Guía práctica de estudio 11:
Manejo de archivos

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 11:
Manejo de archivos

Objetivo:

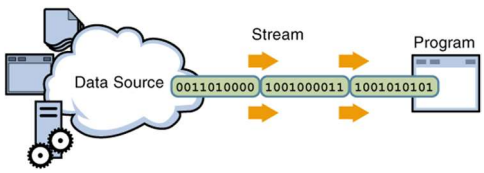
Implementar el intercambio de datos (lectura y escritura) entre fuentes externas (archivos y/o entrada y salida estándar) y un programa (en un lenguaje orientado a objetos).

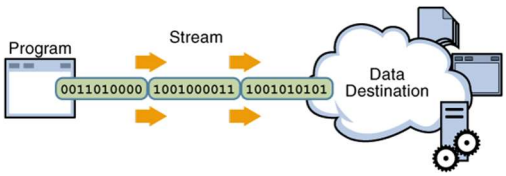
Actividades:

- Crear archivos de texto plano.
- Leer archivos de texto plano.
- Escribir en archivos de texto plano.

Introducción

Los programas necesitan comunicarse con su entorno, tanto para obtener datos e información que deben procesar, como para devolver los resultados obtenidos. El manejo de archivos se realiza a través de **streams** o **flujos de datos** desde una fuente hacia un repositorio. La fuente inicia el flujo de datos, por lo tanto, se conoce como flujo de datos de entrada. El repositorio termina el flujo de datos, por lo tanto, se conoce como flujo de datos de salida. Es decir, tanto la fuente como el repositorio son nodos de flujos de datos.





Archivos

Un **archivo** es un objeto en una computadora que puede almacenar información, configuraciones o comandos, el cual puede ser manipulado como una entidad por el sistema operativo o por cualquier programa o aplicación. Un archivo debe tener un nombre único dentro de la carpeta que lo contiene. Normalmente, el nombre de un archivo contiene un sufijo (extensión) que permite identificar el tipo del archivo.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Flujos de datos

Las entradas y las salidas de datos en java se manejan mediante **streams** (flujos de datos). Un **stream** es una conexión entre el programa y la fuente (lectura) o el destino (escritura) de los datos. La información se traslada en serie a través de esta conexión.

En Java existen 4 jerarquías de clases relacionadas con los flujos de entrada y salida de datos.

- **Flujos de bytes**: las clases derivadas de *InputStream* (para lectura) y de *OutputStream* (para escritura), las cuales manejan los flujos de datos como stream de bytes.

- **Flujos de caracteres**: las clases derivadas de *Reader* (para lectura) y *Writer* (para escritura), las cuales manejan stream de caracteres.

Todas las clases de Java relacionadas con la entrada y salida se agrupan en el paquete *java.io*.

Clase File

La clase **File** permite manejar archivos o carpetas, es decir, crear y borrar tanto archivos como carpetas, entre otras funciones.

Cuando se crea una instancia de la clase *File* no se crea ningún archivo o directorio, solo se crea una referencia hacia un objeto de este tipo. La creación de archivos o carpetas se realizan de manera explícita, invocando a los métodos respectivos. A continuación se enlistan los métodos más útiles que posee la clase File:

- *exists()*
- *createNewFile()*
- *mkdir()*
- *delete()*
- *renameTo()*
- *list()*

Ejemplo:

```
import java.io. File;
import java.io. IOException;

class Escribe {

    public static void main(String [] args) {
        try {
            File archivo = new File("archivo.txt");
            System.out.println(archivo.exists());
            boolean seCreo = archivo.createNewFile();
            System.out.println(seCreo);
            System.out.println(archivo.exists());
        } catch(IOException e) { }
    }

}
```

FileOutputStream

La clase **FileOutputStream** permite crear y escribir un flujo de bytes en un archivo de texto plano. Esta clase hereda de la clase *OutputStream*. Sus constructores más comunes son:

- *FileOutputStream (String nombre)*
- *FileOutputStream (String nombre, boolean añadir)*
- *FileOutputStream (File archivo)*

FileInputStream

FileInputStream permite leer flujos de bytes desde un archivo de texto plano. Hereda de la clase *InputStream*. Sus constructores más comunes son:

- *FileInputStream(String nombre)*
- *FileInputStream(File archivo)*

Ejemplo:

Escritura usando *FileOutputStream*

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ClaseFileOutputStream {

    public static void main (String [] args){
        FileOutputStream fos = null;
        byte[] buffer = new byte[81];
        int nBytes;
        try {
            System.out.println("Escribir el texto a guardar en el archivo:");
            nBytes = System.in.read(buffer);
            fos = new FileOutputStream("fos.txt");
            fos.write(buffer,0,nBytes);
        } catch (IOException ioe){
            System.out.println("Error: " + ioe.toString());
        } finally {
            try {
                if (fos != null) fos.close();
            } catch (IOException ioe) {
                System.out.println("Error al cerrar el archivo.");
            }
        }
    }

}
```

Lectura usando *FileInputStream*

```
import java.io.FileInputStream;
import java.io.IOException;

public class ClaseFileInputStream {
    public static void main (String [] args){
        FileInputStream fis = null;
        byte[] buffer = new byte[81];
        int nbytes;
        try {
            fis = new FileInputStream("leer.txt");
            nbytes = fis.read(buffer, 0, 81);
            String texto = new String(buffer, 0, nbytes);
            System.out.println(texto);
        } catch (IOException ioe) {
            System.out.println("Error: " + ioe.toString());
        } finally {
            try {
                if (fis != null) fis.close();
            } catch (IOException ioe) {
                System.out.println("Error al cerrar el archivo.");
            }
        }
    }
}
```

FileWriter

La clase **FileWriter** hereda de *Writer* y permite escribir un flujo de caracteres en un archivo de texto plano.

BufferedWriter

La clase **BufferedWriter** también deriva de la clase *Writer* y permite crear un buffer para realizar una escritura eficiente de caracteres desde la aplicación hacía el archivo destino.

PrintWriter

La clase **PrintWriter**, que también deriva de *Writer*, permite escribir de forma sencilla en un archivo de texto plano. Posee los métodos *print* y *println*, idénticos a los de *System.out*, y el método *close( )*, el cual cierra el stream de datos.

FileReader

Las clases **Reader** se utilizan para obtener los caracteres ingresados desde una fuente. La clase **FileReader** hereda de *Reader* y permite leer flujos de caracteres de un archivo de texto plano.

InputStreamReader

InputStreamReader es una clase que deriva de *Reader* que convierte los streams de bytes a streams de caracteres. *System.in* es el objeto de la clase *InputStream* el cual recibe datos desde la entrada estándar del sistema (el teclado).

BufferedReader

La clase **BufferedReader**, que también deriva de la clase *Reader*, crea un buffer para realizar una lectura eficiente de caracteres. Dispone del método *readLine* que permite leer una línea de texto y tiene como valor de retorno un *String*.

Ejemplo:

Lectura usando *FileReader*

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ClaseFileReader{
    public static void main (String [] escribir){
        String texto = "";
        try {
            BufferedReader br;
            FileReader fr = new FileReader("leer.txt");
            br = new BufferedReader(fr);
            System.out.println("El texto contenido en el archivo leer.txt es:");
            String linea = br.readLine();
            while (linea != null ) {
                System.out.println(linea);
                linea = br.readLine();
            }
            br.close();
        } catch (IOException ioe){
            System.out.println("\n\nError al abrir o guardar el archivo:");
            ioe.printStackTrace();
        } catch (Exception e){
            System.out.println("\n\nError al leer de teclado:");
            e.printStackTrace();
        }
    }
}
```

Escritura usando *FileWriter*

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.IOException;

public class ClaseFileWriter{
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();
            FileWriter fw = new FileWriter("archivo.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter salida = new PrintWriter(bw);
            salida.println(texto);
            salida.close();
        } catch (IOException ioe){
            System.out.println("\n\nError al abrir o guardar el archivo:");
            ioe.printStackTrace();
        } catch (Exception e){
            System.out.println("\n\nError al leer de teclado:");
            e.printStackTrace();
        }
    }
}
```

Lectura de teclado

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class LeeTecladoBR {
    public static void main (String [] args){
        try {
            String texto = "";
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir el texto deseado:");
            texto = br.readLine();
            System.out.println("El texto escrito fue: " + texto);
        } catch (IOException ioe){
            System.out.println("Error al leer caracteres: \n" + ioe);
        }
    }
}
```

StringTokenizer

La clase **StringTokenizer** permite separar una cadena de texto por palabras (espacios) o por algún otro carácter. La clase *StringTokenizer* pertenece al paquete *java.util.*

Ejemplo:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class LeeTecladoCompleto {
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();

            System.out.println("\n\nEl texto separado por espacios es:");
            StringTokenizer st = new StringTokenizer(texto);
            while(st.hasMoreTokens()) {
                System.out.println(st.nextToken());
            }
        } catch (Exception e){
            System.out.println("\n\nError al leer de teclado:");
            e.printStackTrace();
        }
    }
}
```

Scanner

La clase **Scanner** permite leer flujos de bytes desde la entrada estándar. Pertenece al paquete *java.util*.

Los métodos principales de esta clase son *next( )* y *hasNext( )*. El método *next( )* obtiene el siguiente elemento del flujo de datos. El método *hasNext( )* verifica si el flujo de datos todavía posee elementos, en caso afirmativo regresa *true*, de lo contrario regresa *false*. El

delimitador de la clase *Scanner* (para obtener el siguiente elemento), por defecto, es el espacio en blanco, aunque es posible cambiar el delimitador utilizando el método *useDelimiter* que recibe como parámetro el delimitador (en forma de *String*).

Ejemplo:

```
import java.util.Scanner;

public class EjScanner {

    public static void main(String [] args) {
        try {
            String cad = "";
            Scanner s = new Scanner(System.in);
            System.out.println(s.nextLine());
            s.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

Console

La clase **Console** permite recibir flujos de datos desde la **línea de comandos** (entrada estándar). Se encuentra dentro del paquete java.io. Entre los métodos importantes que posee se encuentran:

- **readLine( ):** lee una cadena de caracteres hasta que encuentra el salto de línea (enter).
- **readPassword( ):** lee una cadena de caracteres hasta que encuentra el salto de línea (enter), ocultando los caracteres como lo hace el sistema operativo que se utilice.

Ejemplo:

```
import java.io.Console;

public class EjConsole {

    public static void main(String [] args){
        Console con = System.console();
        System.out.print("Usuario: ");
        String s = con.readLine();
        System.out.println(s);
        System.out.print("Contraseña: ");
        char [] s2 = con.readPassword();
        System.out.println(s2);
    }
}
```

Serialización

La **serialización** es un mecanismo para guardar los objetos como una secuencia de bytes y poderlos reconstruir en un futuro cuando se necesiten, conservando su estado. Cuando un objeto se **serializa** solo los campos del objeto (atributos) son preservados.

Si un campo hace referencia a un objeto, esta referencia debe ser también **serializable**. Los objetos que no son **serializables** se deben declarar con la palabra reservada **transient** para evitar que se intenten serializar y se genere un error.

Ejemplo:

Serializar un objeto Date

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Date;

public class SerializeDate {

    SerializeDate() {
        Date d = new Date ();
        System.out.println(d);
        try {
            FileOutputStream f = new FileOutputStream ("date.ser");
            ObjectOutputStream s = new ObjectOutputStream (f);
            s.writeObject (d);
            s.close ();
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }

    public static void main (String args[]) {
        new SerializeDate();
    }
}
```

Deserializar objeto Date

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Date;

public class DeSerializeDate {

    DeSerializeDate () {
        Date d = null;
        try {
            FileInputStream f = new FileInputStream ("date.ser");
            ObjectInputStream s = new ObjectInputStream (f);
            d = (Date) s.readObject ();
            s.close ();
        } catch (Exception e) {
            e.printStackTrace ();
        }
        System.out.println( "Deserialized Date object from date.ser");
        System.out.println("Date: "+d);
    }

    public static void main (String args[]) {
        new DeSerializeDate();
    }
}
```

Bibliografía

Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill

Deitel Paul, Deitel Harvey.
Como programar en Java
Septima Edición.
México
Pearson Educación, 2008

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009