Tema 8.3

Descargar estos apuntes en pdf o html

Índice

Serialización de Objetos (Definiciones)

Persistencia

Se define por persistencia en el mundo de la POO, como la **capacidad que tienen los objetos de sobrevivir al proceso padre que los creo**. Esto decir, que su ciclo de vida excede de la del programa que lo instanció.

La persistencia permite al programador pues almacenar, transferir y recuperar fácilmente el estado de los objetos.

¿Cómo podemos conseguir la persistencia?

La forma más común de conseguirlo es mediante la **serialización**.

La serialización es el proceso de convertir el estado de un objeto a un formato que se pueda **almacenar** o **transportar**. Normalmente el proceso producirá una **secuencia** de bytes o texto de marcado.

El complemento de la serialización es la **deserialización**, que convierte una secuencia de bytes o texto de marcado a un objeto. Ambos procesos pues permiten almacenar y transferir fácilmente datos.

¿A qué serializaremos una clase?

.NET ofrece dos tecnologías de serialización:

- La **serialización binaria** conserva la fidelidad de tipos, lo que resulta útil para conservar el estado de un objeto entre distintas llamadas a una aplicación. Puedes serializar un objeto en una secuencia de bytes que cómo hemos visto podemos pasar a disco, la memoria, a través de la red, etc.
  - **Ejemplos**:
    Compartir un objeto entre distintas aplicaciones '*serializándolo*' en el Portapapeles.
    Hacer **streaming** de vídeo serializando objetos con información de vídeo comprimido a través de la red.
- La **serialización a lenguajes de marcado** como:
  - La **serialización a XML** sólo serializa las propiedades públicas y los campos, y no conserva la fidelidad de tipos. Esto resulta útil cuando desea proporcionar o consumir datos sin restringir la aplicación que utiliza los datos.
  - La **serialización a alguna notación de objetos** estándar cada vez más comunes y cuya función sería la misma que el XML pero menos '*verbosas*' y más fáciles de leer y/o modificar para humanos. Los más comunes son:
    - **JSON**: Definir configuraciones o consumo de datos a través de microservicios web o bases de datos NoSQL.
    - **YAML**: Muy usado para definir archivos de configuración de sistemas.

¿Qué interfaz podemos usar para realizar el proceso de serialización?

Aunque **la mayoría de lenguajes** ya la traen implementada. Si tuviéramos implementar nosotros las operaciones para serializar objetos. Podría ser algo parecido a esto...

```
public class <NuestraClase>
{
    public void Serializa(Stream flujo);
    public static <NuestraClase>? Deserializa(Stream flujo);
}
```

A la hora de serializar un objeto llamaríamos a su método `Serializa(Stream flujo)` y este a su vez a los Serializa de los objetos y tipos que contenga, así sucesivamente.
Muchos lenguajes como Java o C# solucionan la serialización de forma sencilla, ya que al serializar un objeto contenedor,

este a su vez serializa mediante un mecanismo de **'*reflexión*'** y de forma trasparente aquellas referencias a objetos que contiene. Lo mismo sucede al cargar o deserializar un objeto.

Durante este proceso, los campos público y privado del objeto y el nombre de la clase, incluido el ensamblado que contiene la clase, se convierten en una secuencia de bytes que, a continuación, se escribe en una secuencia de datos. Cuando, después, el objeto se deserializa, se crea una copia exacta del objeto original.

Con este fin C# me ofrece el marcar mis clases como serializables a través de un **'*atributo*'**.

Atributos en .NET

Un **Atributo** en .NET es una etiqueta de la sintaxis `[nombre]` que podremos aplicar a un ensamblado, clase, constructor, enumeración, campo, interfaz, método, etc… y que **genera información en el ensamblado** en forma de metadatos heredando de la clase `Attribute`.

Por ejemplo, si queremos realizar una simple serialización binaria etiquetaremos **la clase a serializar y todas las que contenga** con el atributo ya definido `[Serializable]`, sobre la definición de la clase.

Serialización a binario

Supongamos la siguiente clase `Alumno` con el atributo `[Serializable]`

```
[Serializable]
public class Alumno
{
    private string nombre;
    private string apellido;
    private int edad;

    public Alumno(string nombre, string apellido, int edad)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
}
```

Para posteriormente serializar el tipo deberemos utilizar un formateador, el más común es el `IFormatter`, que se utiliza de la siguiente manera:

```
IFormatter f = new BinaryFormatter();
f.Serialize(<medioalmacenamiento>, <objetoaserializar>);
```

Ejemplo a partir de la clase `Alumno` anterior:

```
    public static void Main()
    {
        Alumno a = new("Pepa", "Pérez", 25);
        using FileStream s = new("Dato.bin", FileMode.Create, FileAccess.Write);
        IFormatter f = new BinaryFormatter();
        f.Serialize(s, a);
    }
```

Si examinamos en Hexadecimal el fichero serilaizado nos habrá generado algo similar a esto...

```
00000000: 00 01 00 00 00 FF FF FF FF 01 00 00 00 00 00 00    ................
00000010: 00 0C 02 00 00 00 3E 45 6A 65 6D 70 6C 6F 2C 20    ......>Ejemplo,.
00000020: 56 65 72 73 69 6F 6E 3D 31 2E 30 2E 30 2E 30 2C    Version=1.0.0.0,
00000030: 20 43 75 6C 74 75 72 65 3D 6E 65 75 74 72 61 6C    .Culture=neutral
00000040: 2C 20 50 75 62 6C 69 63 4B 65 79 54 6F 6B 65 6E    ,.PublicKeyToken
00000050: 3D 6E 75 6C 6C 05 01 00 00 00 06 41 6C 75 6D 6E    =null......Alumn
00000060: 6F 03 00 00 00 06 6E 6F 6D 62 72 65 08 61 70 65    o.....nombre.ape
00000070: 6C 6C 69 64 6F 04 65 64 61 64 01 01 00 08 02 00    llido.edad......
00000080: 00 00 06 03 00 00 00 04 50 65 70 61 06 04 00 00    ........Pepa....
00000090: 00 06 50 C3 A9 72 65 7A 19 00 00 00 0B             ..PC)rez.....
```

De forma análoga realizaremos la **deserialización**.

```
    public static void Main()
    {
        using FileStream s = new("Dato.bin", FileMode.Open, FileAccess.Read);
        IFormatter f = new BinaryFormatter();
        Alumno a = f.Deserialize(s) as Alumno ?? new Alumno("Desconocido", "", 0);
        Console.WriteLine(a);
    }
```

Saber si una clase es serializable

Podemos utilizar el API de Reflexión para consultar los atributos de un tipo. Una posible forma sería:

```
bool esSerializable = typeof(Alumno).Attributes.ToString().IndexOf("Serializable") > 0;
```

Excluir campos o propiedades de la serialización

Una clase a menudo contiene **campos que no se quieran serializar**. Por ejemplo, **campos calculados** o campos con información que no queremos guardar. Para que un campo no se serialize, deberemos aplicarle el atributo `[NonSerialized]`.

```
    [Serializable]
    public class Alumno
    {
        private string nombre;
        private string apellido;
        private int edad;
7       [NonSerialized] // Excluirá el campo nombreCompleto
                        // de la serialización y deserialización.
        private string nombreCompleto;

        public Alumno(string nombre, string apellido, int edad)
        {
            this.nombre = nombre;
            this.apellido = apellido;
            this.edad = edad;
            nombreCompleto = $"{nombre} {apellido}";
        }
    }
```

Definiendo la Serialización como operaciones de la clase

Si quisiéramos refactorizar el código para pasar la **'*responsabilidad*'** de la serialización a la clase `Alumno`. Tendríamos que implementar las operaciones que definimos al principio dentro de la propia clase.

Ejemplo más completo de serialización para ver esto último:

```
[Serializable]
class Alumno
{
    private string nombre;
    private string apellido;
    private int edad;

    public Alumno(string nombre, string apellido, int edad)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    public void SerializaABinario(Stream s)
    {
        new BinaryFormatter().Serialize(s, this);
    }

    public static Alumno? DeserializaDeBinario(Stream s)
    {
        return new BinaryFormatter().Deserialize(s) as Alumno;
    }

    public override string ToString()
    {
        return $"Nombre: {nombre}\nApellido: {apellido}\nEdad: {edad}";
    }
}
```

```
static class Programa
{
    static void MuestraAlumnos()
    {
        using FileStream s = new("alumnos.bin", FileMode.Open, FileAccess.Read);
        while (s.Position < s.Length)
        {
8           Alumno a = Alumno.DeserializaDeBinario(s);
            if (a == null)
            {
                Console.WriteLine("Error leyendo alumnos/as.");
                break;
            }
            else
                Console.WriteLine(a);
        }
    }

    static void Main()
    {
        // Serializamos datos de tres alumnos.
        using (FileStream s = new("alumnos.bin", FileMode.Append, FileAccess.Write))
        {
24          new Alumno("Pepa", "Pérez", 25).SerializaABinario(s);
            new Alumno("María", "Peláez", 22).SerializaABinario(s);
26          new Alumno("Rosa", "López", 26).SerializaABinario(s);
        }
        MuestraAlumnos();
    }
}
```

Versiones de un objeto serializado

En muchas ocasiones el programa evoluciona y **nuevos requerimientos** hacen que añadamos nuevos campos. Esto hace necesaria la posibilidad de mantener diferentes versiones de un objeto y **asegurar la compatibilidad hacia atrás** del mismo.
Esta posibilidad está contemplada en C# a través de marcar ciertos campos con **atributos** específicos.

Por ejemplo, supongamos que queremos guardar un nuevo objeto `Direccion` junto con los datos del `Alumno`.

Podemos crear una clase dirección `[Serializable]` de forma análoga a como hicimos la de alumno...

```
// Puesto que también se va a serializar por ser una agregación
// de Alumno también deberemos
// marcarla como serializable aunque no añadamos opciones
// de serialización por separado.
[Serializable]
class Direccion
{
    private string calle;
    private int numero;
    private string ciudad;
    private string codigoPostal;

    public Direccion()
    {
        calle = "Desconocida";
        numero = 0;
        ciudad = "Desconocida";
        codigoPostal = "0000";
    }

    public Direccion(string calle, int numero, string ciudad, string codigoPostal)
    {
        this.calle = calle;
        this.numero = numero;
        this.ciudad = ciudad;
        this.codigoPostal = codigoPostal;
    }

    public override string ToString()
    {
        return $"{calle} {numero} {ciudad} {codigoPostal}";
    }
}
```

A continuación, añadimos el nuevo campo dirección a la clase `Alumno`. Indicando con el atributo `[OptionalField(VersionAdded = 2)]` que se tratará de un campo opcional que solo estará a partir de la **versión 2** del la misma.

Además, podemos marcar con el atributo `[OnDeserializing]` un método que se ejecutará durante el proceso de '*deserialización*' y inicializará el campo a una **instáncia por defecto** por si acaso el mismo no existe en la versión del objeto que estamos deserializando.

```
[OnDeserializing]
private void SetDirección(StreamingContext sc)
{
    direccion = new Direccion();
}
```

Por tanto, la clase alumno modificada podría quedar así ...

```
    [Serializable]
    class Alumno
    {
        private string nombre;
        private string apellido;
        private int edad;

8       [OptionalField(VersionAdded = 2)]
9       private Direccion direccion;

        public Alumno(string nombre, string apellido, int edad, Direccion direccion)
        {
            this.nombre = nombre;
            this.apellido = apellido;
            this.edad = edad;
            this.direccion = direccion;
        }

19      [OnDeserializing]
        private void SetDirección(StreamingContext sc)
        {
            direccion = new Direccion();
23      }

        // Métodos de seriralización omitidos para abreviar ...
        public override string ToString()
        {
            return $"Nombre: {nombre}\nApellido: {apellido}\nEdad: {edad}\n" +
29                 $"Dirección: {direccion}";
        }
    }
```

Imaginemos que ahora modificamos el `Main` de tal manera que añadimos objetos `Alumno` con el nuevo dato al fichero `alumnos.bin` y posteriormente '*deserializamos*' todos los objetos `Alumno`, mostrándolos con la mismo método `MuestraAlumnos()`

```
static void Main()
{
    // Serializamos datos de tres alumnos.
    using (FileStream s = new("alumnos.bin", FileMode.Append, FileAccess.Write))
    {
        new Alumno("Juani", "Ortiz", 21,
                   new Direccion("Marqués de Molins", 58, "Alicante", "03004")
                   ).SerializaABinario(s);
        new Alumno("Marcos", "Jiménez", 20,
                   new Direccion("Cerámica", 24, "Alicante", "03010")
                   ).SerializaABinario(s);
    }
    MuestraAlumnos();
}
```

EL programa sabrá recuperar versiones anteriores del objeto `Alumno` mostrando instanciándolo a la información por defecto que hemos definido.

```
    Nombre: Rosa
    Apellido: López
    Edad: 26
4   Dirección: Desconocida 0 Desconocida 0000
    Nombre: Juani
    Apellido: Ortiz
    Edad: 21
8   Dirección: Marqués de Molins 58 Alicante 03004
```