UD 2 ASP.NET Core. Características. El lenguaje C#.
2.3 Repaso de OO.

En esta unidad repasaremos algunos conceptos básicos sobre la OO, y concretamente el modo en que se implementan en C#.

Se recomienda utilizar como referencia el capítulo 3 y 4 del siguiente libro:

https://www.syncfusion.com/ebooks/csharp

Además del apartado correspondiente en la guía de programación de C#, parte de la documentación oficial del lenguaje:

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/index

Es importante recordar, una vez más, que los contenidos de esta unidad son una breve presentación de las características del lenguaje C#. El contenido de este documento, como los anteriores, es una mera introducción, estando toda la documentación disponible en los enlaces y el libro recomendados. Cada alumno deberá consultar dichas fuentes, en función de los conocimientos que haya adquirido con anterioridad en el módulo Programación.

Fundamentos de la programación OO.

Hoy en día, el paradigma de programación más extendido y utilizado en la programación orientada a objetos. Gran parte de los lenguajes más populares o bien, como pueden ser java o C#, son completamente OO, o bien, como en el caso de php, si no fueron diseñados inicialmente como OO, han añadido estructuras para permitir su uso.

Parte del éxito de la OO es que se trata de una técnica de programación que se acerca más al modo en que pensamos las personas. Proporciona herramientas que permiten al programador representar los elementos do problema (objetos) de forma general, sin limitarse a un problema concreto. Se puede hacer lo mismo con otros paradigmas, como la programación estructurada tradicional, pero usando OO obtendremos un código más limpio y reutilizable.

Clases.

Podemos pensar en cualquier elemento de la vida real como un sistema de objetos: un televisor, un teléfono, un coche, o una persona pueden ser objetos con unas determinadas características y capacidades. Un televisor tiene un tamaño, resolución, ... y pode hacer tareas como encender, apagar, cambiar de canal, ... una persona tiene una estatura, color de pelo, color de ojos y otras muchas características que diferencian a una persona de otra. Al proceso de modelar o reducir un objeto del mundo real a un conjunto de características que nos interesan para la resolución de nuestro problema se le llama **abstracción**, y es uno de los elementos clave de la OO.

Cuando tenemos un elemento del mundo real, somos capaces de saber de qué tipo es en función de sus características. Si vemos un teléfono, sabemos distinguir que se trata de un teléfono y no de un televisor por las características que lo definen. Esto, qué puede parecer obvio, es la base de la POO. La idea o abstracción que tenemos de lo que representa a una persona sería una *clase*, mientras que una persona concreta sería una *instancia* u *objeto* de esa clase.

Un ejemplo clásico podría ser una aplicación para la gestión de cuentas bancarias. Tendríamos una clase *Cuenta* con propiedades como número de cuenta, saldo, tipo de cuenta y titular/es. *Cliente* sería una clase con propiedades como NIF, nombre completo, fecha de nacimiento, dirección, teléfono, .. Y *Movimiento* sería una clase con información relativa a los movimientos que se realizan en esa cuenta. Todos estos objetos establecerían relaciones entre si y trabajarían juntos para realizar las diferentes tareas que se necesitan en la aplicación, modelando de este modo las entidades que intervienen en el mundo real.

La POO tiene como ventajas la reutilización de código, la mejor comprensión y legibilidad del mismo, la flexibilidad, a facilidad para dividir las tareas e la capacidad de ampliar una aplicación.

C# es un lenguaje totalmente orientado a objetos, eso quiere decir que todo el código en un programa en C# debe encontrarse dentro de una clase. Una clase encapsula tanto propiedades que describen el estado de un objeto como métodos que implementan el comportamiento de dicho objeto.

Creación de clases.

Una clase es un molde o plantilla a partir del cual podremos crear múltiples objetos, con similares características. Las variables que están definidas dentro de una clase reciben el nombre de atributos o miembros y las funciones definidas en una clase reciben el nombre de métodos. Los métodos de la clase pueden obtener o modificar el estado del objeto almacenado en los atributos del objeto.

A clase define los atributos y métodos comunes a los objetos dese tipo, pero cada objeto tendrá sus propios valores. Una clase debe estar definida para poder crear objetos (instancias) que pertenezcan a esta.

Las clases se definen empreñando a palabra clave `class` seguida del nombre da clase y un bloque de código entre llaves. El nombre de la clase, según las convenciones de código de C# empleará la notación Pascal, es decir, comenzará por mayúscula, y si está formada por más de una palabra, estas irán juntas y con mayúscula la primera letra de cada una de ellas. No se utilizará el guión '_' para separar las palabras. Esta notación será familiar para los programadores de java ya que sigue la misma convención.

Ya que haremos referencia a las notaciones a emplear en diversas ocasiones, se adjunta un enlace a un artículo referente a estas:

https://en.wikipedia.org/wiki/Camel_case

Una clase se guardará en un fichero de código que la contenga, en el caso de C# es un fichero con la extensión .cs, y aunque no es obligatorio, es recomendable que el nombre del fichero coincida con el de la clase que lo contiene, ya que facilita la localización del código en el proyecto y la claridad.

A continuación se muestra un ejemplo de definición de clase

```
// Class definition.
public class CustomClass
{
   // Class members.
   //
   // Properties.
   public int Number { get; set; }

   // Instance Constructors.
   public CustomClass()
   {
       Number = 0;
   }

   // Methods.
   public int Multiply(int num)
   {
       return num * Number;
   }
}
```

Os métodos definidos en una clases son compartidos por todos los objetos de esa clase. La diferencia es que, obviamente, cada uno accede a los atributos o propiedades del propio objeto. Cuando queremos acceder a algún elemento, bien un atributo, propiedad o método del propio objeto en el que se está ejecutando el código utilizamos el operador *this.*

Por ejemplo, en la clase anterior, dentro del método *Multiply()* accedemos a la propiedad *Number*, en este caso, al no haber ningún parámetro o variable local con el mismo nombre no habría problema, pero si quisiéramos especificar que se trata de la propiedad *Number* dentro del objeto actual escribiríamos *this.Number*.

Creación de objetos.

Un objeto una copia creada a partir de una plantilla que es la definición de la clase, y se almacena en una variable cuyo tipo es la propia clase. Para crear un objeto(instancia) de una clase determinada se utiliza el operador new, usando la siguiente sintaxis en nuestro ejemplo:

CustomClass c = new CustomClass();

CustomClass es el tipo de nuestra variable, que en este caso es una clase que hemos creado, y la variable c es una instancia (un objeto) de la clase *CustomClass*. La palabra clave new crea un objeto de tipo *CustomClass*, iniciando todos sus atributos que se guarda en c.

Una vez que tenemos una variable que almacena un objeto de un tipo determinado, podemos acceder a sus propiedades y métodos públicos utilizando el operador '.':

```
c.Multiply(3);
```

Es importante recordar que la especificación de C#, en su guía de estilo, indica que los métodos utilizarán la notación Pascal Case, al contrario de muchos otros lenguajes como java o c/c++. Por esta razón al principio muchos programadores lo olvidan, pero es importante seguir las normas de estilo de un lenguaje ya que de lo contrario, al trabajar con otros programadores, identificar los elementos de un programa puede resultar más complicado. Así que recuerda, los nombres de métodos en C#, **empiezan por mayúscula!!!**

Constantes.

Una clase puede tener constantes, que son valores fijos establecidos en la definición de la clase, que no se pueden modificar en tiempo de ejecución.

Para declarar una constante no interior de una clase usamos la palabra reservada *const* antes del nombre de la constante. Este nombre comenzará en mayúscula para diferenciar las constantes de los atributos y variables. En este caso aunque la norma dice que se utilice la notación Pascal Case también para las constantes, hay muchos programadores que prefieren escribir el nombre de las constantes todo con mayúsculas.

A continuación se muestra un ejemplo de declaración de constantes:

```
class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.

}
```

El valor de una constante está asociado a la clase, y no se hace una copia para cada objeto que se cree. Por esta razón no se puede acceder a las constantes desde un objeto, para ello en su lugar utilizamos el nombre de la clase, en nuestro ejemplo sería algo así: *Constants.Pi*

Encapsulación.

Uno de los fundamentos de la OO es la encapsulación. Se base la restricción de acceso a los miembros de un objeto. La idea es convertir un objeto en una especie de 'caja negra' en la que no nos interesa lo que contiene dentro, y sólo como utilizarla. De nuevo se basa en el principio de abstracción del mundo real, por ejemplo, si utilizamos un cronómetro, disponemos de dos botones, el primero nos permite poner en marcha el cronómetro, mientras que el segundo permite pararlo, o pulsándolo mientras ya está parado, ponerlo a 0. Para utilizar el cronómetro no necesitamos saber como funciona por dentro, ni que tiene guardado en cada momento, ni mucho menos tenemos acceso a modificar nada de lo que que contiene. El cronómetro está diseñado para ser utilizado mediante esos dos botones, que constituyen la 'interfaz' del objeto con el exterior, y es el modo en el que debe utilizarse.

Al igual que el cronómetro del ejemplo, nuestros objetos definirán su comportamiento de modo que sólo expondremos al exterior los métodos y propiedades que queramos que puedan ser modificados desde el exterior, siendo éstos siempre los necesarios para el correcto uso de los mismos, pero nunca permitiendo el acceso ni la modificación de nada más, de modo que al tener controladas las vías de acceso al estado del objeto, el código será mucho más robusto y menos susceptible a errores difíciles de localizar.

Para esto utilizamos los modificadores de acceso, que permiten controlar la visibilidad de los distintos elementos de un objeto:

- Público (public): Cualquiera pode acceder a los atributos y métodos declarados como public. Por convención se establece que sólo debemos hacer públicos los métodos que el objeto expone al exterior para permitir su funcionamiento (como los botones del cronómetro del ejemplo).
- Privado(private): Por convención, en OO, TODOS los atributos deben ser declarados como privados, de modo que no permitamos su acceso o modificación desde el exterior, ya que no tendríamos ningún tipo de control sobre los mismos. El acceso a los atributos y métodos declarados como private está restringido a la clase en la que fueron creados.
- Protegido(protected): Únicamente pueden acceder a estos atributos e métodos la propia clase y las clases derivadas de ella. Esto está relacionado con la herencia, de la que hablaremos más adelante.

Atributos y métodos estáticos / de instancia.

Los atributos y métodos de una clase están encapsuladas en el objeto/clase y únicamente son accesibles a través de la clase u objeto, no globalmente.

Los atributos y métodos por defecto son lo que se denomina ‘de instancia’, es decir, forman parte de cada objeto. Cando creamos un objeto, se crea espacio en memoria y se guarda una copia de cada uno de los atributos de este, de modo que cada objeto tiene su propia copia de dichos atributos, configurando de este modo el estado del objeto. Como ya vimos anteriormente, se puede acceder a ellos a través del objeto, utilizando el operador ‘.’

Hay ocasiones en las que queremos que ciertos atributos o métodos estén disponibles sin necesidad de instanciar la clase (non hace falta crear un objeto para acceder a ellos). En C#, tal y como ocurre con otros lenguajes, para definir atributos y métodos de clase o estáticos se utiliza la a palabra clave `static`.

Este concepto puede llevarse incluso más allá, y definir una clase que únicamente contenga atributos o métodos estáticos. De este modo podemos declarar una clase como estática, no siendo de este modo posible crear objetos de este tipo, y por el contrario utilizando sus métodos siempre a través del nombre de la clase.

A continuación podemos ver un ejemplo de clase estática y su uso:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(double temperatureC)
    {
        // Convert Celsius to Fahrenheit.
        double fahrenheit = (temperatureC * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(double temperatureF)
    {
        // Convert Fahrenheit to Celsius.
        double celsius = (temperatureF - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        var tf = TemperatureConverter.CelsiusToFahrenheit(30);
        Console.WriteLine("Temperatura en Fahrenheit: {0}", tf);
    }
}
```

Constructores.

Las clases tienen por defecto un método especial denominado constructor, que permite inicializar el estado de los objetos (dar valores a los métodos) cando se instancia un objeto.

Si no especificamos un constructor, como hemos dicho, se utilizaría el constructor por defecto y los diferentes atributos son inicializados a su valor por defecto dependiendo del tipo de datos de cada uno. Para crear un constructor personalizado, creamos un método cuyo nombre sea el de la clase, especificando los parámetros que creamos oportunos. Este método será llamado en el momento en que se crea un nuevo objeto con *new*, y las tareas que realiza habitualmente son las de inicializar el estado del objeto.

A continuación podemos ver un ejemplo de constructor:

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }
}
```

Para crear un nuevo objeto libro haríamos, en este caso:

```
Person p = new Person("Kirk", "James T.");
```

Propiedades en C#.

Como ya indicamos anteriormente, la encapsulación es uno de los pilares de la OO. No debemos permitir acceder directamente a los atributos desde el exterior, ya que no tendríamos el control sobre qué tipo de manipulaciones sobre los datos se pueden realizar, pudiendo dejar nuestro estado en un estado inconsistente.

Por esta razón, los lenguajes OO proporcionan un mecanismo que permite obtener o modificar el estado de un objeto de un modo controlado. Esto se hace proporcionando una serie de métodos especiales cuyo objetivo es el de permitir obtener o modificar el valor de cada uno de los atributos de una clase.

Los que hayan trabajado en java en el módulo de programación habrán trabajado con los métodos denominados getters y setters. En C# utilizaremos otro mecanismo similar, que proporciona el mismo nivel de control pero una sintaxis más sencilla y natural. Son las llamadas **propiedades**.

Una propiedad es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados descriptores de acceso. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos. Por medio de las propiedades podemos especificar también si un atributo o miembro puede ser accedido en modo lectura, escritura o ambos, especificando los elementos get o set de la propiedad.

Vamos a utilizar de nuevo nuestra clase *Person*, de modo que contenga un atributo *age*, y una propiedad que controle el acceso al mismo:

```
public class Person
{
   private int age;

   public int Age
   {
        get { return age; }
        set {
                if(value < 0) {
                        age = 0;
                }
                else {
                        age = value;
                }
        }
   }
}
```

Comencemos por el nombre. La guía de C# indica que el nombre de la propiedad debe ser el mismo que el del miembro pero comenzando por mayúscula, de modo que cuando vemos nuestro código, sabemos que después de un operador '.', si lo que tenemos después empieza con minúscula se trata de un campo o atributo (una variable), mientras que si comienza por mayúscula se trataría de un método o una propiedad (código). En nuestro caso al tratarse del atributo *age*, la propiedad equivalente se llamará *Age*. El equivalente en java serían los métodos get*Age()* y *setAge()*.

A continuación, definimos los bloques get y/o set. El código de cada uno de los bloques determinará cómo se leen o escriben los datos en el atributo desde el exterior. El mismo mecanismo se utiliza para definir que se puede o no hacer con el atributo, por ejemplo, si queremos que sea una propiedad de sólo lectura basta con no definir el bloque set.

En nuestro ejemplo vemos que el bloque get devuelve sin más el valor del atributo, mientras que el método set añade una comprobación de seguridad que garantiza que no podamos establecer edades negativas. El valor pasado a la propiedad se obtiene con la palabra clave *value*.

Una vez definida una propiedad, la utilización de la misma se lleva a cabo con una sintaxis similar a si estuviéramos accediendo directamente a una variable que contiene un valor, permitiendo un código mucho más claro e intuitivo, a pesar de que en realidad estemos ejecutando un fragmento de código. De este modo para leer y modificar la edad de un objeto haríamos lo siguiente:

```
var p = new Person();
p.Age = -5; // incorrecto, será arreglado por el código set

Console.WriteLine(p.Age); // mostraría el valor 0
```

Primero creamos un nuevo objeto de tipo *Person*, luego establecemos su edad (del mismo modo que asignamos un valor a una variable, aunque en realidad se está ejecutando el bloque set de la propiedad) y por último obtenemos la edad y la mostramos por pantalla.

Propiedades autoimplementadas.

En muchas ocasiones, nuestras propiedades son simples mecanismos para conservar el criterio de encapsulación y no añadimos ningún tipo de código adicional de comprobación más allá de devolver o modificar el atributo al que dan acceso. Al mismo tiempo, el atributo que protegen no es necesario, ya que se puede acceder al mismo a través de las propiedades con el mismo resultado.

Para este caso se introdujeron en C# las llamadas propiedades autoimplementadas. Lo que hacemos es declarar una propiedad de un tipo determinado y especificamos el tipo de acceso permitido: get, set o get/set. Cuando el compilador detecta una propiedad definida de este modo se encarga de crear automáticamente un atributo del mismo tipo, que es en realidad donde se almacenará el estado y el código para acceder al mismo:

```
public string Name { get; set; }
```

Con el siguiente código declaramos una propiedad autoimplementada de lectura y escritura. El compilador creará automáticamente un atributo de tipo string y generará el código que permite leer los valores de ese atributo y modificar el mismo.

Uso de objetos.

En C#, un objeto es un tipo de datos por referencia, contrariamente a los tipos de datos primitivos, que son datos por valor. Esto quiere decir que si creamos un objeto y este es pasado como parámetro a un método en el que se modifique el estado del mismo, en realidad estamos modificando el valor del objeto original.

Vamos a recordar la diferencia entre los tipos de datos por valor y por referencia en el siguiente ejemplo:

```
class Program
{
    static void Main()
    {
        int n = 1;
        Increment(n);
        Console.WriteLine(n);
    }

    private static void Increment(int number)
    {
        number++;
    }
}
```

En este ejemplo, declaramos una variable de tipo entero con el valor 1. A continuación llamamos a un método *Increment()* que aumenta el valor del argumento recibido en 1. Por último mostramos en pantalla el valor de la variable. En este caso, el valor mostrado en la salida sería '1', pese a haber llamado al método primero, esto es porque al tratarse de un tipo por valor, en realidad, al pasar el argumento se hace una copia del mismo, que es lo que recibe el método. De esta forma, el valor que se incrementa es la copia y no el valor almacenado en *n*, razón por la que la salida seguirá siendo '1'.

```
class Program
{
    static void Main()
    {
        var p = new Person();
        p.Age = 22;
        IncrementAge(p);
        Console.WriteLine(p.Age);
    }

    private static void IncrementAge(Person person)
    {
        person.Age++;
    }
}
```

En este otro ejemplo, sin embargo, pasamos un objeto como argumento del método. En este caso, al tratarse de un tipo por referencia, en lugar de copiarse el valor, lo que se pasa al método es una referencia a la dirección de memoria en la que se encuentra el objeto, es decir, estamos pasando un enlace al propio objeto. Al incrementar la edad del argumento recibido, estamos incrementando la edad del objeto original, razón por la que la salida de este programa sería '23'.

Herencia.

La herencia, junto con la encapsulación y el polimorfismo, es una de las tres características principales de la programación orientada a objetos. La herencia permite crear clases nuevas que reutilizan, extienden y modifican el comportamiento que se define en otras clases. La clase cuyos miembros se heredan se denomina *clase base* y la clase que hereda esos miembros se denomina *clase derivada*. Una clase derivada solo puede tener una clase base directa, pero la herencia es transitiva. Si ClaseC se deriva de ClaseB y ClaseB se deriva de ClaseA, ClaseC hereda los miembros declarados en ClaseB y ClaseA. Para indicar una herencia se utiliza el operador ':'.

Veamos un ejemplo con una clase *Person* y otra *Student* que hereda de la primera:

```
class Person
{
    protected string firstName;
    protected string lastName;
}

class Student : Person
{
    private DateTime enrollmentDate;
}
```

Así *Student* automáticamente contiene todos los miembros de *Person* así como sus métodos, con lo cual un alumno tendrá los tres métodos: `firstName`, `lastName` y `enrollmentDate`.

La herencia permite reutilizar eficientemente el código. Podemos decir que un *Student* es un tipo de *Person*, y estaremos reutilizando el mismo código en dos clases distintas haciendo el código más ligero.

Las nuevas clases que heredan se conocen como *subclases*. La clase de la que heredan se llama *clase base* o *superclase*. Los nuevos objetos que se instancien a partir de la subclase son también objetos da superclase. Esto permitiría, por ejemplo, definir un método que reciba un argumento de tipo *Person*. Más adelante podríamos tener un objeto de tipo *Student* que podría ser pasado a dicho método, debido a que al ser una subclase, un objeto *Student* es también un *Person*. Esta es la base del polimorfismo.

Impedir la herencia.

En algún caso puede que queramos evitar que se pueda heredar de una clase. Para ello, en la declaración de la clase la definimos como sellada o *sealed* :

```
sealed class Person
```

Se puede sellar la clase, como hemos hecho en el ejemplo, para impedir que se herede de ella, o simplemente un método para evitar que sea sobreescrito.

Clases abstractas e interfaces.

Clases abstractas.

Las clases abstractas se declaran con la palabra clave *abstract*, y son similares a cualquier clase normal, excepto en dos aspectos:

- NO se pueden crear objetos a partir de ellas, una clase abstracta no puede ser instanciada.
- Una clase abstracta puede incorporar métodos abstractos, que son aquellos para los que sólo existe declaración, dejando la implementación para las clases hijas o derivadas.

Lo normal es que declaremos una clase como abstracta cuando sabemos que no va a ser instanciada y su función es servir de clase base para otra mediante la herencia. Cualquier clase que defina un método abstracto debe ser declarada como abstracta (al no tener implementado algún método, no se podrán instanciar objetos a partir de ella).

Interfaces.

Los interfaces permiten establecer las características que debe cumplir una clase. Permiten definir qué métodos deben ser declarados en una clase de forma similar a las clases abstractas. Esto es lo que se denomina contrato de la clase, ya que especifica lo que una clase debe cumplir para ser considerada del tipo del interface.

La principal diferencia es que no se puede heredar de un interface. Un interface se implemente, lo que permite heredar de una clase base e implementar una o más interfaces a la vez.
Un interface es el recurso ideal para la implementación del polimorfismo, ya que únicamente declaran los métodos que deben ser codificados en las clases que os implementan, es decir, definen el comportamiento.

Declarar un interface es similar a una clase, substituyendo la palabra `class` por `interface`.
Un interface no permite definir atributos a implementar en las clases, solo los métodos y propiedades.

Por ejemplo, imaginemos un coche y un cronómetro. En principio no tienen nada en común, pero tanto uno como otro se pueden arrancar o parar en un momento dado. Por lo tanto, podríamos definir un interface que especifique este comportamiento, obligando a las clases que lo implementen a implementar estos dos métodos.

```
interface IStartable
{
    void Start();
    void Stop();
}
```

Los nombres de interfaces en C#, comienzan por I por convención, para distinguirlos de las clases. A continuación definimos los métodos que tiene que implementar una clase para que pueda ser considerada como un objeto de tipo *IStartable*.

A continuación, en la declaración de la clase se especifica que se implementará la interface, del mismo modo que cuando se hereda una clase, poniendo ':' después del nombre de clase y especificando a continuación los interfaces que se implementan, separados por comas. Todos los métodos definidos en el interface deberán implementarse en la clase, si no se producirá un error de compilación.

```
class Car : IStartable
{
    public void Start()
    {
    }

    public void Stop()
    {
    }
}
```

Cada clase puede implementar puede implementar tantas interfaces como quiera, con la única excepción de que no puede implementar interfaces que tengan métodos con el mismo nombre.

Un interface es un contrato que la clase debe cumplir. Al implementar todos los métodos declarados en el interface se asegura la interoperabilidad entre clases. Si se sabe que una clase implementa un interface determinado, se conoce el nombre de sus métodos y que parámetros necesitan.

Namespaces.

En C# no se pueden tener dos clases con el mismo nombre, aunque se encuentren en ficheros diferentes de nuestra aplicación. Para evitar este problema se utilizan los espacios de nombres o namespaces. De esta forma el nombre único de una clase está formado por su `namespace` y por el nombre de la propia clase. De este modo sólo necesitamos asegurarnos de que no tenemos más de una clase con el mismo nombre en un único `namespace`, pero si podríamos utilizar otra clase con un nombre igual que pertenezca a un `namespace` diferente.

Para declarar un espacio de nombres debemos usar la palabra clave `namespace` seguido del nombre que le queremos dar.

Para evitar colisionar nombres de clases y espacios de nombres se recomienda utilizar como `namespace` un esquema jerárquico. La recomendación es la de utilizar un nombre de dominio en orden inverso, del mismo modo que se hace en java. Sin embargo muchos desarrolladores optan por un esquema más simple, que incluye el nombre de la empresa/desarrollador y a continuación el de la aplicación/librería. Con eso debería ser más que suficiente.

```
namespace dwes.ejemplo
```

En este caso definimos un espacio de nombres con el nombre de la asignatura, seguido de la aplicación (aquí no hay aplicación si no un simple ejemplo).

Por defecto, una clase que se encuentra en un espacio de nombres determinado sólo puede acceder al resto de clases de ese mismo espacio de nombres, incluso aunque estas se hayan declarado como *public*. Para permitir la utilización de clases de otro espacio de nombres incluiremos tantas líneas *using* como sea necesario para incluir los espacios de nombres que contengan las clases que nos interesa utilizar. Es habitual incluir el espacio de nombres *System*, que contiene las clases base que proporciona .NET.

```
using System;
using System.Text;
```

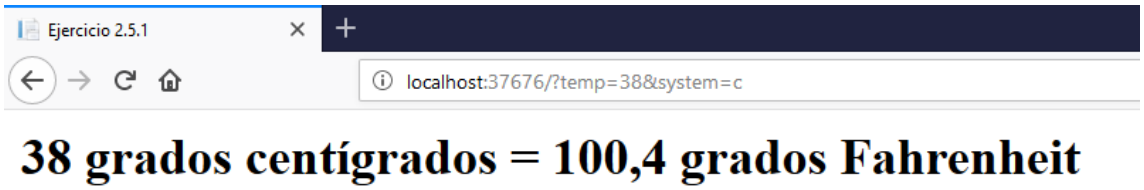
Ejercicio 2.5.1.

Modifica el ejercicio 2.3.1, que convertía una temperatura en grados centígrados a grados Fahrenheit.

Los cambios consistirán en lo siguiente:

- La temperatura se recogerá de la url con un parámetro llamado *temp*, en lugar de *cent*.
- Tendremos otro parámetro *system*, que indique si se trata de una temperatura en grados centígrados o grados Fahrenheit, mediante los valores 'c' y 'f' respectivamente.
- Se creará una clase estática que contenga dos métodos, uno para pasar la temperatura de cada uno de los sistemas al otro.

La aplicación funcionará del mismo modo, mostrando el valor en el sistema original y el valor en el sistema al que se ha convertido, siendo la salida similar a la que sigue:

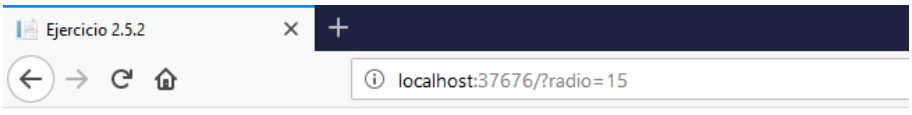


38 grados centígrados = 100,4 grados Fahrenheit

Ejercicio 2.5.2.

Para la realización de este segundo ejercicio crearemos una clase que represente una esfera. Esta clase tendrá un atributo que almacene el radio de la misma. También ofrecerá un método que calcule y devuelva el volumen de la esfera (4/3 * PI * radio^3).

Para probar el funcionamiento de la aplicación pasaremos el radio como parámetro en la url y se mostrará una página en el navegador que ofrezca la información de la esfera, tal y como se muestra a continuación:



Información de la esfera:

- Radio: 15
- Volumen: 10602,8752058656

Ejercicio 2.5.3.

Escribir una clase que represente a los polinomios de la forma ax^5+bx^3+cx+d y que tenga un método que devuelva el valor para un valor determinado de x, y un constructor que reciba los cuatro coeficientes.

A partir de esta clase crearemos una aplicación web que recibirá por su url los valores de los coeficientes a, b, c y d, además del valor de x, siendo todos números enteros.

A continuación mostrará en una página el resultado del polinomio, de modo similar a la imagen de ejemplo: