t as Empleado;

- Sólo es aplicable a tipos referencia.
- En caso de que se solicite hacer una conversión inválida *as* devuelve null

En la redefinición del *Equals* de Empleado, utilizamos los dos operadores haciendo una conversión de tipos segura:

```
public override bool Equals(object obj)
{
    //
    // See the full list of guidelines at
    //   http://go.microsoft.com/fwlink/?LinkID=85237
    // and also the guidance for operator== at
    //   http://go.microsoft.com/fwlink/?LinkId=85238
    //

    if (obj == null || GetType() != obj.GetType())
    {
        return false;
    }

    // TODO: write your implementation of Equals() here
    return (obj is Empleado) && this.DNI == (obj as Empleado).DNI;
}
```

Fecha	Versión	Observaciones
15/07/2018	1.0	Versión Original
02/06/2018	2.0	Adaptación para netcore
09/06/2018	2.1	Mejoras en la redacción