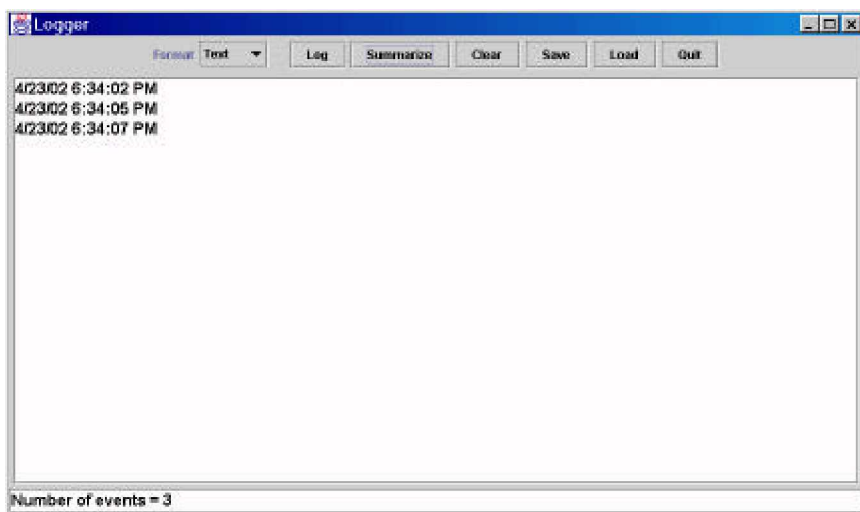


Clase 29

Aprendizaje activo: flujos

La aplicación Logger



Objetivos

- **En el marco de trabajo de la aplicación `Logger`, analizaremos tres formas de leer y escribir datos utilizando flujos de Java:**
 1. como texto
 2. como datos binarios
 3. como objetos serializados
- **Aprenderemos a utilizar la clase incorporada `LinkedList` de Java y su clase de ayuda `ListIterator` para mantener el registro.**

3

Arquitectura de Logger

`Logger.java`:

- proporciona el método `main()` y la GUI
- genera datos de registro ("`Log`") como objetos `Date`
- resume y realiza comprobaciones mínimas de validez ("`Summarize`");
- selecciona archivos para lectura y escritura utilizando un `JFileChooser`
- llama a métodos en `LoggerIO` para leer y escribir datos

Como estudiante, escribirá esos métodos.

4

`java.util.LinkedList`

- `java.util.LinkedList` posee los mismos métodos que nuestra `SLinkedList`, y muchos más.
- `Logger` utiliza una `LinkedList` para contener los datos del registro y los pasa a los métodos `LoggerIO` que tendrá que escribir. Por ejemplo:

```
void saveToText( List list, File f )
```

5

Iteradores

- Un *iterador* es una clase de ayuda diseñada para utilizarse con una lista o con una clase de colección.
- Incluye métodos para devolver los miembros de la colección de uno en uno.
- Los iteradores también implementan métodos que permiten modificar la colección con relación a la posición actual del iterador.

6

Interfaz ListIterator

```
public interface ListIterator {
    public boolean hasNext();
    public Object next()
        throws NoSuchElementException;
    public boolean hasPrevious();
    public Object previous()
        throws NoSuchElementException;
    public void remove()
        throws IllegalStateException;
    public void add( Object o );
    public void set( Object o )
        throws IllegalStateException
    . . . .
}
```

Métodos Iterator

- Los iteradores de Java devuelven un nuevo elemento y avanzan hasta el siguiente con la misma operación `next()`. El `ListIterator` de Java también permite ir hacia atrás (`hasPrevious()` y `previous()`).
- El elemento más reciente devuelto por `next()` es el elemento actual.
- `remove()` eliminará el elemento actual de la colección subyacente. `set()` lo modificará.
- `add()` insertará un nuevo elemento después del actual y antes del elemento que desee que se devuelva en la siguiente llamada a `next()`. Tras una llamada a `add()`, el elemento insertado se convierte en el actual. Una llamada a `next()` devolverá el elemento después del insertado.
- La primera llamada a `next()` debería devolver el primer elemento de la lista.

Cómo utilizar un iterador

```
List myList = new LinkedList();
. . .
ListIterator iter = myList.listIterator();
. . .
while ( iter.hasNext() )
{
    Object o = iter.next();
    . . .
}
```

9

E/S tradicional

El enfoque tradicional utiliza distintos esquemas según el tipo de origen o destino, p.ej.,

- **entrada de teclado**
- **salida de pantalla**
- **archivos**
- **conductos de interproceso**
- ***sockets* de red**

10

E/S de Java

- El enfoque preferido de Java es tratar los datos de entrada y de salida utilizando *flujos* (C++ fue el pionero)
- Java ofrece
 - un conjunto de clases de flujos abstractos que definen las *interfaces* del flujo
 - una jerarquía de implementaciones de flujo
- Java le permite tratar algunos datos de entrada y salida de forma distinta, p.ej. `RandomAccessFile`

11

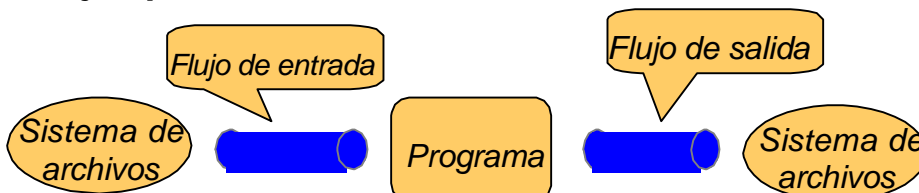
Flujos de entrada y de salida

- Los flujos son *unidireccionales*
- Un *flujo de entrada* controla los datos que entran en el programa y provienen de algún origen
- Un *flujo de salida* controla los datos que salen del programa hacia algún destino
- Si quiere leer y escribir el mismo destino, utilice 2 flujos

12

Flujos y canales E/S

Normalmente, el otro extremo de un flujo conduce a o surge desde un servicio de medios de una plataforma específica, por ejemplo, un sistema de archivos



13

Qué comparten los flujos

- Los flujos de Java son colas FIFO
 - Los flujos entregan información en el orden en que se insertó en el canal subyacente
- Los flujos estándar de Java sólo ofrecen acceso secuencial y no acceso de retroceso, de copia de seguridad ni aleatorio

14

Flujos de acoplamiento

- Los flujos de Java pueden combinarse utilizando un flujo como argumento constructor del otro
- Esto se suele hacer para agregar funcionalidad y para convertir los datos
- Los conductos de flujos se construyen
 - desde el origen de datos hacia el programa o
 - desde el destino de datos de vuelta hacia el programa

15

Conducto de flujo, I



16

Conducto de flujo, II

- Un `FileInputStream` lee bytes desde un archivo
- Un `InputStreamReader` convierte un flujo de entrada en caracteres
- Un `BufferedReader` almacena un flujo de caracteres en el búfer para una mayor eficacia
- Un `StreamTokenizer` analiza un flujo de caracteres en testigos

17

Construcción del conducto

```
FileInputStream f =  
    new FileInputStream( path );  
InputStreamReader i =  
    new InputStreamReader( f );  
BufferedReader b =  
    new BufferedReader( i );  
StreamTokenizer t =  
    new StreamTokenizer( b );
```

18

Ejercicio de Logger

- Descargue los dos archivos `Logger.java` y `LoggerIO.java` del sitio web de clase (enlaces de la clase 29). Guárdelos en un nuevo directorio.
- Cree un nuevo proyecto y monte el directorio en el que acaba de guardar los dos archivos java. No inserte declaraciones de paquetes. Déjelos en el paquete predeterminado.
- Compile el proyecto y pruébelo. Intente guardarlo en un nuevo archivo en modo de texto. ¿Se ha creado un archivo? ¿Contiene algo? (Ábralo con el Bloc de notas).

19

`saveToText ()`

- Ya le hemos facilitado casi todo el método `saveToText ()`; ésta es la idea del patrón:
 - Los bloques *try/catch*
 - Cómo construir el flujo, en este caso un *FileWriter*
 - Cómo iterar descendentemente la lista
 - Cómo cerrar el flujo cuando hemos terminado (al cerrar el flujo se guarda el archivo)
- Busque el método en `FileWriter` para escribir la cadena de fecha y utilícela para obtener `nString` a la salida en `writer`.
- Pruébelo. Vea lo que ha escrito en el Bloc de notas.

20

loadFromText(), 1

- Ahora continuaremos con el método `loadFromText()`. De nuevo, observe el patrón:
 - Construir un *BufferedReader* desde un *FileReader*; necesitaremos un *BufferedReader* para leer las líneas de texto de una en una;
 - Analice la condición *while*; leeremos la siguiente línea y comprobaremos el EOF en una sentencia. Muchos métodos de lectura de flujos de entrada devuelven un valor especial al llegar al final del archivo.
 - Comprender por qué utilizamos bloques *try/catch* anidados y varias cláusulas *catch*.
- Escriba una línea de código para convertir una línea de texto en una fecha utilizando

```
Date parse( String s ) throws ParseException
```

21

loadFromText(), 2

- Escriba una segunda línea de código para agregar `Date` a la lista que devuelve el método.
- Borre la línea (sólo sirve para proporcionar una excepción artificial antes de escribir el verdadero código):

```
if ( false ) throw new ParseException("",0);
```
- Pruebe el código. Escriba los datos de salida, borre el registro y compruebe si puede volver a cargarlo en modo de texto.
- Utilice el Bloc de notas para escribir a mano una nueva entrada de registro utilizando el patrón *date* de las entradas anteriores. Compruebe si puede leer los datos modificados. Hágalo con el comando *Summarize*.

22

saveToData (), 1

- Dentro del objeto `Date` hay un campo de datos `private long`, que representa el número de milisegundos desde las 12:00 am del 1 de enero de 1970.
- `saveToData()` guarda las horas del registro como `longs` y no como `Strings`. Puede recuperar los *long* con el método `long getTime()` de *date*.
- Deberá construir un `DataOutputStream` para poder acceder a los métodos capaces de escribir *longs*. No puede construir uno directamente desde un archivo: primero deberá construir un `FileOutputStream`.

23

saveToData (), 2

- Utilice `saveToText()` como modelo para el resto del método. Recuerde detectar excepciones.
- Ahora Pruébalo. Intente escribir el mismo conjunto de horas de registro como texto y como datos en archivos distintos. Abra los dos en el Bloc de notas y compruebe su longitud (puede obtener un recuento exacto de bytes seleccionando Propiedades con el botón derecho en el Explorador).

24

loadFromData(), 1

- Escriba un bucle `while` para leer *longs* de `DataInputStream` siempre y cuando haya alguno (compruebe la documentación), para convertir cada uno de ellos a un `Date` (hay un constructor `Date(long)`) y para agregar el `Date` a la lista devuelta.
- Preste especial atención a lo que ocurre cuando se queda sin *longs* en el flujo de entrada. ¿Cómo puede salir del bucle *while*?
- Compílelo y Pruébalo.

25

loadFromData(), 2

- Asegúrese de que puede escribir datos de registro en modo de datos, limpiar el registro y leer de nuevo los mismos datos.
- ¿Puede crear datos de registro en este formato con el Bloc de notas?
- ¿Puede leer de nuevo datos escritos en modo de datos como texto? ¿Y al contrario? ¿Por qué?

26

`saveToObject()`, 1

- En este método escribiremos toda la lista de fechas como un único objeto utilizando un `ObjectOutputStream`.
- Consulte la documentación para saber cómo crear un `ObjectOutputStream` y cómo llamar al método `writeObject()`. Esta llamada simple escribirá la lista y todo lo que contiene. Analizaremos su funcionamiento en la siguiente clase.
- Compílelo y pruébelo. Escriba algunos datos de registro en modo de objeto y ábralos en el Bloc de notas. Compruebe la longitud del archivo. ¿Cómo es en comparación con el modo de datos y texto?

27

`loadFromObject()`

- Escriba el contenido del bloque *try/catch* para leer toda la lista del registro utilizando un `ObjectInputStream`. Tendrá que construir el flujo, llamar a `readObject()` y, después, cerrar el flujo.
- Compílelo y pruébelo. ¿Puede volver a leerlo en la lista del registro que creó anteriormente en el modo de objeto?
- ¿Puede leer de nuevo los datos en modo de texto o de datos utilizando el modo de objeto?
- ¿Qué ventajas encuentra al comparar cada modo de datos?

28