FLUJOS Y ARCHIVOS

PARADIGMAS DE PROGRAMACIÓN II

Carlos Rojas Sánchez

Licenciatura en Informática

Universidad del Mar

Contenido

Conceptos generales

Flujos (streams) y Archivos (files)

En C#, los flujos (streams) y archivos (files) son fundamentales para trabajar con entrada/salida de datos, especialmente para leer, escribir, y manipular archivos en disco.

¿Qué son los Flujos?

- Un **flujo (stream)** es una secuencia de bytes.
- Permiten la transferencia de datos entre fuentes y destinos.
- Tipos:
  - **Flujos de entrada**: lectura.
  - **Flujos de salida**: escritura.

Espacios de nombres

- Para trabajar con archivos y flujos en C#, se usan:

```
using System;
using System.IO;
```

Clases comunes

Clase	Descripción
File	Métodos estáticos para archivos
FileInfo	Información y manipulación de archivos
Stream	Clase base para flujos
FileStream	Lectura/escritura de bytes
StreamReader	Lectura de texto
StreamWriter	Escritura de texto
BinaryReader	Lectura binaria
BinaryWriter	Escritura binaria

Leer un archivo de texto

```
string path = "archivo.txt";
if (File.Exists(path))
{
    string contenido = File.ReadAllText(path);
    Console.WriteLine(contenido);
}
```

Escribir en un archivo de texto

```
string path = "archivo.txt";
File.WriteAllText(path,
"Hola, mundo desde C#!");
```

Leer línea por línea

```
using (StreamReader lector =
new StreamReader("archivo.txt"))
{
 string linea;
 while ((linea = lector.ReadLine()) != null)
 {
  Console.WriteLine(linea);
 }
}
```

Escribir línea por línea

```
using (StreamWriter escritor =
new StreamWriter("archivo.txt"))
{
 escritor.WriteLine("Primera línea");
 escritor.WriteLine("Segunda línea");
}
```

Archivos Binarios

Escribir:

```
using (BinaryWriter bw =
new BinaryWriter(File.Open("datos.bin",
FileMode.Create)))
{
 bw.Write(1234);
 bw.Write(3.14);
 bw.Write("Texto en binario");
}
```

Archivos Binarios

Leer:

```
using (BinaryReader br =
new BinaryReader(File.Open("datos.bin",
FileMode.Open)))
{
 int entero = br.ReadInt32();
 double real = br.ReadDouble();
 string texto = br.ReadString();
}
```

Buenas prácticas

- Usa `using` para liberar recursos automáticamente.
- Verifica si el archivo existe antes de abrirlo.
- Maneja errores con bloques `try-catch`.

Rutas relativas y absolutas

Rutas relativas y absolutas

Ruta absoluta:

- Indica la ubicación completa de un archivo desde la raíz del sistema.
- Ejemplo:

Windows
`C:\Usuarios\Kralos\documentos\archivo.txt`

Linux/macOS
`/home/kralos/documentos/archivo.txt`

Rutas relativas y absolutas

Ruta relativa:

- Relativa al directorio actual de ejecución del programa.
- Ejemplo:

```
datos/archivo.txt
```

Ejemplo de rutas en C#

Obtener ruta absoluta desde relativa

```
string rutaRelativa = "datos/archivo.txt";
string rutaAbsoluta =
Path.GetFullPath(rutaRelativa);
Console.WriteLine(rutaAbsoluta);
```

Ejemplo de rutas en C#

Ruta del directorio actual

```
string actual = Directory.GetCurrentDirectory();
Console.WriteLine(actual);
```

Operaciones básicas en archivos de texto y binarios

Operaciones en archivos de texto

Lectura y escritura comunes:

- `File.WriteAllText(path, texto)`: escribe todo el contenido.
- `File.ReadAllText(path)`: lee todo el contenido.
- `StreamWriter`: para escribir línea por línea.
- `StreamReader`: para leer línea por línea.

Otras operaciones:

- `File.AppendAllText(path, texto)`: agrega texto sin borrar.
- `File.Exists(path)`: verifica si existe el archivo.
- `File.Delete(path)`: elimina el archivo.

Operaciones en archivos binarios

Escribir datos binarios:

- Usar `BinaryWriter` para escribir tipos como `int`, `double`, `string`.
- Método común: `Write(valor)`.

Leer datos binarios:

- Usar `BinaryReader`.
- Leer en el mismo orden en que se escribieron.
- Método común: `ReadInt32()`, `ReadDouble()`, `ReadString()`, etc.

Ejemplo:

```
int x = reader.ReadInt32();
```

Buenas prácticas con archivos

- Siempre usar `using` para cerrar automáticamente el archivo.
- Manejar excepciones con `try-catch`.
- Verificar si el archivo existe antes de abrir.
- Leer y escribir binarios en el mismo orden y tipo.
- No mezclar acceso de texto con binario en el mismo archivo.

Manejo de excepciones

Manejo de excepciones con `try-catch`

¿Por qué usar `try-catch`?

- Para evitar que el programa se detenga por errores en tiempo de ejecución.
- Permite manejar errores como archivos no encontrados, acceso denegado, etc.

Manejo de excepciones con `try-catch`

Ejemplo:

```
try
{
 string texto = File.ReadAllText("archivo.txt");
 Console.WriteLine(texto);
}
catch (FileNotFoundException)
{
 Console.WriteLine("El archivo no existe.");
}
catch (UnauthorizedAccessException)
{
 Console.WriteLine("Acceso denegado al archivo.");
}
catch (Exception ex)
{
 Console.WriteLine($"Error inesperado: {ex.Message}");
}
```

Excepciones comunes en archivos

Principales excepciones que pueden ocurrir:

- `FileNotFoundException`
  - Ocurre si el archivo no existe.
- `DirectoryNotFoundException`
  - El directorio especificado no existe.
- `UnauthorizedAccessException`
  - No hay permisos para acceder al archivo o directorio.
- `IOException`
  - Error de entrada/salida general (archivo en uso, disco lleno, etc.).
- `PathTooLongException`
  - La ruta o el nombre del archivo es demasiado largo.

Flujos

¿Qué es un flujo (Stream)?

- Es una secuencia de bytes para leer o escribir datos.
- Facilita la comunicación entre archivos, memoria y dispositivos.
- Puede ser de entrada (lectura) o de salida (escritura).

Tipos de flujos en C#

- **FileStream:** lectura/escritura de archivos binarios.
- **StreamReader / StreamWriter:** manejo de texto.
- **MemoryStream:** flujo en memoria RAM.
- **BufferedStream:** mejora el rendimiento mediante buffering.
- **NetworkStream:** comunicación sobre redes.

Características de los flujos

- **Unidireccionales:** lectura o escritura, pero no ambas (la mayoría).
- **Secuenciales:** acceso de datos en orden.
- **Abstracción:** ocultan detalles del dispositivo subyacente.
- **Jerarquía:** basados en la clase abstracta `Stream`.

Encadenamiento de flujos

- Se pueden encadenar varios flujos para procesar datos.
- Ejemplo: un `FileStream` puede ser envuelto por un `BufferedStream` y luego por un `StreamReader`.
- Esto permite aplicar distintas funciones en capas (buffering, lectura, compresión, etc.).

Ejemplo típico

```
using var sr = new StreamReader(
new BufferedStream(
new FileStream("archivo.txt", FileMode.Open)));
```

Usos comunes de los flujos

- Leer y escribir archivos de texto o binarios.
- Transferir datos entre red y aplicación.
- Manipular datos en memoria temporalmente.
- Serialización de objetos.
- Procesamiento de archivos grandes sin cargarlos completamente en memoria.

Operaciones Básicas con Archivos XML en C#

¿Qué es un archivo XML?

- XML (eXtensible Markup Language) es un formato de texto estructurado.
- Se usa para almacenar y transportar datos jerárquicos.
- En C#, se puede manipular usando:
  - `XmlDocument` (DOM)
  - `XmlReader` / `XmlWriter` (flujo)
  - `XDocument` (LINQ to XML)

Leer un archivo XML

Ejemplo: leer el valor de un nodo

```
XmlDocument doc = new XmlDocument();
doc.Load("usuarios.xml");

XmlNode nodo = doc.SelectSingleNode("/usuarios/usuario");
if (nodo != null)
{
    Console.WriteLine(nodo.InnerText);
}
```

Crear un archivo XML

Ejemplo: crear un archivo con un nodo raíz y un nodo hijo

```
XmlDocument doc = new XmlDocument();
XmlElement raiz = doc.CreateElement("usuarios");
doc.AppendChild(raiz);

XmlElement usuario = doc.CreateElement("usuario");
usuario.InnerText = "Carlos";
raiz.AppendChild(usuario);

doc.Save("usuarios.xml");
```

Modificar un nodo XML

Ejemplo: cambiar el valor de un nodo existente

```
XmlDocument doc = new XmlDocument();
doc.Load("usuarios.xml");

XmlNode nodo = doc.SelectSingleNode("/usuarios/usuario");
if (nodo != null)
{
    nodo.InnerText = "Ana";
    doc.Save("usuarios.xml");
}
```

Eliminar un nodo XML

Ejemplo: eliminar un nodo del documento

```
XmlDocument doc = new XmlDocument();
doc.Load("usuarios.xml");

XmlNode nodo = doc.SelectSingleNode("/usuarios/usuario");
if (nodo != null && nodo.ParentNode != null)
{
    nodo.ParentNode.RemoveChild(nodo);
    doc.Save("usuarios.xml");
}
```

Guardar un archivo XML

- Luego de realizar cambios con `XmlDocument`, debes guardar:

```
doc.Save("archivo.xml");
```

- Esto escribe todos los cambios en disco.
- También puedes usar un `Stream` o una `TextWriter`.

Buenas prácticas al usar XML

- Validar si el archivo existe antes de cargarlo.
- Usar bloques `try-catch` para manejar excepciones.
- Usar `XPath` correctamente en `SelectSingleNode()`.
- Guardar siempre después de modificar.
- Evitar sobrescribir datos sin confirmar.

Excepciones comunes con archivos XML

Al trabajar con XML en C#, pueden ocurrir:

- `FileNotFoundException`
  El archivo XML no existe.
- `DirectoryNotFoundException`
  El directorio del archivo no existe.
- `UnauthorizedAccessException`
  Acceso denegado al archivo o ruta.
- `IOException`
  Error general de entrada/salida.

Excepciones comunes con archivos XML

Al trabajar con XML en C#, pueden ocurrir:

- `XmlException`
  El XML está mal formado o tiene errores de sintaxis.
- `InvalidOperationException`
  El nodo seleccionado no existe o es inválido.
- `ArgumentNullException`
  Se pasó un valor nulo cuando no debía.

Manejo de excepciones en archivos XML

Ejemplo de uso de `try-catch` para cargar un archivo XML:

```
try {
 XmlDocument doc = new XmlDocument();
 doc.Load("usuarios.xml");
 Console.WriteLine("Archivo cargado correctamente.");
} catch (FileNotFoundException) {
 Console.WriteLine("El archivo no fue encontrado.");
} catch (XmlException ex) {
 Console.WriteLine("Error de formato XML: " + ex.Message);
} catch (IOException ex) {
 Console.WriteLine("Error de E/S: " + ex.Message);
} catch (Exception ex) {
 Console.WriteLine("Error inesperado: " + ex.Message);
}
```

Operaciones Básicas con Archivos JSON en C#

¿Qué es JSON?

- JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos.
- Muy usado para comunicar APIs y almacenar configuraciones.
- En C#, se trabaja con la librería `System.Text.Json`.

Definición de clase de ejemplo

Antes de serializar o deserializar, define tus clases:

```
public class Usuario
{
   public string Nombre { get; set; }
   public int Edad { get; set; }
}
```

Leer (deserializar) desde archivo JSON

Convierte JSON a un objeto C#:

```
string json = File.ReadAllText("usuario.json");
Usuario usuario = JsonSerializer.Deserialize<Usuario>(json);
Console.WriteLine(usuario.Nombre);
```

Escribir (serializar) a archivo JSON

Convierte un objeto C# a JSON y lo guarda:

```
Usuario usuario = new Usuario { Nombre = "Ana", Edad = 30 };
string json = JsonSerializer.Serialize(usuario);
File.WriteAllText("usuario.json", json);
```

Leer una lista de objetos JSON

Deserializar un arreglo de objetos:

```
string json = File.ReadAllText("usuarios.json");
List<Usuario> usuarios =
   JsonSerializer.Deserialize<List<Usuario>>(json);
foreach (var u in usuarios)
Console.WriteLine(u.Nombre);
```

Serializar una lista de objetos JSON

Guardar varios objetos en un archivo JSON:

```
List<Usuario> usuarios = new List<Usuario>
{
    new Usuario { Nombre = "Carlos", Edad = 25 },
    new Usuario { Nombre = "Luisa", Edad = 28 }
};

string json = JsonSerializer.Serialize(usuarios);
File.WriteAllText("usuarios.json", json);
```

Buenas prácticas

- Validar si el archivo existe antes de leerlo.
- Manejar excepciones: `FileNotFoundException`, `JsonException`, etc.
- Usar nombres de propiedades en C# que coincidan con el JSON.
- Configurar la serialización si necesitas camelCase o formatos específicos.

Manejo de errores con JSON

Lectura segura desde un archivo JSON usando `try-catch`:

```
try {
  string json = File.ReadAllText("usuario.json");
  Usuario usuario =
   JsonSerializer.Deserialize<Usuario>(json);
  Console.WriteLine(usuario.Nombre);
} catch (FileNotFoundException) {
  Console.WriteLine("El archivo no fue encontrado.");
} catch (JsonException ex) {
  Console.WriteLine("El contenido JSON es inválido: "
  + ex.Message);
} catch (IOException ex) {
  Console.WriteLine("Error de lectura: " + ex.Message);
} catch (Exception ex) {
  Console.WriteLine("Error inesperado: " + ex.Message);
}
```

Excepciones comunes en archivos JSON

Al trabajar con archivos JSON pueden surgir:

- `FileNotFoundException`: El archivo no existe.
- `JsonException`: El contenido del archivo no tiene formato JSON válido.
- `IOException`: Fallo de lectura o escritura (permiso, bloqueo).
- `UnauthorizedAccessException`: No tienes permiso para acceder al archivo.
- `Exception`: Otros errores no previstos.

Proyectos de Clase
Flujos y Archivos en C#

Propósito de los proyectos

- Aplicar conceptos de archivos y flujos en C#
- Desarrollar habilidades prácticas en lectura y escritura de datos
- Enfrentar diferentes formatos: texto, binario, XML, JSON y red
- Preparar para escenarios reales con buenas prácticas

Proyecto 1: Editor de notas personales

Tipo: Archivos de texto
Temas: File, StreamWriter, StreamReader

Funciones:

- Crear, guardar y leer notas en archivos .txt
- Borrar notas
- Organización por carpetas (importantes, tareas, etc.)

Extras: rutas absolutas y relativas, excepciones

Proyecto 2: Gestor de clientes con JSON

Tipo: Archivos JSON
Temas: System.Text.Json, serialización

Funciones:

- Registrar y mostrar clientes desde un archivo JSON
- Editar y eliminar clientes
- Guardado automático de cambios

Extras: validación de entrada, interfaz simple

Proyecto 3: Inventario binario de productos

Tipo: Archivos binarios
Temas: BinaryWriter, BinaryReader

Funciones:

- Añadir productos con código, nombre, precio y cantidad
- Consultar productos desde un archivo binario
- Buscar por código

Extras: consola interactiva, menú de opciones

Proyecto 4: Chat básico con NetworkStream

Tipo: Flujos de red (Sockets)
Temas: TcpClient, TcpListener, NetworkStream

Funciones:

- Comunicación entre cliente y servidor en red local
- Envío de mensajes de texto en tiempo real

Extras: uso de hilos para conexión simultánea

Proyecto 5: Agenda en XML

Tipo: Archivos XML
Temas: XmlDocument, nodos, atributos

Funciones:

- Guardar contactos (nombre, teléfono, correo)
- Buscar, editar y eliminar desde un archivo XML

Extras: estructura jerárquica y validación básica

Resumen de Proyectos

Proyecto	Tipo	Tema Principal
Notas personales	Texto	Lectura/Escritura básica
Clientes	JSON	Serialización
Inventario	Binario	Lectura estructurada
Chat	NetworkStream	Comunicación en red
Agenda	XML	Jerarquía de nodos

Requisitos

- Implementar pruebas con datos reales
- Agregar manejo de errores con `try-catch`
- Documentar el proceso con comentarios y diagramas