Capítulo 10

Archivos, Flujos y Persistencia de Objetos

Archivos y Flujos

La unidad mínima de representación de un valor en una computadora es el bit. La unidad mínima de procesamiento de datos es el byte. Uno o más bytes pueden representar un carácter (un dígito, una letra o un símbolo) o un número (integral o de punto flotante). Ésta es la plataforma base sobre la cual los lenguajes de programación dan un valor agregado a las capacidades brindadas al programador para manejar estructuras de datos. Una descripción clásica de la forma de organización de la información por los programas es:

"Un conjunto de bytes con significado agregado y que se operan como una unidad se le denomina campo. Un conjunto de campos con significado agregado y que se operan como una unidad se denomina registro. Un conjunto de registros pueden ser almacenados de forma persistente en un archivo. Un conjunto de archivos de registros forman una base de datos."

El concepto de registro está estrechamente ligado al de estructura de un archivo: El registro es una estructura de datos que representa la unidad de lectura y escritura sobre un archivo. Algunos lenguajes de programación requieren la especificación del tipo de registro al manipular un archivo (por ejemplo Pascal), mientras que otros no (por ejemplo C, Java y C#), visualizando a los archivos como un conjunto de bytes a los que el programa da significado durante la misma lectura.

Los datos almacenados en memoria volátil (el caso de la RAM) tienen un ciclo de vida limitado al del programa que los crea. Estos se denominan datos temporales. Los datos almacenados en memoria no-volátil (el caso del disco duro) tienen un ciclo de vida independiente al de la instancia del programa que los crea o manipula. Estos datos se denominan datos persistentes. En el presente contexto, el término memoria refiere a cualquier dispositivo físico de almacenamiento o parte de él.

Existen dos formas de acceder a los datos persistentes: de forma **secuencial** y **aleatoria**. Dicho acceso puede tener un tipo de permiso asignado: Sólo lectura, sólo escritura o lectura-escritura. Si es factible que más de un programa acceda a dichos datos, pueden establecerse permisos para compartir dichos datos. Las formas y permisos dependen de las capacidades de la memoria que almacena dichos datos.

En general, el acceso a un dato involucra el traspaso de éste de una memoria a otra. Esta transferencia puede hacerse físicamente por bloques de bytes o de byte en byte. Esta transferencia puede involucrar una sola operación de copia (todos los datos transferidos están disponibles a la vez) o varias operaciones secuenciales. En este último caso, los datos están disponibles conforme van llegando. Es aquí donde el concepto de flujo es útil.

Un **flujo** es una secuencia de bytes que son accedidos en el mismo orden en que fueron creados. Un flujo opera de manera similar a una cola: Los datos son leídos desde un flujo en el mismo orden en que fueron escritos en él. Dado que cualquier memoria permite por lo menos un acceso secuencial a sus datos, ya sea para lectura o

escritura, ya sea que los datos estén disponibles a la vez o secuencialmente, éstas pueden trabajarse siempre como flujos. Si bien por definición un flujo permite un acceso secuencial a los datos de la memoria que maneja, también puede permitir un acceso aleatorio, dependiendo del tipo de memoria.

Objetos Persistentes

En la POO, el concepto de objeto reemplaza sobremanera al de registro. Un objeto que es almacenado en memoria persistente se le conoce como **objeto persistente**. Una forma de lograr esta persistencia es mediante una técnica llamada **serialización**.

Se dice que un objeto es serializado cuando es escrito hacia un flujo de una forma tal que pueda ser leído luego y reconstruido. El proceso de lectura y reconstrucción de un objeto se conoce como **deserialización**. La serialización implica mucho más que sólo escribir los datos de un objeto en un orden y leerlos en el mismo orden. Existen dos problemas principales en la serialización de un objeto:

- Determinar quién es responsable de serializar y deserealizar un objeto: El mismo objeto o el mecanismo de serialización, ya sea éste implementado a nivel del lenguaje de programación o con una librería.

- Cómo manejar los diagramas de clases.

Encargar la responsabilidad de serializar/deserealizar un objeto al mismo objeto tiene la ventaja de darle al programador de la clase de dicho objeto el control completo del proceso. La clara contraparte es el mayor trabajo de programación requerido y por consiguiente, el aumento de la tasa de error. Encargar la responsabilidad al mecanismo de serialización tiene la ventaja de simplificar la programación, pero la desventaja de requerirse un mecanismo de reflexión de apoyo, dado que el mecanismo de serialización deberá poder reconocer en tiempo de ejecución a qué tipo de objeto corresponde los datos leídos y crearlo, todo de manera automática.

Los diagramas de clases se forman cuando los datos miembros de un objeto refieren a otros objetos, formando un diagrama de conexiones entre objetos. Cuando uno de los objetos de este diagrama debe serializarse, también deberá serializarse todos los objetos a los que refiere, directa o indirectamente. Esto no sólo implica un rastreo recursivo de todas las referencias del objeto serializado, sino además la resolución de conflictos tales como: ¿Qué sucede cuando al rastrear dicho diagrama durante la serialización se llega a un mismo objeto más de una vez?, ¿Cómo reconocer que ya se serializó un objeto previamente en el mismo flujo? Para el manejo de los diagramas de objetos, como es claro, el delegar el trabajo de serialización al programador puede aumentar enormemente la complejidad del código final.

Manejo desde C#

Descripción General de las Capacidades

C# permite el manejo de archivos de texto y binarios, secuenciales y aleatorios. Dichos archivos son manejados como flujos. Cuando un archivo es abierto, C# crea un objeto y lo relaciona con el flujo de dicho archivo.

Existen tres flujos estándar con objetos relacionados que son creados automáticamente para todo programa en C#:

- El flujo de entrada estándar, mediante la referencia System.Console.In.

- El flujo de salida estándar, mediante la referencia System.Console.Out.

- El flujo de error estándar, mediante la referencia System.Console.Error.

Los métodos de la clase Console de lectura tales como Read y ReadLine, y de escritura tales como Write y WriteLine hacen uso de Console.In y Console.Out respectivamente.

Todas las clases para el manejo de las operaciones de entrada y salida se encuentran en el espacio de nombres System.IO. Algunas de las más usuales se muestran en la figura 10.1

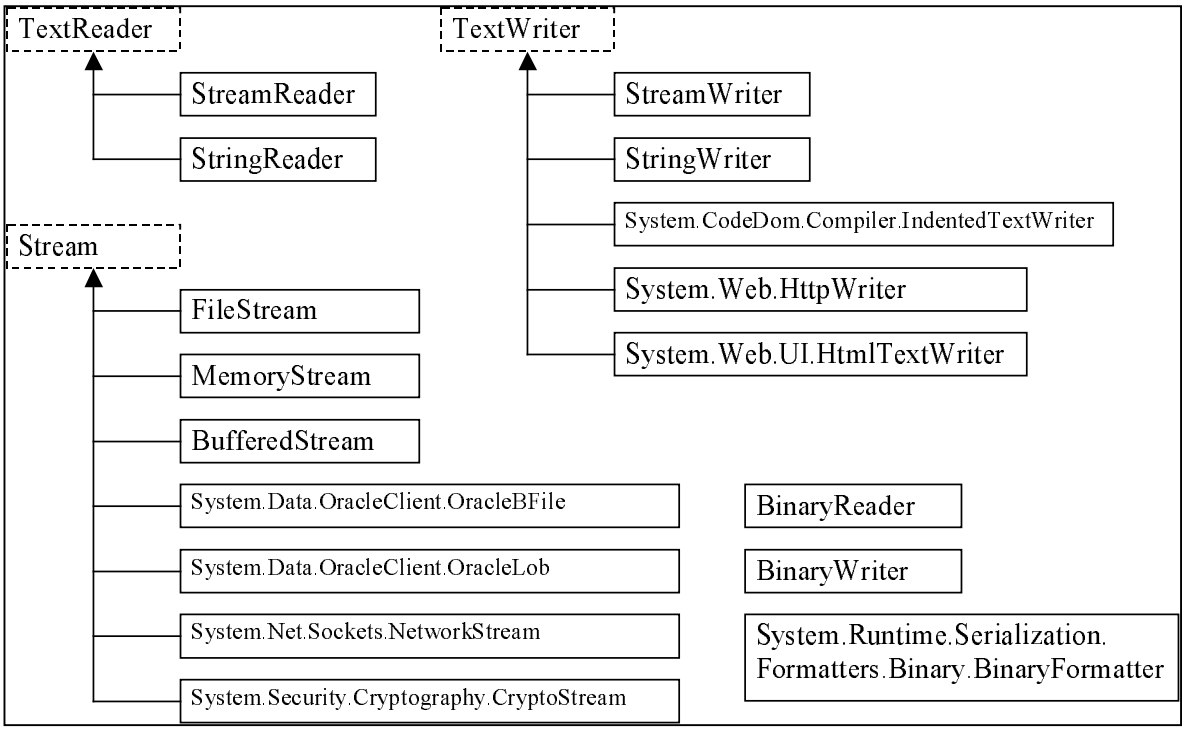

Figura 10.1. Las clases para manejo de flujos en C#

Las clases TextReader, TextWriter y Stream son clases abstractas para la lectura y escritura de archivos de texto y binarios. Los objetos referenciados por Console.In y Console.Out son de tipo TextReader y TextWriter respectivamente

Las clases StreamReader y StringReader permiten manejar como flujos de salida archivos de texto y cadenas de carácteres respectivamente. Su contraparte para flujos de entrada son StreamWriter y StringWriter.

Las clases FileStream, MemoryStream y BufferedStream son para manejo de flujos binarios, para lectura y escritura. A dichos flujos se pueden serializar objetos, para lo que se utiliza la clase BinaryFormatter. Esta clase puede serializar y deserializar cualquier objeto, junto con su diagrama de objetos, marcado como serializable. Dicha marca se realiza mediante el atributo estándar serializable, como se verá más adelante.

Finalmente, C# permite el acceso al sistema de archivos del sistema operativo, mediante las clases File y Directory, como se verá en la siguiente sección.

Acceso al Sistema de Archivos

Para el manejo del sistema de archivos se provee de dos clases:

- File, para manipular archivos

- Directory, para manipular directorios

Ambas clases proveen métodos estáticos para manejar archivos y directorios. Las tabla 10.1 muestra los métodos de File más comunes.

Tabla 10.1. Métodos de la clase File de .NET

Manipulación de archivos	
Nombre	**Descripción**
Copy	Copia un archivo a un nuevo archivo.
Move	Mueve un archivo.
Delete	Elimina un archivo.
Información de un archivo	
Nombre	**Descripción**
Exists	Averigua si un archivo existe.
GetCreationTime	Retorna la fecha en que se creó el archivo.
GetLastAccessTime	Retorna la fecha en que se accedió por última vez al archivo.
GetLastWriteTime	Retorna la fecha en que se escribió por última vez en el archivo.
Archivos de texto	
Nombre	**Descripción**
CreateText	Crea un archivo de texto y retorna un objeto StreamWriter asociado.
OpenText	Apertura un archivo de texto y retorna un objeto StreamReader asociado.
AppendedText	Retorna un objeto StreamWriter para agregar datos a un archivo existente o crea uno nuevo.
Archivos binarios	
Nombre	**Descripción**
Create	Crea un archivo y retorna un objeto FileStream asociado.
Open	Abre un archivo, bajo distintos modos, y retorna un objeto FileStream asociado.
OpenRead	Abre un archivo existente para lectura.
OpenWrite	Abre un archivo existente para escritura.

Las tabla 10.2 muestra los métodos de Directory más comunes.

Tabla 10.2. Métodos de la clase Directory de ..NET

Manipulación de directorios	
Nombre	**Descripción**
CreateDirectory	Crea un nuevo directorio.
Move	Mueve un directorio.
Delete	Elimina un directorio.
Información de un directorio	
Nombre	**Descripción**
Exists	Averigua si un directorio existe.
GetCreationTime	Retorna la fecha en que se creó el directorio.
GetLastAccessTime	Retorna la fecha en que se accedió por última vez al directorio.
GetLastWriteTime	Retorna la fecha en que se escribió por última vez en el directorio.

Manejo de Consola

La entrada, salida y error estándar se manejan mediante las propiedades estáticas públicas de sólo lectura In, Out y Error de la clase Console, las que retornan referencias de tipo TextReader, TextWriter y TextWriter respectivamente. Luego, las siguientes sentencias son equivalentes:

```
Console.Write(dato);       equivale a    Console.Out.Write(dato);
Console.WriteLine(dato);   equivale a    Console.Out.WriteLine(dato);
dato = Console.Read();     equivale a    dato = Console.In.Read();
dato = Console.ReadLine(); equivale a    dato = Console.In.ReadLine();
```

La clase abstracta TextReader permite la lectura de carácteres, uno a uno o por líneas, desde un flujo de texto. La clase abstracta TextWriter permite la escritura de carácteres, uno a uno o por líneas, hacia un flujo de texto. Los objetos referidos por las propiedades de Console realmente son de un tipo concreto que hereda de estas clases. Esto puede comprobarse fácilmente con el siguiente código:

```
Type tipo = Console.In.GetType();
Type tipoBase = tipo.BaseType;
Console.WriteLine("Console.In es de tipo = " + tipo.Name);
Console.WriteLine("El que hereda del tipo = " + tipoBase.Name);
```

El nombre del tipo que se muestra en la ventana de consola es SyncTextReader, el cual es un tipo interno (no público) del espacio de nombres System.IO.

El siguiente programa es un ejemplo del uso de la consola leyendo carácter por carácter.

```
using System;

class Consola1 {
    public static void Main(string[] args) {
        Console.Write( "Ingrese el texto a Leer." );
        Console.WriteLine( " Finalizar con la combinacion CTRL+Z." );
        while(true) {
            int Entero = Console.In.Read();
            if( Entero == -1 )
                break;
            char Caracter = (char)Entero;
            Console.Write( "Leido Entero=" + Entero );
            Console.WriteLine( ", correspondiente al carácter=" + Caracter );
        }
        Console.WriteLine( "Fin del ingreso" );
    }
}
```

La figura 10.2 muestra el resultado de una ejecución de este programa.

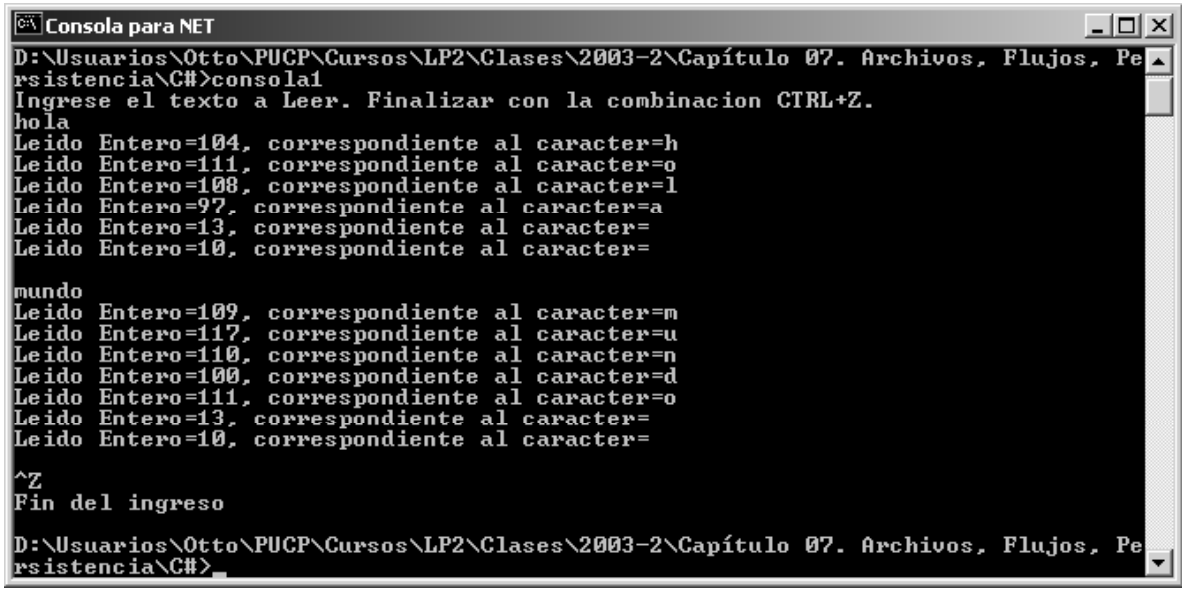

Figura 10.2. Manejo de consola en C#: Ejecución del programa 1

El siguiente programa es un ejemplo del uso de la consola leyendo línea por línea. Note que cuando, al inicio de la línea, el usuario ingresa CTRL+Z, el método ReadLine reconoce esto como un indicador de fin de ingreso, retornando null.

```
using System;

class Consola2 {
    public static void Main(string[] args) {
        Console.WriteLine( "Ingrese una secuencia de enteros." );
        Console.WriteLine( "Finalice el ingreso con '*'." );
        int Contador = 0;
        int Suma = 0;
        while(true) {
            string Linea = Console.In.ReadLine();
            if( Linea == null )
                break;
            if( Linea[0] == '*' )
                break;
            Suma += Int32.Parse(Linea);
            Contador++;
        }
        Console.WriteLine( "Se ingresaron " + Contador + " enteros." );
        Console.WriteLine( "El promedio es " + (Suma / Contador) );
    }
}
```

La figura 10.3 muestra el resultado de una ejecución de este programa.

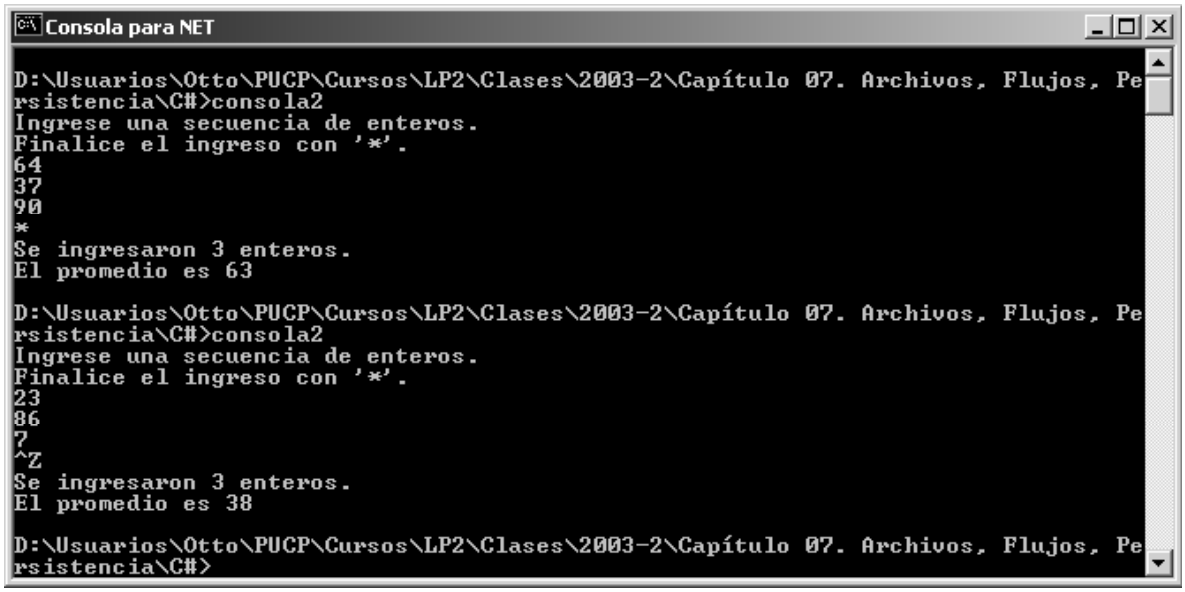

Figura 10.3. Manejo de consola en C#: Ejecución del programa 2

El siguiente programa es un ejemplo de la interpretación de la lectura, cuando se ingresa más de un dato a la vez, en cuyo caso se puede partir dicha lectura mediante el método **Split** de la clase string, pasándole como parámetro un arreglo de carácteres que deberán ser utilizados como delimitadores.

```
using System;

class Consola3 {
    public static void Main(string[] args) {
        Console.WriteLine( "Ingrese una secuencia de palabras." );
        string Linea = Console.In.ReadLine();
        char[] separadores = {' ', ',', '.', ':'};
        string[] Tokens = Linea.Split( separadores );
        for(int i = 0; i < Tokens.Length; i++)
            Console.WriteLine("Token " + (i + 1) + " = " + Tokens[i]);
    }
}
```

La figura 10.4 muestra una corrida de este programa.

Figura 10.4. Manejo de consola en C#: Ejecución del programa 3

Manejo de Archivos de Texto

Los archivos de texto se manejan de manera similar a la consola. El siguiente ejemplo muestra la creación y lectura de un archivo de texto.

```
using System;
using System.IO;

class ArchivoTexto {
    public static void Main( string[] args ) {
        FileStream Archivo;
Archivo = new FileStream( args[0], FileMode.Create, FileAccess.Write );
        StreamWriter Escritor = new StreamWriter( Archivo );

        Console.WriteLine( "Ingrese el texto a almacenar: " );
        while( true ) {
            string Linea = Console.ReadLine();
            if( Linea == null )
                break;
            Escritor.WriteLine( Linea );
        }
        Escritor.Close();
        Archivo.Close();

        Console.WriteLine( "Leyendo el archivo creado." );
        Archivo = new FileStream( args[0], FileMode.Open, FileAccess.Read );
        StreamReader Lector = new StreamReader( Archivo );
        while( true ) {
            string Linea = Lector.ReadLine();
            if( Linea == null )
                break;
            Console.WriteLine( Linea );
        }
        Lector.Close();
        Archivo.Close();
    }
}
```

Para crear un archivo se crea un flujo desde un archivo utilizando la clase FileStream. Esta clase provee métodos para leer y escribir únicamente bytes, sin ninguna interpretación. Para escribir o leer los datos de este flujo como texto, se crea otro flujo utilizando la clase StringWriter y StringReader respectivamente. Ambas clases ofrecen además un constructor que recibe directamente el nombre del archivo de texto, de forma que no sea necesario crear el objeto FileStream manualmente.

Manejo de Archivos Binarios Secuencialmente

Los archivos binarios secuenciales son comúnmente manejados mediante la técnica de serialización. En .NET, la serialización se realiza utilizando una clase utilitaria llamada formateador (que tiene la misma función que el formateador en .NET Remoting) que implemente los métodos Serialize y Deserialize. El único formateador de este tipo en la versión actual de la librería de .NET es **BinaryFormatter**.

La clase BinaryFormatter se basa en un atributo para determinar si los objetos de una clase deben ser serializados: Serializable. Un atributo en .NET es una metadata relacionada a un elemento de programación, como la definición de un tipo de dato, la declaración de un método o de un dato miembro, etc. El siguiente ejemplo muestra el uso del atributo Serializable y de la clase BinaryFormatter.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

[Serializable]
class Direccion {
    int numero;
    string calle;
    string distrito;
    public Direccion(int numero, string calle, string distrito)       {
        this.numero = numero;
        this.calle = calle;
        this.distrito = distrito;
    }
    public override string ToString() {
        return calle + " " + numero + ", " + distrito;
    }
}

[Serializable]
class Persona {
    private string nombre;
    private int edad;
    private Direccion dir;
    public Persona(string nombre, int edad, Direccion dir) {
        this.nombre = nombre;
        this.edad = edad;
        this.dir = dir;
    }
    public override string ToString() {
        return "Sr(a). " + nombre + ", " + edad + " años, direccion = " + dir;
    }
}


class ArchivoBinarioSecuencial {
    public static void Main(string[] args) {
        if(args.Length < 2) {
            Console.WriteLine("Error en argumentos.");
            return;
        }
        if(args[0] == "/e") {
            FileStream Output = new FileStream(args[1], FileMode.Create, FileAccess.Write);
            BinaryFormatter Formateador = new BinaryFormatter();
            Persona[] ListaPersonas = {
                new Persona("Jose", 25, new Direccion(123, "Jose Leal", "San Juan")),
                new Persona("Mara", 26, new Direccion(456, "Carrión", "Barranco")),
                new Persona("Ana", 35, new Direccion(789, "Fresnos", "Lince"))
            };
            foreach(Persona p in ListaPersonas)
                Formateador.Serialize(Output, p);
            Output.Close();
        } else if(args[0] == "/l") {
            FileStream Input = new FileStream(args[1], FileMode.Open, FileAccess.Read);
            BinaryFormatter Formateador = new BinaryFormatter();
            while(true) {
```

```
                Persona p;
                try { p = (Persona)Formateador.Deserialize(Input); }
                catch(SerializationException){ break; }
                Console.WriteLine("Persona : " + p);
            }
            Input.Close();
        }
    }
}
```

El programa utiliza los argumentos de la línea de comandos para determinar si debe crear un archivo o si debe de leerlo.

Al crear el archivo se inicializa un arreglo de objetos Persona, los que serializan utilizando el método Serialize de la clase BinaryFormatter. Tome en cuenta que cuando se serializa un objeto, se serializa también todos los objetos a los que hace referencia directa o indirectamente. Si se intenta serializar un objeto no marcado con el atributo Serializable, el método Serialize dispara una excepción.

En forma similar al leer el archivo, se utiliza el método Deserialize, el cual retorna una referencia de tipo object a la que debe aplicársele una operación cast para obtener una referencia al tipo real. Hay dos cosas que pueden fallar durante una deserialización: Que se haya llegado al final del archivo sin haberse podido leer todos los datos de un objeto o bien que la operación de cast falle, es decir, se leyó un objeto diferente al que se esperaba. En ambos casos se produce una excepción.

Es importante recordar que la serialización no requiere que el flujo reciba objetos del mismo tipo. Se puede serializar a un mismo flujo objetos de distintos tipos, en cuyo caso deberá asegurarse que el proceso de deserilización sea realizado en el mismo orden que el proceso de serialización sobre el flujo tratado.

Manejo de Archivos Binarios Aleatoriamente

.NET no fuerza un formato sobre los archivos binarios, por lo que tampoco provee de mecanismos para acceder aleatoriamente a los objetos serializados hacia un flujo de la misma forma como lo hacen otros lenguajes como Pascal. Es importante tomar en cuenta que al serializar dos objetos del mismo tipo no necesariamente se escriben el mismo número de bytes en el flujo. Un ejemplo claro es el de los objetos string. Si serializamos dos objetos de este tipo podrán ocupar espacios de diferente tamaño dentro del flujo. Debido a esto, tampoco es sencillo colocar el apuntador del archivo al inicio del n-ésimo objeto y leerlo. Si deseamos hacer esto, es necesario garantizar que durante la serialización se escriba por cada objeto siempre la misma cantidad de bytes, o implementar manualmente algún otro mecanismo para realizar un acceso aleatorio.

El siguiente programa serializa manualmente objetos de la clase Combo a un archivo binario, garantizando que los datos miembros de cada objeto siempre ocupen el mismo espacio, de manera que nos podamos mover por el archivo y leer directamente cualquiera de los objetos del flujo.

```
using System;
using System.IO;
using System.Runtime.Serialization;

class Combo {
    public const int LONG_NOMBRE = 10;
    public const int LONG_DESCRIPCION = 30;
    public const int LONG_CARACTER = 1; // longitud de un char escrito en un archivo TXT
    public const int LONG_REGISTRO =
        (LONG_NOMBRE + LONG_DESCRIPCION) * LONG_CARACTER + sizeof(double);

    private char[] Nombre = new char[LONG_NOMBRE];
    private char[] Descripcion = new char[LONG_DESCRIPCION];
    private double Precio;

    public Combo(string Nombre, string Descripcion, double Precio) {
```

```
            Copiar(this.Nombre, Nombre);
            Copiar(this.Descripcion, Descripcion);
            this.Precio = Precio;
        }
        private static void Copiar(char[] Destino, string Origen) {
            if(Origen.Length < Destino.Length) {
                Origen = Origen.PadRight(Destino.Length);
                Origen.CopyTo(0, Destino, 0, Destino.Length);
            } else
                Origen.CopyTo(0, Destino, 0, Destino.Length);

        }
        public override string ToString() {
            string Nombre = new string(this.Nombre);
            string Descripcion = new string(this.Descripcion);
            return "Combo " + Nombre + ": " + Descripcion + ", a S/. " + Precio;
        }
        public void SetPrecio(double Precio) {
            this.Precio = Precio;
        }
        public void Serialize(BinaryWriter Output) {
            Output.Write(Nombre);
            Output.Write(Descripcion);
            Output.Write(Precio);
        }
        public static Combo Deserialize(BinaryReader Input) {
            char[] Nombre = Input.ReadChars(LONG_NOMBRE);
            char[] Descripcion = Input.ReadChars(LONG_DESCRIPCION);
            double Precio = Input.ReadDouble();
            return new Combo(new string(Nombre), new string(Descripcion), Precio);
        }
    }

    class ArchvoBinarioAleatorio {
        public static void Main(string[] args) {
            if(args.Length != 1) {
                Console.WriteLine("Error en parametros.");
                return;
            }

            // Creo el archivo de salida
            FileStream Flujo = new FileStream(args[0], FileMode.Create, FileAccess.Write);
            BinaryWriter Salida = new BinaryWriter(Flujo);
            Combo[] Combos = {
                new Combo("Tradicional", "Bembos trad. medium + papas chicas +
                    gaseosa chica", 10.9),
                new Combo("Peruano", "Bembos peruana tradicional + papas chicas +
                    helado", 12.9),
                new Combo("Frances", "Bembos francesa medium + ensalada + copa de " +
                    "vino", 15.9)
            };
            foreach(Combo C in Combos)
                C.Serialize(Salida);
            Console.WriteLine("Tamaño del archivo creado = " + Flujo.Length);
            Salida.Close();
            Flujo.Close();

            // Leo el archivo creado
            Flujo = new FileStream(args[0], FileMode.Open, FileAccess.Read);
            BinaryReader Entrada = new BinaryReader(Flujo);
            while(true) {
                try {
                    Combo C = Combo.Deserialize(Entrada);
                    Console.WriteLine("Leido = " + C);
                }
                catch(EndOfStreamException) {
                    break;
                }
            }
            Entrada.Close();
            Flujo.Close();

            // Abro nuevamente para lectura/escritura
```

```
            Flujo = new FileStream(args[0], FileMode.Open, FileAccess.ReadWrite);
            Entrada = new BinaryReader(Flujo);
            Salida = new BinaryWriter(Flujo);
            bool Fin = false;
            while(!Fin) {
                Console.WriteLine("Seleccione una opcion:");
                Console.WriteLine("1. Leer un registro");
                Console.WriteLine("2. Modificar un Registro");
                Console.WriteLine("3. Terminar");
                try {
                    int Opcion;
                    Opcion = Int32.Parse(Console.In.ReadLine());
                    switch(Opcion) {
                        case 1:
                            Opcion1(Flujo, Entrada);
                            break;
                        case 2:
                            Opcion2(Flujo, Entrada, Salida);
                            break;
                        case 3:
                            Fin = true;
                            break;
                        default:
                            Console.WriteLine("Opción inválida.");
                            break;
                    }
                }
                catch(FormatException) {
                    Console.WriteLine("Debe ingresar un número.");
                    continue;
                }
            }
            Entrada.Close();
            Salida.Close();
            Flujo.Close();

        }
        public static void Opcion1(Stream Flujo, BinaryReader Entrada) {
            Console.WriteLine("Número de registro a leer:");
            int Registro = Int32.Parse(Console.In.ReadLine());
            int TotalRegistros = (int)Flujo.Length / Combo.LONG_REGISTRO;
            if(Registro >= TotalRegistros)
                Console.WriteLine("Número de registro inválido.");
            else {
                Flujo.Position = Registro * Combo.LONG_REGISTRO;
                Console.WriteLine("Puntero del archivo en la pos:" + Flujo.Position);
                Combo C = Combo.Deserialize(Entrada);
                Console.WriteLine("Leido = " + C);
            }
        }
        public static void Opcion2(Stream Flujo, BinaryReader Entrada, BinaryWriter Salida){
            Console.WriteLine("Número de registro a leer:");
            int Registro = Int32.Parse(Console.In.ReadLine());
            int TotalRegistros = (int)Flujo.Length / Combo.LONG_REGISTRO;
            if(Registro >= TotalRegistros)
                Console.WriteLine("Número de registro inválido.");
            else {
                Flujo.Position = Registro * Combo.LONG_REGISTRO;
                Combo C = Combo.Deserialize(Entrada);
                Console.WriteLine("Leido = " + C);
                Console.WriteLine("Nuevo precio:");
                C.SetPrecio(Double.Parse(Console.ReadLine()));
                Flujo.Position = Registro * Combo.LONG_REGISTRO;
                C.Serialize(Salida);
            }
        }
    }
```

La clase Combo utiliza arreglos de carácteres en lugar de objetos string de manera que se controle el tamaño de los datos escritos y leídos desde el flujo. Es interesante notar el hecho de que se utiliza, para el cálculo del tamaño de los datos escritos al flujo, el tamaño de un carácter como de 1 byte, cuando realmente un dato tipo char en

.NET ocupa 2 bytes. Esto se debe a la forma al sistema de codificación de carácteres utilizados por defecto por los flujos creados. Bajo el sistema de codificación de carácteres por defecto, se escribe en el flujo la representación ASCII, de 1 byte por char, de cada carácter UNICODE utilizada por los tipos char, de 2 bytes por carácter.

Para moverse por el archivo de manera aleatoria, se modifica el valor de la propiedad pública de lectura y escritura **Position** de la clase FileStream. Esta clase también ofrece un método alternativo que provee de mayor funcionalidad para desplazarse por el archivo.

Manejo desde Java

Flujos

Java maneja las entradas y salidas de datos, que no sean mediante componentes GUI, mediante **flujos** o **streams**. Un flujo es una secuencia de datos que son comúnmente procesados o leídos en el mismo orden en que fueron colocados o escritos en dicho flujo. Un flujo puede ser de entrada o de salida y si bien comúnmente son procesados secuencialmente, según el tipo del flujo, también es posible procesarlos aleatoriamente.

El concepto de flujo es implementado por Java para manejar, de manera uniforme:

- La entrada, salida y error estándar.
- La lectura y escritura de archivos.
- La lectura y escritura de objetos en buffers de memoria.

Cada flujo es manejado por un objeto de alguna de las clases definidas en el paquete java.io para este fin. La figura 10.5 muestra la jerarquía de las principales clases de manejo de flujos:

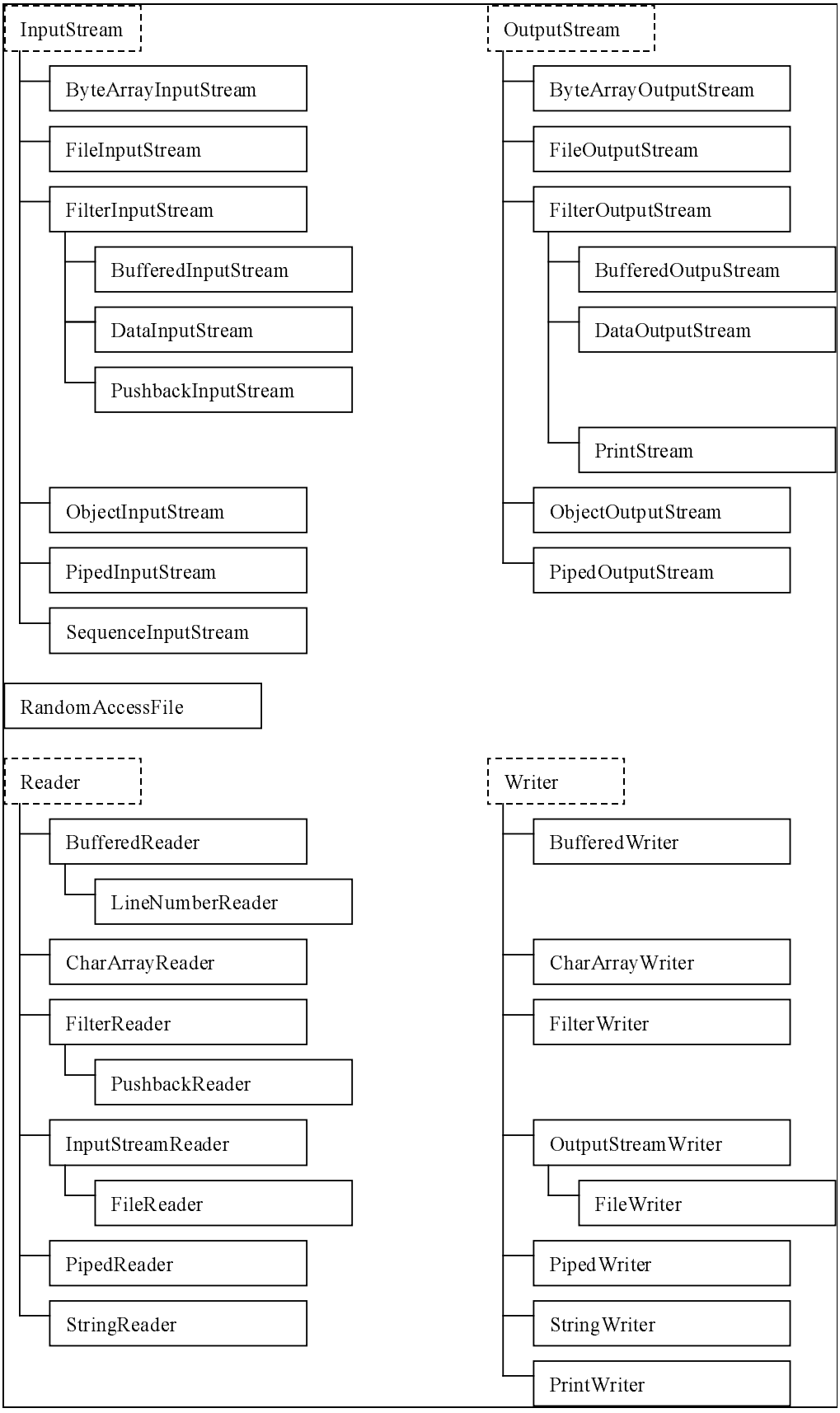

Figura 10.5. Las clases para manejo de flujos en Java

A continuación una breve descripción de la funcionalidad de cada una de estas clases:

- **InputStream / OutputStream:** Clases base abstractas para lectura / escritura de flujos de bytes. Permite leer/escribir bytes y arreglos de bytes.

- **ByteArrayInputStream / ByteArrayOutputStream:** Permiten el manejo de un arreglo de bytes como un flujo. ByteArrayInputStream se crea en base a un arreglo de bytes externo. ByteArrayOutputStream crea su propio arreglo de bytes.

- **FileInputStream / FileOutputStream:** Permiten el manejo de archivos como flujos. Ambos se crean en base a un archivo.

- **FilterInputStream / FilterOutputStream:** Clases base para todos los filtros de flujos de bytes. FilterInputStream se crea en base a un objeto InputStream. FilterOutputStream se crea en base a un objeto OutputStream.

- **BufferedInputStream / BufferedOutpuStream:** Proveen buferizacion en memoria de entrada / salida, mejorando el desempeño de ésta. Esto es útil cuando se trabaja con un medio de acceso lento con respecto a la memoria directa (RAM), por ejemplo un archivo. BufferedInputStream se crea en base a un objeto InputStream. BufferedOutputStream se crea en base a un objeto OutputStream.

- **DataInputStream / DataOutputStream:** Permiten leer / escribir tipos de datos primitivos de Java de un flujo de bytes en forma independiente a la plataforma. DataInputStream se crea en base a un objeto InputStream. DataOutputStream se crea en base a un objeto OutputStream.

- **PushbackInputStream:** Permite retornar bytes leídos de un flujo de entrada, para volver a ser leídos posteriormente. Esto es útil cuando se realiza reconocimiento de elementos en donde existen casos donde sólo es posible determinar que un elemento fue completamente leído cuando se comenzó a leer otro, por lo que, lo más conveniente es retornar la última lectura al flujo para luego volver a iniciar el reconocimiento del nuevo elemento. Se crea en base a un objeto InputStream.

- **PrintStream:** Agrega funcionalidad a OutputStream permitiendo serializar datos primitivos y objetos en base a sus representaciones como texto. Cada carácter del texto enviado se convierte a su representación estándar en bytes, en base a la codificación por defecto de carácteres de la plataforma actual. Se crea en base a un objeto OutputStream.

- **ObjectInputStream / ObjectOutputStream:** Agregan funcionalidad a InputStream / OutputStream permitiendo serializar / deserializar datos primitivos y objetos. Los objetos deben implementar la interface java.io.Serializable para poder ser serializados. Dicha interfaz no define ningún método, sólo sirve como forma de marcar a un objeto como serializable. Si un objeto referencia a otros, éstos también son serializados. ObjectInputStream se crea en base a un objeto InputStream. ObjectOutputStream se crea en base a un objeto OutputStream.

- **PipedInputStream / PipedOutputStream:** Permiten leer / escribir bytes y arreglos de bytes a pipes (tuberías). Un pipe es un objeto de comunicación unidireccional entre hilos. Un objeto PipedInputStream es conectado a un objeto PipedOutputStream. El primero es utilizado por un hilo, el cual escribe bytes que son leídos por otro hilo utilizando el segundo objeto.

- **SequenceInputStream:** Permite la concatenación lógica de varios objetos InputStream de forma que cuando se llega al final de la lectura de los datos de uno de ellos, se continúa leyendo en el siguiente, de forma transparente. Se crea en base a 2 o más objetos InputStream.

- **RandomAccessFile:** Permite manejar un archivo como un flujo secuencial o aleatorio. Se crea en base a un archivo especificándose el tipo de acceso que se requiere: de lectura, de escritura o ambos. Permite la lectura y escritura de bytes, arreglos de bytes, datos primitivos y cadenas de texto en un formato especial.

- **Reader / Writer:** Clases base abstractas para lectura / escritura de flujos de carácteres. Permite leer / escribir carácteres y arreglos de carácteres. Writer permite adicionalmente escribir cadenas de texto (String).

- **BufferedReader / BufferedWriter:** Provee buferización en memoria de la entrada, mejorando el desempeño de ésta. Esto es útil cuando se trabaja con un medio de acceso lento con respecto a la memoria directa (RAM), por ejemplo un archivo. Agregan la capacidad de leer líneas. BufferedReader se crea en base a un objeto Reader. BufferedWriter se crea en base a un objeto Writer.

- **LineNumberReader:** Ofrece la misma funcionalidad que BufferedReader agregando métodos para obtener y establecer el número de línea actual de lectura.

- **CharArrayReader / CharArrayWriter:** Permite el manejo de un arreglo de carácteres como un flujo. CharArrayReader se crea en base a un arreglo de carácteres externo. CharArrayWriter crea su propio arreglo de carácteres.

- **FilterReader / FilterWriter:** Clase abstracta base para todos los filtros de flujos de carácteres. FilterReader se crea en base a un objeto Reader. FilterWriter se crea en base a un objeto Writer.

- **PushbackReader:** Permite retornar carácteres leídos de un flujo de entrada, para volver a ser leídos posteriormente. Esto es útil cuando se realiza reconocimiento de elementos en un texto (como por ejemplo, un intérprete de ecuaciones aritméticas ingresadas como texto) en donde existen casos donde sólo es posible determinar que un elemento fue completamente leído cuando se comenzó a leer otro, por lo que lo mas conveniente es retornar la última lectura al flujo para luego volver a iniciar el reconocimiento del nuevo elemento. Se crea en base a un objeto Reader.

- **InputStreamReader / OutputStreamWriter:** Proveen un puente entre un flujo de bytes y un flujo de carácteres. Los bytes leídos son convertidos a carácteres, los carácteres escritos son convertidos a bytes. Las conversiones se realizan según la codificación por defecto de carácteres de la plataforma actual (llamado también charset). InputStreamReader se crea en base a un objeto InputStream. OutputStreamReader se crea en base a un objeto OutputStream.

- **FileReader / FileWriter:** Permiten la misma funcionalidad que InputStreamReader y OutputStreamWriter, pero el flujo se obtiene desde un archivo. Ambos se crean en base a un archivo.

- **PipedReader / PipedWriter:** Permiten leer / escribir carácteres y arreglos de carácteres a pipes. Un pipe es un objeto de comunicación unidireccional entre hilos. Un objeto PipedReader es conectado a un objeto PipedWriter. El primero es utilizado por un hilo, el cual escribe carácteres que son leídos por otro hilo utilizando el segundo objeto.

- **StringReader / StringWriter:** Permite el manejo de un objeto String como un flujo de carácteres. StringReader se crea en base a un objeto String externo. StringWriter crea su propio objeto String.

- **PrintWriter:** Permite enviar representaciones de texto de datos primitivos y objetos a un flujo de carácteres de salida. Se puede crear en base a un objeto OutputStream o a un objeto Writer.

Una clase de flujo comúnmente es concatenada con otra de forma que se combine la funcionalidad de ambas. Es importante tener en cuenta que, los flujos de datos son interpretados por las clases asumiendo la forma en que

fueron creados. Como ejemplo, si tuviésemos un flujo de bytes donde se escribieron datos primitivos enteros, se leerá basura si se intenta recuperarlo como los carácteres que representen dichos valores, y viceversa, si tuviésemos un flujo de carácteres con números enteros, no se puede recuperar directamente los valores enteros que estos pudiesen representar. Como conclusión, no toda combinación de flujos arrojará resultados correctos. Por lo común, un flujo de escritura es leído mediante su flujo de lectura correspondiente, por ejemplo, un flujo creado con la clase DataOutputStream es leído utilizando la clase DataInputStream.

En las siguientes secciones examinaremos los dos casos más comunes de manejo de flujos:

- El manejo de flujos desde consola.
- El manejo de flujos desde archivos.

Manejo de Consola

La salida desde consola se suele realizar utilizando directamente el objeto referenciado por el dato miembro estático **out** de la clase System. Este objeto es del tipo PrintWriter, el cual ofrece la funcionalidad suficiente para escribir datos primitivos y objetos convirtiendo estos a cadenas de texto. Los métodos más utilizados de esta clase son **print** y **println**.

Como contraparte, la clase System provee un dato miembro estático **in** del tipo BufferedInputStream. Si se desea trabajar el flujo leyendo byte a byte, este flujo es suficiente. El siguiente ejemplo muestra el uso de esta clase directamente.

```
import java.io.*;

class Consola1 {
    public static void main( String args[] ) throws IOException {
        System.out.print( "Ingrese el texto a Leer." );
        System.out.println( " Finalizar con la combinacion CTRL+Z." );
        while(true)    {
            int Entero = System.in.read();
            if( Entero == -1 )
                break;
            char Caracter = (char)Entero;
            System.out.println( "leido: numero=" + Entero +
                ", correspondiente al caracter=" + Caracter );
        }
        System.out.println( "Fin del ingreso" );
    }
}
```

La figura 10.6 muestra el resultado de una ejecución de ejemplo de este programa.

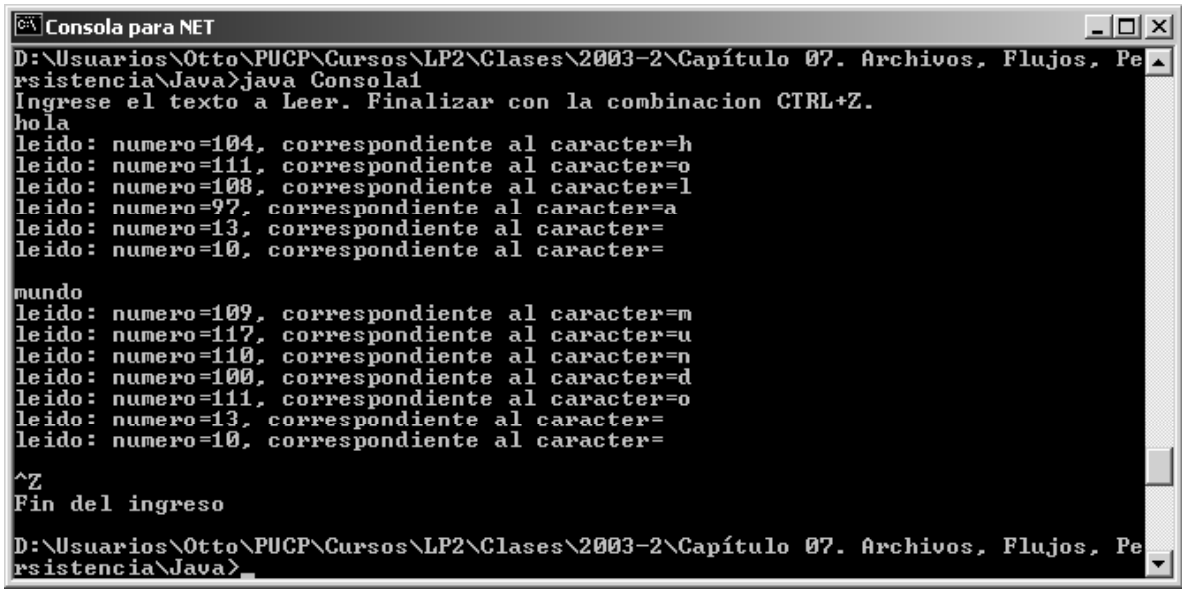

Figura 10.6. Manejo de consola en Java: Ejecución del programa

La sobrecarga del método read utilizada devuelve un valor entero entre 0 y 255 cuando la lectura es correcta. Dicho valor puede ser convertido directamente a un char. Cuando el método detecta un fin de archivo, que en el caso del ingreso por consola equivale a ingresar CTRL+Z, el valor devuelto es –1. Nótese que el retorno de carro (con código ASCII 13) y el cambio de línea (con código ASCII 10) también son leídos.

Sin embargo, en la mayoría de los casos, lo que se desea leer son líneas de texto que luego serán procesadas. Lo más común en este caso es combinar System.in con las clases InputStreamReader y BufferedReader. El siguiente ejemplo muestra el uso de esta combinación.

```
import java.io.*;

class Consola2 {
    public static void main(String args[]) throws IOException {
        System.out.println( "Ingrese una secuencia de enteros." );
        System.out.println( "Finalice el ingreso con '*'." );

        BufferedReader Entrada = new BufferedReader( new InputStreamReader( System.in ) );
        int Contador = 0;
        int Suma = 0;
        while(true) {
            String Linea = Entrada.readLine();
            if( Linea == null )
                break;
            if( Linea.charAt(0) == '*' )
                break;
            Suma += Integer.parseInt( Linea );
            Contador++;
        }
        System.out.println( "Se ingresaron " + Contador + " Enteros." );
        System.out.println( "El promedio es " + Suma / Contador );
    }
}
```

La figura 10.7 muestra el resultado de una ejecución de ejemplo de este programa.

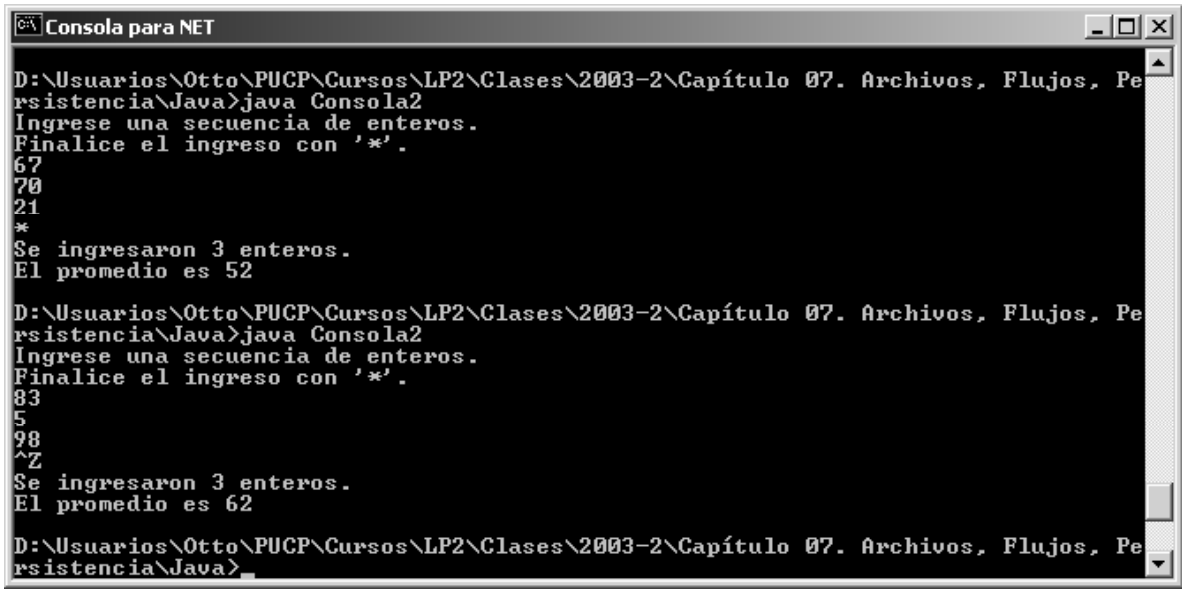

Figura 10.7. Manejo de consola en Java: Ejecución del programa

Es importante notar que el método Integer.parseInt puede arrojar una excepción si la conversión de la cadena pasada como parámetro no representa un número entero válido. Dado que no nos interesa, para este ejemplo, manejar nosotros mismo ese tipo de excepción, lo manifestamos mediante la palabra *throws* al final del encabezado del método main. En este caso no podemos dejar la excepción sin declararla, dado que IOException no es una excepción *runtime*, es decir, no deriva de la clase RuntimeException.

En otros casos es necesario procesar una línea de texto de forma que se particione ésta en elementos, cada elemento puede ser una palabra, un número, etc. Los elementos son reconocidos dado que comúnmente los separamos unos de otros mediante delimitadores, como espacios en blanco o comas. A este tipo de trabajo se llama **tokenización**. Un token es una sub-cadena de carácteres dentro de una cadena, delimitados por carácteres especiales denominados "delimitadores". Para realizar este trabajo se puede utilizar la clase StringTokenizer del paquete java.util. El siguiente ejemplo muestra el uso de esta clase.

```
import java.io.*;
import java.util.StringTokenizer;

class Consola3 {
    public static void main(String args[]) throws IOException {
        System.out.println( "Ingrese una secuencia de palabras." );

        BufferedReader Entrada = new BufferedReader( new InputStreamReader( System.in ) );
        String Linea = Entrada.readLine();

        StringTokenizer Tokens = new StringTokenizer( Linea );
        int Contador = 0;
        while ( Tokens.hasMoreTokens() )
            System.out.println( "Token " + (++Contador) + " = " + Tokens.nextToken() );
    }
}
```

La figura 10.8 muestra el resultado de una ejecución de ejemplo de este programa.

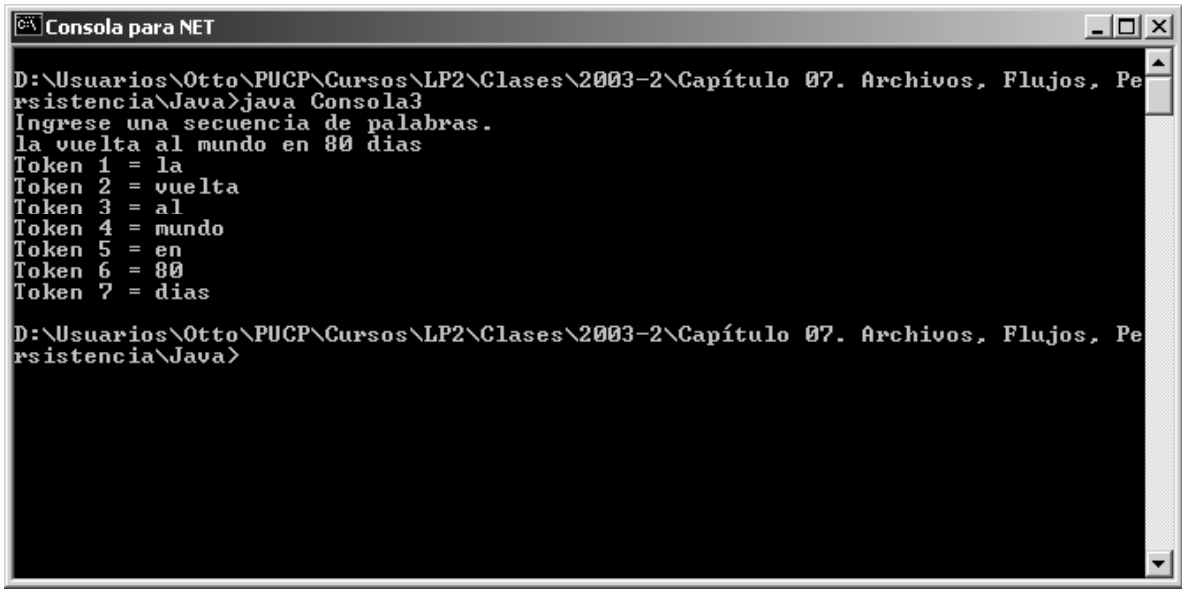

Figura 10.8. Manejo de consola en Java: Ejecución del programa

El constructor utilizado para crear el objeto StringTokenizer recibe como parámetro la cadena a tokenizar. Este constructor utiliza como delimitadores por defecto la siguiente cadena: " \t\n\r\f". Es posible utilizar un segundo constructor que permite pasar como segundo parámetro una cadena con los carácteres que deseamos funcionen como delimitadores.

Si se desea un manejo más complejo de un flujo de texto, reconociéndose cadenas de texto, números, comentarios, etc. (a la manera como un compilador procesa un archivo fuente), se puede utilizar la clase StreamTokenizer del paquete java.io. El uso de esta clase escapa de los alcances del curso.

Por último, al igual que en la programación en C y C++, son 3 los flujos estándar que ofrece Java mediante datos miembro (referencias a objetos de flujo) de la clase System:

- System.in Entrada estándar, por defecto direccionado al teclado desde la consola.
- System.out Salida estándar, por defecto direccionado a la pantalla de la consola.
- System.err Salida de error estándar, por defecto direccionado a la pantalla de la consola.

Cualquiera de éstos puede ser redireccionado por el programa hacia, por ejemplo, un archivo. El proceso de redireccionamiento consiste simplemente es asignar un nuevo objeto a estas referencias.

Es importante recordar que todo programa en Java, sea de consola o gráfico, crea automáticamente una consola, por lo que a ésta siempre es posible utilizarla.

Manejo de Archivos

Cuando un programa accede a un archivo podría, dependiendo del modo de acceso solicitado, leer cualquier parte de el directamente, sin necesidad de seguir un orden, cosa muy diferente al concepto de un flujo el cual funciona como una cola o FIFO (First In First Out). Si bien los archivos pueden trabajarse lógicamente como un flujo, lo que equivale al conocido modo de acceso secuencial, también es posible, dado las características propias de éste, trabajarlo accediendo aleatoriamente.

Se examinaran los siguientes casos de manejo de archivos:

- Manejo de archivos de texto.
- Manejo de archivos binarios secuencialmente.
- Manejo de archivos binarios aleatoriamente.

Archivos de Texto

Los archivos de texto se manejan comúnmente utilizando las clases FileWriter y FileReader. El siguiente programa muestra en uso de estas clases.

```
import java.io.*;

class ArchivoTexto {
    public static void main(String args[]) throws IOException {
        FileWriter Escritor = new FileWriter( args[0] );
        System.out.println( "Ingrese el texto a almacenar: " );

        BufferedReader Entrada = new BufferedReader( new InputStreamReader( System.in ) );
        while(true) {
            String Linea = Entrada.readLine();
            if( Linea == null )
                break;
            Escritor.write( Linea + "\n"  );
        }
        Escritor.close();
        Entrada.close();

        System.out.println( "Leyendo el archivo creado." );
        FileReader Lector = new FileReader( args[0] );
        while( Lector.ready() )
            System.out.print( (char)Lector.read() );
        Lector.close();
    }
}
```

La figura 10.9 muestra el resultado de una ejecución de ejemplo de este programa.

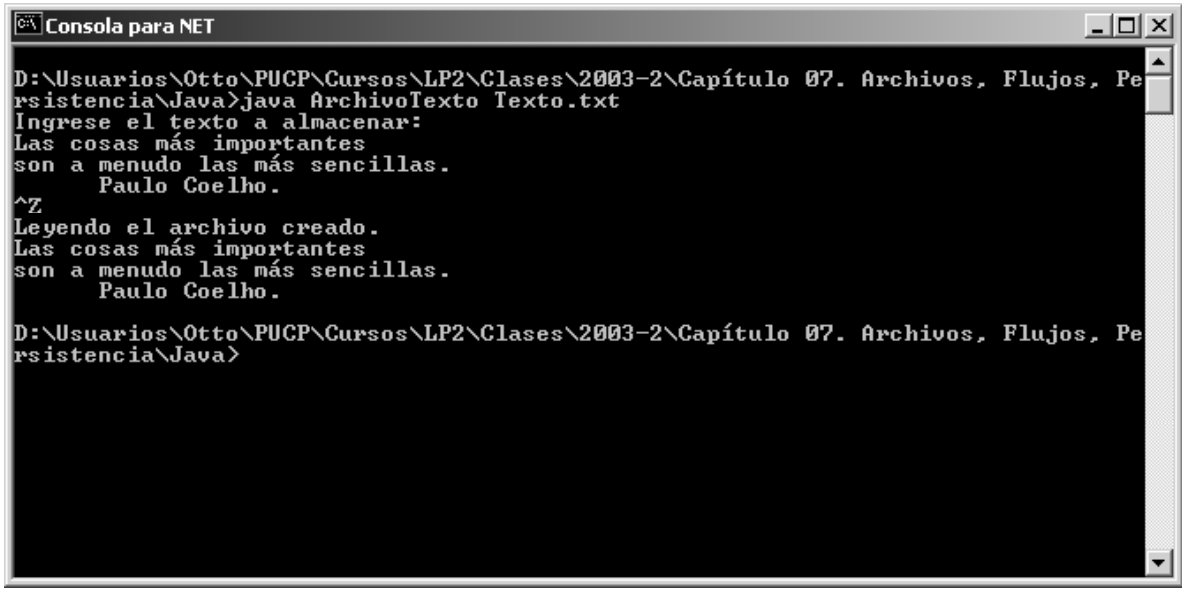

Figura 10.9. Manejo de consola en Java: Ejecución del programa

Note que la funcionalidad ofrecida por FileWriter es diferente de la de FileReader. FileWriter extiende Writer, el cual permite escribir objetos de tipo cadena de texto (String), lo cual facilita mucho la escritura. FileReader extiende Reader, el cual sólo permite leer carácteres y arreglos de carácteres. Lo común es leer línea por línea un texto. Para realizar esto podemos combinar la clase FileReader con la clase BufferedReader, dado que la última provee lectura de líneas completas. El código de lectura del programa pudo reemplazarse de la siguiente forma:

```
BufferedReader Lector = new BufferedReader( new FileReader( args[0] ) );
while( Lector.ready( ) )
    System.out.println( Lector.readLine( ) );
Lector.close();
```

De igual forma, si deseamos utilizar las capacidades de la clase PrintWriter, como con el objeto referenciado por System.out, podríamos utilizar un código como el siguiente:

```
PrintWriter Escritor = new PrintWriter( new FileWriter( args[0] ) );
    BufferedReader Entrada = new BufferedReader( new InputStreamReader( System.in ) );
while(true) {
        String Linea = Entrada.readLine();
        if( Linea == null )
            break;
    Escritor.println( Linea );
}
Escritor.close();
```

Archivos Binarios Secuenciales

Para crear un stream binario desde un archivo se utilizan las clases FileInputStream y FileOutputStream para lectura y escritura respectivamente. Una vez creado los flujos relacionados a los archivos, es común utilizar las clases BufferedInputStream y BufferedOutputStream respectivamente si se desea mejorar la performance cuando se realizan muchas lecturas o escrituras desde y hacia disco.

Para leer y escribir datos de tipo primitivo se suele utilizar las clases DataInputStream y DataOutputStream. Luego, la creación de los streams de lectura y escritura para estos casos sería:

```
DataInputStream input = new DataInputStream(
new BufferedInputStream( new FileInputStream( “MiArchivo.dat” ) ) );

DataOutputStream output = new DataOutputStream(
new BufferedOutputStream( new FileOutputStream( “MiArchivo.dat” ) ) );
```

Para los programas que sólo requieren almacenar y recuperar algunos datos simples el uso de estas clases puede ser suficiente, para programas complejos la información que se maneja provienen de objetos con datos de configuración o bien arreglos de objetos, es decir, más que datos primitivos lo que se desea es almacenar y recuperar objetos completos. Aquí entran a tallar dos conceptos importantes: **Persistencia** y **serialización**.

Se dice que un objeto es persistente si su existencia trasciende el de su creador y sus usuarios. Para nuestro punto de vista como programadores, quien crea y usa un objeto es un programa.

La serialización es una forma de implementación del concepto de persistencia, utilizando para esto un medio de almacenamiento también persistente, como el disco duro (su información se conserva mas allá de que cada instancia de ejecución de los programas que crean o utilizan la información, inclusive del sistema operativo mismo y de que la maquina se apague o prenda). La serialización consiste en la lectura y escritura de objetos a y desde un flujo amarrado a dicho medio persistente.

Para serializar un objeto es necesario marcar su clase como serializable. Esto consiste en hacer que la clase a la que pertenece el objeto que deseamos serializar implemente la interfaz Serializable. Dicha interfaz no define

ningún método ni constante, sólo sirve para marcar la clase. Si la clase contiene datos miembro que sean referencias, los tipos de dichas referencias también deben marcarse como serializables.

Las clases que permiten la serialización de objetos son ObjectInputStream y ObjectOutputStream.

ObjectOutputStream define un método writeObject el cual recibe como parámetro un objeto mediante una referencia de tipo Object. Este método se encarga de averiguar si la clase fue marcada como Serializable. Si no fue marcada se produce una excepción en tiempo de ejecución. Si fue marcada, se serializa tanto la información concerniente al tipo de objeto marcado (de forma que después pueda recrearce el mismo tipo de objeto), como los datos del objeto.

ObjectInputStream define un método readObject, el cual no recibe parámetros y retorna una referencia de tipo Object del objeto leído y recreado. El método utiliza la información almacenada sobre el tipo del objeto original para recrear uno del mismo tipo, y la información sobre sus datos miembro para inicializar los del objeto.

El siguiente programa muestra el uso de estas clases.

```
import java.io.*;

class Direccion implements Serializable {
    int numero;
    String calle;
    String distrito;
    public Direccion(int numero, String calle, String distrito)       {
        this.numero = numero;
        this.calle = calle;
        this.distrito = distrito;
    }
    public String toString() {
        return calle + " " + numero + ", " + distrito;
    }
}

class Persona implements Serializable {
    private String nombre;
    private int edad;
    private Direccion dir;
    public Persona(String nombre, int edad, Direccion dir) {
        this.nombre = nombre;
        this.edad = edad;
        this.dir = dir;
    }
    public String toString() {
        return "Sr(a). " + nombre + ", " + edad + " años, direccion = " + dir;
    }
}

class ArchivoBinarioSecuencial {
    public static void main(String args[]) throws IOException, ClassNotFoundException {
        if(args.length < 2) {
            System.out.println("Error en argumentos.");
            return;
        }
        if(args[0].equals("/e")) {
            ObjectOutputStream Escritor = new ObjectOutputStream(
                new BufferedOutputStream(new FileOutputStream(args[1])));
            Persona[] ListaPersonas = {
                new Persona("Jose", 25, new Direccion(123, "Jose Leal", "San Juan")),
                new Persona("Mara", 26, new Direccion(456, "Carrión", "Barranco")),
                new Persona("Ana", 35, new Direccion(789, "Fresnos", "Lince"))
            };
            for(int i = 0; i < ListaPersonas.length; i++)
                Escritor.writeObject( ListaPersonas[i] );
            Escritor.close();
        } else if(args[0].equals("/l")) {
            ObjectInputStream Lector = new ObjectInputStream(
                new BufferedInputStream(new FileInputStream(args[1])));
```

```
            while(true) {
                Persona p;
                try { p = (Persona)Lector.readObject(); }
                catch(EOFException ex){ break; }
                System.out.println("Persona : " + p);
            }
            Lector.close();
        }
    }
}
```

La figura 10.10 muestra el resultado de una ejecución de ejemplo de este programa.

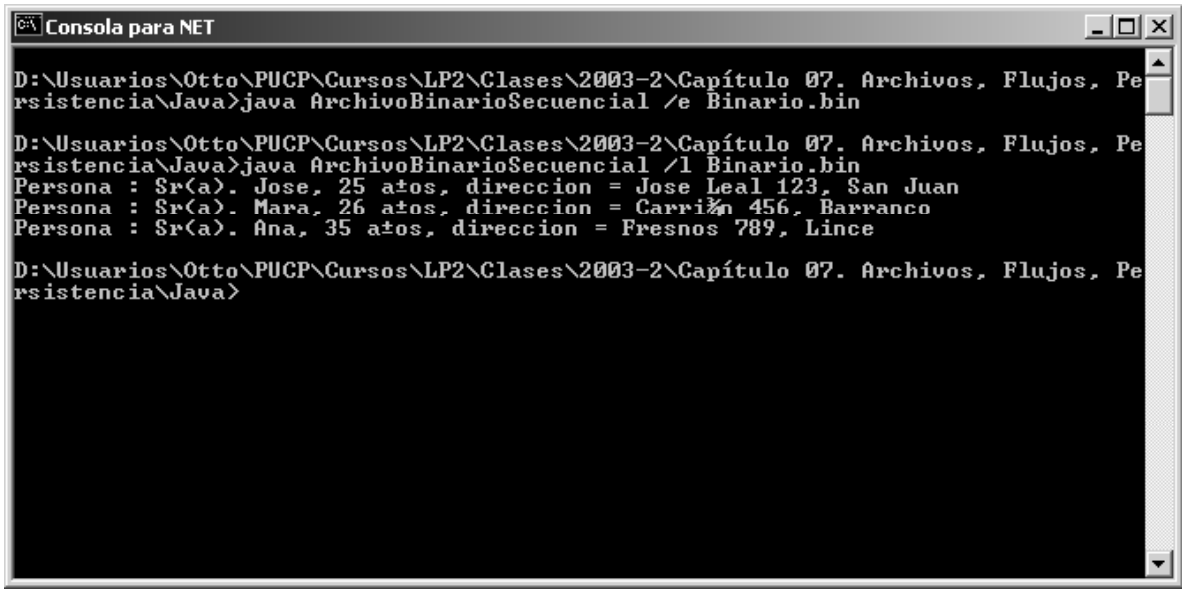

Figura 10.10. Archivos binarios secuenciales en Java: Ejecución de un ejemplo

Archivos Binarios Aleatorios

Para manejar aleatoriamente un archivo se utiliza la clase RandomAccessFile. Esta clase provee la misma funcionalidad de las clases DataInputStream y DataOutputStream. Ésta permite manejar un archivo como un arreglo de bytes, pudiendo posicionarnos en cualquier lugar del arreglo para realizar una lectura o escritura. El siguiente programa muestra el uso de esta clase:

```
import java.io.*;

class ArchivoBinarioAleatorio {
    public static void main(String args[]) throws IOException {
        RandomAccessFile Lector = new RandomAccessFile( "datos.dat", "rw" );
        Lector.writeInt( 100 );
        Lector.writeBoolean( false );
        Lector.writeDouble( 6.54 );
        Lector.writeUTF( "Hola mundo" );
        System.out.println( "Archivo llenado" );
        System.out.println( "Tamano del archivo = " + Lector.length() );

        System.out.println( "\nLeyendo el archivo" );
        Lector.seek( 0 );
        System.out.println( "Entero = " + Lector.readInt() );
        System.out.println( "Booleano = " + Lector.readBoolean() );
        System.out.println( "Punto Flotante = " + Lector.readDouble() );
        System.out.println( "Cadena = " + Lector.readUTF() );
```

```
            Lector.close();
        }
    }
```

La figura 10.11 muestra el resultado de una ejecución de ejemplo de este programa.

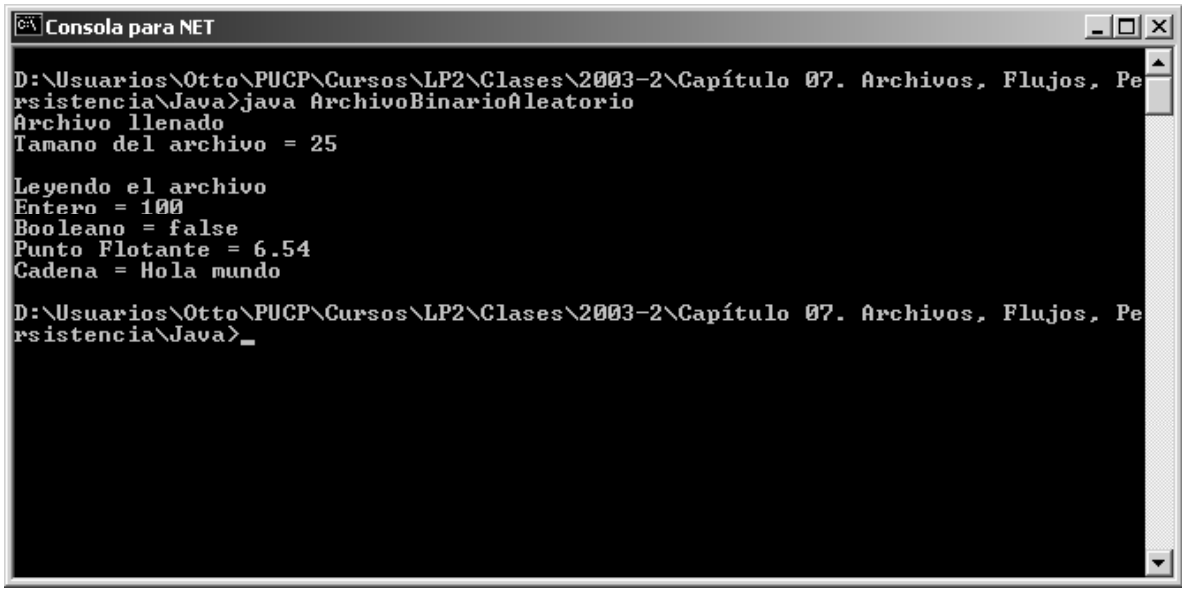

Figura 10.11. Archivos binarios aleatorios en Java: Ejecución del programa

Para cada tipo de dato primitivo se cuenta con un método read y un método write adecuado. Cada vez que realizamos una lectura o una escritura, el apuntador del archivo se desplaza el número de bytes acorde con el tamaño del dato leído. En cualquier momento podemos recolocar el apuntador mediante el método *seek*, el cual recibe como parámetro un entero que expresa el número de bytes a desplazarse con respecto al inicio del archivo.

Note que sólo tenemos la capacidad de manejar datos primitivos y cadenas de texto en formato UTF y si bien es posible que nos desplacemos a cualquier posición del archivo para realizar una lectura a menos que conozcamos qué datos están guardados y en qué orden, o bien todos los datos guardados sean del mismo tipo y tamaño, resultará imposible leer cualquier dato aleatoriamente. Adicionalmente, dado que no es posible escribir y leer objetos, tampoco es posible manejar el archivo con una estructura de registros, como es la costumbre para archivos binarios en C y C++. Tampoco es posible combinar la clase RandomAccessFile con otra clase de flujo, dado que no cuenta con constructores que acepten estos. Tampoco hereda de otra clase de flujo, por lo que no es posible pasar una referencia del tipo RandomAccessFile a otro flujo. Esta clase está pensada para ser utilizada sola.

Debido a que la extensa librería de clases de Java está pensada para solucionar los problemas más comunes de programación, es de esperar que exista alguna clase o paquete que permita poder manejar un juego de registros en archivos y acceder a estos registros directamente, esto es, de manera aleatoria, sin necesidad de tener que recorrer toda la información desde el inicio cada vez, hasta llegar a dicha información. Este tipo de trabajo es en mucho lo que permite un sistema de manejo de base de datos. Luego, la solución común cuando se desea realizar este tipo de trabajo es utilizar los paquetes de la librería JDBC para crear y administrar base de datos. Otra solución posible es extender la clase RandomAccessFile para permitir escribir y leer objetos y proveer alguna funcionalidad adicional para poder utilizar llaves únicas a cada objeto/registro guardado, de forma que se pueda acceder directamente a dicha información. Estos aspectos van más allá del alcance del curso.