Tema 10
Serialización

Sumario

10 Persistencia de objetos

10.1 ¿Por qué la persistencia?

Para que la información utilizada por un programa esté disponible para una ejecución posterior, utilizamos la persistencia.

En programación, la persistencia es la acción de preservar la información de un objeto de forma permanente (guardado), pero a su vez también se refiere a poder recuperar la información del mismo (leerlo) para que pueda ser nuevamente utilizado.

10.2 Serialización / Deserialización

La serialización consiste en un proceso de codificación de un objeto en un medio de almacenamiento como puede ser un archivo, en un formato humanamente legible como XML o JSON, entre otros.

La deserialización consiste en el proceso contrario, a partir de un archivo se puede usar para crear un nuevo objeto que es idéntico en todo al original

La serialización es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones.

10.3 Serializar a un fichero de texo de forma “manual”

La serialización manual implica convertir los datos de un objeto en una representación de texto que pueda ser almacenada o transmitida y posteriormente reconstruida en su estado original.

En este ejercicio, se usa un formato de texto para escribir y leer objetos desde un archivo. Cada objeto se convierte en una cadena con valores separados por punto y coma, permitiendo que el programa almacene y recupere la información sin necesidad de bibliotecas externas.

Este método es útil cuando se requiere simplicidad y compatibilidad con otros sistemas.

10.3.1 Ejercicios propuestos

10.3.1.1 Serializar “a mano”

Se debe implementar una clase base llamada "Persona" con atributos básicos como nombre y edad. A partir de esta clase, se deben crear dos clases derivadas: "Estudiante" y "Profesor".

La clase "Estudiante" debe incluir un atributo adicional para el curso.

La clase "Profesor" debe incluir un atributo para la asignatura que imparte.

El programa debe permitir almacenar varios objetos de estas clases en una lista genérica (puedes introducirlos manualmente, no hace falta pedir los datos).

Posteriormente, la lista debe ser guardada en un archivo de texto, donde cada línea contenga

la información de una persona en un formato legible y estructurado como este ejemplo

```
Alumno;Juan Sin Nombre;22;1DAMA
Profesor;Pablo Coches;45;Programación
Persona;Gran Colorado;65
```

Al ejecutarse nuevamente el programa, debe poder leer el archivo y reconstruir la lista de personas. Finalmente, el programa debe mostrar por consola la información de las personas cargadas.

10.4 Serializar a un fichero XML utilizando System.Xml.Serialization

10.4.1 Sin clases derivadas

No necesitamos marcar la clase a serializar con la anotación [serializable] y podemos serializar arrays de objetos o listas con <generics>

Para crear un objeto serializable utilizamos:

```
XmlSerializer Serializer = new XmlSerializer(typeof(List<Persona>));
```

y para serializar utilizamos ese objeto seriallizador pasándole el fichero y la List<Objeto> la lista de objetos a serializar:

```
Serializer.Serialize(stream, ListaDeObjetos);
```

Para desserializar utilizamos el objeto serializer.Deserialize() pasandole el fichero a leer y devuelve en este caso una Lista de Personas:

```
List<Persona> personas =(List<Persona>)serializer.Deserialize(stream);
```

IMPORTANTE: Todas las clases deben ser públicas y tener un constructor vacío

```
using System;
using System.Xml.Serialization;
using System.IO;
using System.Collections.Generic;

namespace Ejercicio_10_xmlSerializer
{
    class Program
    {
       private static List<Persona> personas = new List<Persona>();
       private static string fichero = "../../../personas.xml";

       static void Main(string[] args)
       {
            poblar();
            Guardar(fichero, personas);
            Cargar(fichero).ForEach(p => Console.WriteLine(p));
       }
       private static void poblar()
       {
            personas.Add(new Persona("Juan", "Pérez"));
            personas.Add(new Persona("Pedro", "Domeq"));
       }
      public static void Guardar(string fichero, List<Persona> objetos)
      {
            StreamWriter stream = new StreamWriter(fichero);
            XmlSerializer Serializer = new XmlSerializer(typeof(List<Persona>));
            Serializer.Serialize(stream, objetos);
            stream.Close();
      }
      public static List<Persona> Cargar(string fichero)
      {

            Stream stream = new FileStream(fichero,FileMode.Open);
            XmlSerializer serializer = new mlSerializer(typeof(List<Persona>));
            List<Persona> personas =(List<Persona>)serializer.Deserialize(stream);
            stream.Close();
            return (personas);
      }
      public class Persona
      {
            public string Nombre { get; set; }
            public string Apellidos { get; set; }
            public Persona() { }
            public Persona(string nombre, string apellidos)
            {
```

```
                Nombre = nombre;
                Apellidos = apellidos;
            }
            public override string ToString()
            {
                return (this.Nombre + ", " + this.Apellidos);
            }
        }
}
```

10.4.2 Ejercicios propuestos:

(10.3.2.1) A partir del ejemplo anterior añade un atributo dni a la clase Persona y haz las modificaciones necesarias para que funcione, debes controlar también las posibles errores añadiendo try-catch donde sea necesario, crea un menú para añadir, listar, eliminar y guardar (serializar) personas
(10.3.2.2) En el ejercicio 8.3.4 habías ampliado la "base de datos de ficheros", de modo que los datos se leyeran desde fichero. Crea una versión alternativa que emplee serialización.

10.4.3 Con clases derivadas

Ahora un ejemplo algo más complejo con una clase virtual Persona y dos clases derivadas que heredan de esta Profesor y Alumno que añaden atributos propios

Hay que temer en cuenta lo siguiente:

1. Todas las clases deben de ser public
2. Añadir constructor vacio para todas las clases (lo exije la librería System.Xml.Serialization)
3. Cuando llamamos al constructor para crear un objeto XmlSerializer le pasamos dos parámetros: el tipo base y un array con los tipos derivados

```
Serializer = new XmlSerializer(typeof(List<Persona>),new Type[] { typeof(Alumno),
typeof(Profesor) });
```

```
using System;
using System.Xml.Serialization;
using System.IO;
using System.Collections.Generic;

namespace xml_serializar_clases_derivadas {

    class Program
    {
        static List<Persona> personas = new List<Persona>();
        static string fichero = "../../../personas.xml";

        static void Main(string[] args)
        {
            Poblar();
            Guardar(fichero, personas);
            Cargar(fichero).ForEach(x => Console.WriteLine(x));
        }
        public static void Poblar()
        {
```

```
            personas.Add(new Profesor("Juan", "Pérez", "Matemáticas"));
            personas.Add(new Alumno("Pedro", "Domeq", "1º DAM"));
        }
        public static void Guardar(string fichero, List<Persona> objetos)
        {
            StreamWriter stream = new StreamWriter(fichero);
             //añadimos un array al constructor con las clases derivadas
            XmlSerializer Serializer = new XmlSerializer(typeof(List<Persona>),
                new Type[] {typeof(Alumno),typeof(Profesor) });
            Serializer.Serialize(stream, objetos);
            stream.Close();
        }
        public static List<Persona> Cargar(string fichero)
        {
             //añadimos un array al constructor con las clases derivadas
            XmlSerializer serializer = new XmlSerializer(typeof(List<Persona>),
            new Type[] { typeof(Alumno), typeof(Profesor) });
            Stream stream = new FileStream(fichero, FileMode.Open);
            List<Persona> personas = (List<Persona>)serializer.Deserialize(stream);
            stream.Close();
            return (personas);
        }
    }
    public abstract class Persona
    {
        public string Nombre { get; set; }
        public string Apellidos { get; set; }

        public Persona()  {}
        public Persona(string nombre, string apellidos)
        {
            Nombre = nombre;
            Apellidos = apellidos;
        }
        public override string ToString()
        {
            return (this.GetType().Name+";"+this.Nombre + ";" + this.Apellidos);
        }
    }
    public class Alumno : Persona
    {
        public string Curso { get; set; }

        public Alumno(){ }
        public Alumno(string nombre, string apellidos, string curso) : base(nombre, apellidos)
        {
            this.Curso = curso;
        }
        public override string ToString()
        {
            return (base.ToString()+";"+ this.Curso);
        }
    }
    public class Profesor : Persona
    {
        public string Departamento { get; set; }

        public Profesor()   { }
        public Profesor(string nombre, string apellidos,string departamento)
                    :base(nombre, apellidos)
        {
            this.Departamento = departamento;
        }
        public override string ToString()
        {
            return (base.ToString() + ";" + this.Departamento);
```

```
        }
    }

}
```

10.4.4 Ejercicios propuestos:

(10.5.4.1) Añade al ejercicio de los trabajadores (6.8.1) la posibilidad de guardar sus datos en formato XML.

(10.5.4.2) A partir del ejemplo anterior añade un atributo dni y además realiza lo siguiente:

1. añade una clase PersonalAdministrativo que hereda de persona y que añade la fecha de nacimiento de tipo DateTime
2. Crea un menú para poder añadir (persomas, profesores, alumnos y personal administrativo), listar, borrar y guardar (serializar)

10.5 Serializar a un fichero JSON utilizando System.Text.Json

En la actualidad, el formato XML está siendo en muchos casos desplazado como formato de intercambio de datos por el formato JSON (JavaScript Object Notation).

No necesitamos marcar la clase a serializar con la anotación [serializable] y podemos serializar arrays de objetos o listas con <generics>

En ese caso, deberíamos añadir un "using" distinto:

using System.Text.Json;

Y nos ayudaríamos de la clase "JsonSerializer".

Como detalle adicional para poder emplear este formateador, necesitarás que los datos sean propiedades, en vez de atributos accesibles a través de "getters" y "setters" convencionales:

10.5.1 Sin clases derivadas

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


class Program
{
       static List<Persona> personas = new List<Persona>();
       static string fichero = "../../../personas.json";

    static void Main(string[] args)
       {
              Poblar();
              Guardar(fichero, personas);
              personas = Cargar(fichero);
              personas.ForEach(p => Console.WriteLine(p));
       }
       private static void Poblar()
       {
        personas.Add(new Persona("Juan", "Pérez"));
        personas.Add(new Persona("Pedro", "Domeq"));
    }
       public static void Guardar(string fichero, List<Persona> objetos)
       {
              string jsonString =  JsonSerializer.Serialize(objetos);
              File.WriteAllText(fichero, jsonString);
       }
       public static List<Persona> Cargar(string fichero)
       {
              string jsonString = File.ReadAllText(fichero);
              List<Persona> personas = JsonSerializer.Deserialize<List<Persona>>(jsonString);
              return (personas);
       }
}
public class Persona
{
       public string Nombre { get; set; }
       public string Apellidos { get; set; }

       public Persona()     {}
       public Persona(string nombre, string apellidos)
       {
```

```
                Nombre = nombre;
                Apellidos = apellidos;
        }
        public override string ToString()
        {
                return (this.Nombre + ", " + this.Apellidos);
        }
}
```

10.5.2 Ejercicios propuestos:

(**10.6.1.1)** A partir del ejemplo anterior vamos un programa capaz de serializar/deserializar una lista de Animales controlando los posibles errores con try-catch

- Clase Animal:
  - Propiedades: nombre string, especie string, peso int.
  - Constructor por defecto (sin propiedades) y un constructor completo (con todas las propiedades)
  - Método ToString() que devuelve un string como por ejemplo:

    “Animal – Tintín – perro - 10”
  - Tenemos que hacer que el objeto se pueda comparar con otros por el nombre y por la especie, para ello debe implementar la interfaz IComparable o utilizar una función lambda en lista.sort(), repasar el tema 7 donde hay ejemplos
- Opciones del menú:
  - Poblar: permite introducir animales a la lista hasta que el usuario introduzca datos vacíos
  - Guardar: guarda la lista que previamente habremos ordenado a un fichero de nombre “animales.json” en una carpeta llamada datos a partir la la carpeta actual, es decir: “./datos/animales.json” si la carpeta no existe la creamos desde el programa. Debemos controlar los posibles errores como que el fichero este abierto o que no tengamos permisos de escritura.
  - Cargar: lee la lista de animales desde el fichero (“./datos/animales.json”) y lo guarda en la lista. Solo se cargarán los animales que no estén duplicados, es decir que su nombre y su especie sea diferente.
  - Listar: Imprime los animales de la lista por la consola.
  - Borrar: Borra el fichero

Cuando arranca el programa si existe el fichero “animales.json” debe cargar los animales a la lista (sin duplicados)

Se debe controlar los errores tanto al leer o escribir ficheros como al leer números.

10.5.3 Con clases derivadas

Para trabajar con un array o una lista con objetos polimórficos es decir con una jerarquía de objetos, basta con aplicar el atributo [JsonDerivedType] en las clases derivadas para indicar cuáles de sus subtipos pueden ser serializados polimórficamente:

```
[JsonDerivedType(typeof(Alumnos))]
public class Alumnos
{
  ...
}
```

El atributo [JsonDerivedType] también puede ser utilizado para generar en el JSON de salida metadatos que permitirán luego realizar la deserialización polimórfica. En la práctica, esto consiste en la inserción de propiedades adicionales en el JSON que ayuden a identificar el tipo específico que fue serializado.

Por ejemplo, volvamos a los tipos Persona , Alumno y Profesor, pero esta vez vamos a insertar dos atributos [JsonDerivedType] para indicar el identificador o discriminador a usar en cada uno de ellos:

```
[JsonDerivedType(typeof(Alumno), typeDiscriminator: nameof(Alumno))]
[JsonDerivedType(typeof(Profesor), typeDiscriminator: nameof(Profesor))]
```

Veamos un ejemplo, primero las clases base Persona, Alumno y Profesor

```
[JsonDerivedType(typeof(Alumno), typeDiscriminator: nameof(Alumno))]
[JsonDerivedType(typeof(Profesor), typeDiscriminator: nameof(Profesor))]

 public  class Persona
 {
     public string Nombre { get; set; }
     public string Apellidos { get; set; }
     public Persona()        {        }
     public Persona(string nombre, string apellidos)     {
         Nombre = nombre;
         Apellidos = apellidos;
     }
     public override string ToString()   {
         return (this.GetType().Name + ";" + this.Nombre + ";" + this.Apellidos);
     }
 }
 public class Alumno : Persona
 {
     public string Curso { get; set; }

     //public Alumno():base() { }
     public Alumno(string nombre, string apellidos, string curso) : base(nombre, apellidos)
{
         this.Curso = curso;
```

```
    }
    public override string ToString()    {
        return (base.ToString() + ";" + this.Curso);
    }
}
public class Profesor : Persona  {
    public string Departamento { get; set; }
    public Profesor():base() { }
    public Profesor(string nombre, string apellidos, string departamento) : base(nombre,
apellidos)   {
        this.Departamento = departamento;
    }
    public override string ToString()    {
        return (base.ToString() + ";" + this.Departamento);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        List<Persona> personas = new List<Persona>();
        personas.Add(new Persona("Pedro", "Domeq"));
        personas.Add(new Profesor("Juan", "Pérez", "Matemáticas"));
        personas.Add(new Alumno("Pedro", "Domeq", "1º DAM"));
        Guardar("personas.json", personas);
        personas = Cargar("personas.json");
        if (personas != null) personas.ForEach(x => Console.WriteLine(x));
    }
    public static void Guardar(string fichero, List<Persona> objetos)
    {
        try {
            string jsonString = JsonSerializer.Serialize<object>(objetos);
            File.WriteAllText(fichero, jsonString);
        }catch (Exception e) { Console.WriteLine(" Error al Guardar() "+e.Message);
                               Console.WriteLine(e.InnerException);}
    }
    public static List<Persona> Cargar(string fichero)
    {
        try
           {
                string jsonString = File.ReadAllText(fichero);
                List<Persona> personas = JsonSerializer.Deserialize<List<Persona>>(jsonString);
                return (personas);
        }catch (Exception e) { Console.WriteLine( " Error al Cargar() "+e.Message);
                        Console.WriteLine(e.InnerException); return (null); }
    }
}
```

10.5.4 Ejercicios propuestos:

(**10.6.4.1)** A partir del ejemplo anterior vamos un programa capaz de serializar/deserializar una lista de Animales controlando los posibles errores con try-catch

- Clase Animal:
  - Propiedades: nombre string, peso int.
  - Constructor por defecto y un constructor completo.
  - Método ToString() que devuelve un string como por ejemplo:

    “Tintín;10”

- Clase Perro, hereda de Animal y añade como atributos propios:
  - Propiedades: raza, string.
  - Constructor por defecto y un constructor completo.
  - Método ToString() que devuelve un string como por ejemplo:

    “Tintín;10;Fox Terrier”

- Clase Gato, , hereda de Animal y añade como atributos propios:
  - propiedades: color, string.
  - Constructor por defecto y un constructor completo.
  - Método ToString() que devuelve un string como por ejemplo:

    “Gatunch;10;amarillo”

- Opciones del menú:
  - Añadir: permite introducir animales a la lista
  - Guardar: guarda la lista que previamente habremos ordenado a un fichero de nombre “animales.json” en una carpeta llamada datos a partir la la carpeta donde están los fuentes en c#, si la carpeta no existe la creamos desde el programa. Debemos controlar los posibles errores como que el fichero este abierto o que no tengamos permisos de escritura.
  - Listar: Imprime los animales de la lista por la consola.
  - Buscar: por nombre, edad, color o raza.
  - Borrar: Borra el fichero

Cuando arranca el programa si existe el fichero “animales.json” debe cargar los animales a la lista