Programación 2

Entrada/Salida con Archivos en Java

Profesores:
Ismael Figueroa - ifigueroap@gmail.com
Eduardo Godoy - eduardo.gl@gmail.com

Librerías de Entrada/Salida para Archivos

En Java 7 y superior coexisten dos librerías para manejo de flujos con archivos:

- `java.io`: librería introducida desde Java 1.0, hoy se considera como "deprecated", pero es relativamente sencilla de usar. Funciona de manera secuencial y bloqueante, en base a flujos, similar al uso de `Scanner` o `System.out`

- `java.nio`: librería moderna de Entrada/Salida, muy flexible y configurable, pero en general más complicada de usar. Está orientada además al procesamiento concurrente de E/S.

Importante!

En el curso trabajaremos solamente con `java.io`

Flujos de Entrada/Salida

Un flujo de entrada/salida representa una fuente de entrada o un destino de salida. Este concepto general puede representar diversos tipos de fuentes y destinos: archivos, dispositivos, otros programas, arreglos, etc.

Los flujos soportan distintos tipos de datos: bytes, tipos de datos primitivos, u objetos. No obstante, *todos los flujos tienen el mismo modelo de programación*

Modelo de Flujos Entrada/Salida

Para el programador todos los flujos de entrada/salida funcionan de manera similar: los flujos son construcciones secuenciales de datos.

- Un *flujo de entrada* se usa en un programa para *leer datos desde la fuente*, de a un elemento cada vez.
- Un *flujo de salida* se usa para *escribir datos en un destino*, también de un elemento a la vez.

Byte Streams

Definición

Se usan para trabajar directamente con bytes, o sea items de 8 bits. Se usan en general para trabajar con datos/archivos binarios, como por ejemplo imagenes.

Clases para uso de Byte Streams:

- `java.io.FileInputStream`: para leer archivos byte por byte. Es subclase de `java.io.InputStream`.
- `java.io.FileOutputStream`: para escribir archivos byte por byte. Es subclase de `java.io.OutputStream`.

Ejemplo Byte Stream: Copiar archivo byte por byte

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

public class Main {
  public static void main(String[] args) throws IOException {
    FileInputStream  fin  = null;
    FileOutputStream fout = null;
    int c;
    try {
      fin  = new FileInputStream("hola.txt");
      fout = new FileOutputStream("salida.txt");
      while((c = fin.read()) != -1) {
        fout.write(c);
      }
    } finally {
      if (fin != null)  { fin.close(); }
      if (fout != null) { fout.close(); }
    }
  }}
```

Character Streams

Definición

Se usan para trabajar con texto en base a caracteres. En Java los caracteres se almacenan usando Unicode, y cuando se procesan, el sistema considera la transformación al juego de caracteres local, según el país o región que esté configurada.

Clases para uso de Character Streams:

- `java.io.FileReader`: para leer archivos de texto, caracter por caracter.
- `java.io.FileWriter`: para escribir archivos de texto, caracter por caracter.

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

public class Main {
  public static void main(String[] args) throws IOException {
    FileReader  fin  = null;
    FileWriter fout = null;
    int c;
    try {
      fin  = new FileReader("hola.txt");
      fout = new FileWriter("salida.txt");
      while((c = fin.read()) != -1) {
        fout.write(c);
      }
    } finally {
      if (fin != null)  { fin.close(); }
      if (fout != null) { fout.close(); }
    }
  }}
```

Buffered Streams

El uso directo de los flujos de entrada o salida es poco eficiente porque cada invocación se delega directamente al sistema operativo, elemento por elemento. Java implementa un mecanismo de *buffered streams* que acumula datos de entrada/salida en un área de memoria denominada *buffer*. Entonces se invoca al sistema operativo cuando ya se tiene bastante información en el buffer, mejorando así la eficiencia.

Clases para BufferedStreams

Para trabajar con buffered streams en base a caracteres se usan las siguientes clases:

- `java.io.BufferedReader` para trabajar con entrada de datos de caracteres
- `java.io.BufferedWriter` para trabajar con salida de datos de caracteres

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

public class Main {
  public static void main(String[] args) throws IOException {
    BufferedReader  fin  = null;
    BufferedWriter fout = null;
    int c;
    try {
      fin  = new BufferedReader(new FileReader("hola.txt"));
      fout = new BufferedWriter(new FileWriter("salida.txt"));
      while((c = fin.read()) != -1) {
        fout.write(c);
      }
    } finally {
      if (fin != null)  { fin.close(); }
      if (fout != null) { fout.close(); }
    }
  }}
```

Cómo nos gustaría que funcione la Entrada y Salida

Mundo Ideal

Para no tener que pensar tanto cuando programamos, sería ideal que nuestros programas se escribieran y funcionaran de manera similar para:

- Leer desde el teclado o desde un archivo usando la clase `Scanner`
- Escribir en la pantalla o en un archivo, usando métodos como `println` u otros similares
- Además nos gustaría que esto fuera eficiente, utilizando Buffered Streams!

Leyendo archivos con Scanner

Por suerte para nosotros, la clase Scanner ya está preparada para lo que queremos hacer...

- En vez de ejecutar `new Scanner(System.in)` para leer desde el teclado
- Ejecutamos:

```
Scanner s = new Scanner(
                new BufferedReader(
                    new FileReader("archivo.txt")));
```

Escribiendo archivos con PrintWriter

Para escribir en un archivo de la misma forma que si estuvieramos usando `System.out`, debemos crear un objeto de clase `java.io.PrintWriter`:

```
 PrintWriter writer = new PrintWriter(
                          new BufferedWriter(
                              new FileWriter("archivo_salida.txt")));

writer.println("Hola archivo!");
```

- En la documentación de `PrintWriter` están todos sus métodos, pero se destacan: `print`, `println`, `format`.
- Una desventaja es que los métodos nunca arrojan excepciones. Se debe usar el método `checkError` para ver si hubo problemas en la escritura...

Ejemplo a desarrollar: Lectura datos de alumnos

Enunciado

Considere un archivo de texto que contiene un listado de alumnos con los siguientes datos: nombre, apellido, nombre de asignatura, nota 1, nota 2 y nota 3. Cada campo está separado por un espacio.

Problema

Escriba un programa en Java que lea el archivo con los datos y escriba un archivo "informe.txt" indicando los promedios de nota de cada alumno, indicando a qué asignatura corresponden. Use las clases `Scanner` y `PrintWriter` para leer y escribir los archivos, respectivamente.

Preguntas

Preguntas?