TexFiles

1 Introducción

`TexFiles` es un paquete Java que contiene varias clases e interfaces de que permiten manipular ficheros de texto.

2 Licencia y copyright

3 Los ficheros

Los ficheros que componen `TexFiles` son dos:

- `TexFiles.jar` es el código ejecutable, y el único imprescindible. También contiene el código fuente, archivos de extensión `java`. El fichero `TexFiles.bat` crea el fichero `TexFiles.jar`.
- `TexFiles.pdf` es este documento que estás leyendo. Incluye algunas explicaciones a modo de manual, y el código completo, que es la referencia definitiva sobre el comportamiento de `TexFiles`.

4 Las clases e interfaces

El código se compone de las siguientes clases e interfaces:

- `CharFile` que trae ficheros de texto a memoria y viceversa.
- `Sort` que ordena alfabéticamente líneas de texto.
- `Regex` que reemplaza caracteres usando expresiones regulares.
- `FilterW` que proporciona una GUI (graphical user interface) a las clases que implementan la interfaz `Filter`. Es la clase principal del fichero `TexFiles.jar` y, por lo tanto, la ejecución del `jar` se arranca llamando al método `main` de `FilterW`.
- `Filter` que es una interfaz que exige un único método `execute(String, String)`.

5 La máquina

Para ejecutar `TexFiles` es preciso un sistema informático en el que esté instalada una máquina virtual Java (Java Virtual Machine, JVM).

En concreto, `TexFiles` funciona con la JVM incluida en la versión 1.4.2 del entorno de tiempo de ejecución Java (Java Runtime Environment, JRE) de Sun. Esta máquina se puede descargar de su sitio oficial en Internet. Seguramente funcionará con otras versiones, pero no lo sé.

6 El arranque

Para arrancar `TexFiles` hay que llamar a la máquina virtual Java y decirle que ejecute el código contenido en el fichero `TexFiles.jar`. Eso es todo.

En Windows la orden es:

```
java.exe -jar TexFiles.jar
```

7 La ventana principal

Si todo ha ido bien, se abrirá la ventana principal de `TexFiles`. La ventana principal tiene dos partes: una barra de herramientas, y un cuerpo.

La estructura completa de la barra de herramientas es como sigue:

- `Toolbar` (barra de herramientas)
  - `File` (fichero)
    - `Cerrar`
  - `Filters` (filtros)
    - `Copy` (copiar)
    - `Replace` (reemplazar)
    - `Sort` (ordenar)
  - `Help` (ayuda)
    - `About TexFiles` (sobre `TexFiles`)

Para elegir la operación a realizar sobre el fichero de texto hay que pulsar `Filters`. Si dentro de `Filters` se elige `Copy`, que es la opción por defecto al arrancar, entonces se copia el fichero. La orden `Replace` pide dos cadenas, una con el patrón, que puede ser una expresión regular cualquiera, y otra con la cadena que sustituirá cada patrón encontrado en cada una de las líneas del fichero. Por último `Sort` ordena alfabéticamente las líneas del fichero.

El funcionamiento del cuerpo es sencillo. Se escriben los nombres de dos ficheros. Uno es el que se lee (`Input file:`), y el otro en el que se escribe (`Output file:`). Los botones `Search in` y `Search out` arrancan exploradores para buscar los ficheros en el sistema de directorios del ordenador. Por fin, el botón `Execute` ejecuta el filtro cuyo nombre figura arriba del botón sobre el fichero que se lee y escribe el resultado en el que se escribe.

8 Ficheros

8.1 Fichero: CharFile.java

```
1 /**
```

Clase: CharFile

La clase `CharFile` permite traer a memoria (`load`) un fichero de texto, almacenándolo como un array de `String`s de nombre `line`. Este array es accesible (`public`), por lo que puede manipularse de cualquier modo que se quiera. También permite salvar (`store`) en un fichero de texto el contenido, modificado o no, de `line`. Además, incluye métodos que permiten hacer sustituciones múltiples. Si se utiliza por sí misma, la clase `CharFile` copia un fichero de texto pudiendo hacer múltiples sustituciones (véase `main(String[])`).

`@author` © Ramón Casares 2000

`@version` 2000.05.02

```
14 */
15 package TexFiles;
16
17 import java.io.BufferedReader;
18 import java.io.FileReader;
19 import java.io.BufferedWriter;
20 import java.io.FileWriter;
21 import java.util.Vector;
22
23 public class CharFile {
24
25  /**
```

Variable: `filename`

El nombre del fichero a leer. Por defecto es `delete.txt`.

```
28 */
29  String filename = "delete.txt";
30
31  /**
```

Variable: `line`

Un array de `String`s en donde se almacena el contenido del fichero.

```
34 */
35  public String[] line = null;
36
37  /**
```

Constructor: `Charfile()`

Usa los valores por defecto.

```
40 */
41  public CharFile() { }
42
43  /**
```

Constructor: `Charfile(String)`

Construye el objeto y carga `line` con el contenido del fichero cuyo nombre toma como parámetro.

`@param filename` es el nombre del fichero

```
49 */
50  public CharFile(String filename) throws java.io.IOException {
51   this.filename = filename; this.load(); }
52
53  /**
```

Método: `load(String)`

Carga en memoria el contenido del fichero cuyo nombre es el parámetro pasado.

`@param filename` nombre del fichero que se trae a memoria

`@return` el contenido del fichero

`@exception java.io.IOException` si falla la lectura del fichero

```
61 */
62  public String[] load(String filename) throws java.io.IOException {
63   this.filename = filename;
64   return(load());
65  }
66
67  /**
```

Método: `load()`

Carga en memoria el contenido del fichero cuyo nombre guarda la variable `filename`; por defecto `delete.txt`.

`@return` el contenido del fichero

`@exception java.io.IOException` si falla la lectura del fichero

```
74 */
75  public String[] load() throws java.io.IOException {
76   this.line = null;
77   Vector inbuffer = new Vector();
78   BufferedReader in = new BufferedReader(new FileReader(filename));
79   String newline = in.readLine();
80   while (newline != null) {
81    inbuffer.addElement(newline);
82    newline = in.readLine();
83   }
84   in.close();
85   this.line = new String[inbuffer.size()];
86   for (int i=0; i<this.line.length; i++) {
87    this.line[i] = (String)inbuffer.elementAt(i);
88   }
89   return(this.line);
90  }
91
92  /**
```

Método: `store(String)`

Actualiza el valor de la variable `filename` y salva en el fichero `filename` el contenido de la variable `contents`.

`@param filename` es el nombre del fichero

`@exception java.io.IOException` si hay problemas para escribir el fichero

```
99  */
100  public void store(String filename) throws java.io.IOException {
101   this.filename = filename;
102   store();
103  }
104
105  /**
```

Método: `store()`

Salva en el fichero cuyo nombre contiene la variable `filename` el contenido de la variable `contents`. Los otros métodos `store` terminan llamando a éste.

`@exception java.io.IOException` si hay problemas para escribir el fichero

```
112  */
113  public void store() throws java.io.IOException {
114   BufferedWriter out = new BufferedWriter(new FileWriter(filename));
115   for (int i=0; i<line.length; i++) { out.write(line[i]); out.newLine(); }
116   out.close();
117  }
118
119  /**
```

Método: `append(String, String)`

Añade a cada `String` de `line` el `String pre` al comienzo y el `String post` al final.

`@param pre` se añade al comienzo de cada línea

`@param post` se añade al final de cada línea

```
126  */
127  public void append(String pre, String post) {
128   for (int l=0; l<line.length; l++) { line[l] = pre + line[l] + post; }
129  }
130
131  /**
```

Método: `delimite(String)`

Cambia la forma de `line` poniendo los fines de línea en donde encuentra el `String del`.

`@param del` marca dónde se hacen las divisiones de línea

```
137  */
138  public void delimite(String del) {
139   String[] nline = new String[count(del)+1]; int nl = 0;
140   for (int l=0; l<nline.length; l++) { nline[l] = ""; }
141   for (int l=0; l<line.length; l++) { int i = 0;  int j;
```

```
142     while ( i < line[l].length() ) { j = line[l].indexOf(del,i);
143      if ( j == -1 ) {
144       nline[nl] = nline[nl] + line[l].substring(i);
145       i = line[l].length();
146      } else {
147       nline[nl] = nline[nl] + line[l].substring(i,j);
148       nl++;
149       i = j + del.length();
150    }}}
151    line = nline;
152   }
153
154   /**
```

Método: `count(String, String)`

Cuenta el número de veces que el `String so` ocurre en el `String s`.

`@param so` es el `String` a contar

`@param s` es el `String` donde se hace la cuenta

`@return` el número de veces

```
162  */
163   public static int count(String so, String s) {
164    int c = 0;  int i = 0;  int j;
165    while ( i < s.length() ) { j = s.indexOf(so,i);
166     if ( j == -1 ) { i = s.length(); }
167     else { c++; i = j + so.length(); }
168    }
169    return c;
170   }
171
172   /**
```

Método: `count(String)`

Cuenta el número de veces que el `String so` ocurre en `line`.

`@param so` es el `String` a contar

`@return` el número de veces

```
178  */
179   public int count(String so) {
180    int c = 0;
181    for(int l=0; l<line.length; l++) { c = c + count(so,line[l]); }
182    return c;
183   }
184
185   /**
```

Método: `replace(String, String, String)`

Reemplaza en el `String s` cada ocurrencia del `String olds` por el `String news`.

`@param olds` es el `String` a sustituir

`@param news` es el `String` por el que se sustituye

`@param s` es el `String` donde se hace la sustitución

`@return` el resultado de hacer la sustitución

```
194  */
195   public static String replace(String olds, String news, String s) {
196    String t = "";  int i = 0;  int j;
197    while ( i < s.length() ) { j = s.indexOf(olds,i);
198     if ( j == -1 ) { t = t + s.substring(i); i = s.length(); }
199     else { t = t + s.substring(i,j) + news; i = j + olds.length(); }
200    }
201    return t;
202   }
203
204   /**
```

Método: `replace(String[], String[])`

Reemplaza en `line` cada ocurrencia de los `String`s del array `olds[]` por los correspondientes `String`s del array `news[]`.

`@param olds` es el array de `String`s a sustituir

`@param news` es el array con los `String`s que sustituyen

```
212  */
213   public void replace(String[] olds, String[] news) {
214    for ( int l = 0; l < line.length; l++)
215     for ( int j = 0; (j < olds.length) && (j < news.length); j++)
216      line[l] = replace(olds[j],news[j],line[l]);
217   }
218
219   /**
```

Método: `main(String[])`

Los argumentos pueden ser cuatro, tres o dos.

Si se llama con cuatro argumentos, entonces cada uno contiene el nombre de un fichero. El primero contiene en cada línea un `String` cuya aparición ha de ser sustituida por el `String` que ocupa la misma línea en el segundo fichero. El tercer fichero es el que se lee y el cuarto es en el que se escribe el resultado de la sustitución múltiple efectuada.

Así, los ficheros `TeXcode.txt` y `Wincode.txt` permiten traducir ficheros codificados en ASCII, con las convenciones de TEX, en ANSI, que es la codificación utilizada por Windows, y viceversa. Por ejemplo:

```
java TeXFilter TeXcode.txt Wincode.txt TeXfile WinFile
```

hace una réplica del fichero `TeXFile` de nombre `WinFile` pero, por ejemplo, donde aparece `\'a` en `TeXFile` escribe `á` en `WinFile`, mientras que:

```
java TeXFilter Wincode.txt TeXcode.txt Winfile TeXFile
```

hace lo inverso, es decir, replica el fichero `WinFile` en el fichero `TeXFile`, pero, por ejemplo, donde aparece `á` en `WinFile` escribe `\'a` en `TeXFile`.

Se listan los ficheros `TeXcode.txt` y `Wincode.txt`:

TeXcode.txt	**Wincode.txt**
\'a	á
\'A	Á
\'e	é
\'E	É
\'i	í
\'I	Í
\'o	ó
\'O	Ó
\'u	ú
\'U	Ú
\"u	ü
\"U	Ü
\~n	ñ
\~N	Ñ
\tura{a}	ª
\tura{o}	º
?‘	¿
!‘	¡

Si se llama con tres argumentos, entonces el primero es el `String` delimitador, el segundo el archivo a leer y el tercero el archivo a escribir.

Si se llama con dos argumentos, entonces copia le fichero cuyo nombre es el primer argumento en un fichero de nombre el pasado como segundo argumento.

Si no son dos, tres o cuatro, muestra el uso autónomo de esta clase.

`@param args` son los argumentos de la línea de comandos

```
268  */
269   public static void main(String[] args) {
270    if (args.length == 4) {
271     try {
272      CharFile oldf = new CharFile(args[0]);
273      CharFile newf = new CharFile(args[1]);
274      CharFile af = new CharFile(args[2]);
275      af.replace(oldf.line,newf.line);
276      af.store(args[3]);
277     } catch (java.io.IOException e) { System.out.println(e); }
278    } else if (args.length == 3) {
279     try {
280      CharFile in = new CharFile(args[1]);
281      in.delimite(args[0]);
282      in.store(args[2]);
283     } catch (java.io.IOException e) { System.out.println(e); }
```

```
284   } else if (args.length == 2) {
285    try {
286     CharFile in = new CharFile(args[0]);
287     in.store(args[1]);
288    } catch (java.io.IOException e) { System.out.println(e); }
289   } else {
290    System.out.println("Syntax: TeXFilter [ <old> <new> ] <base> <mod>");
291    System.out.println("   <old>:  file with substrings to replace from");
292    System.out.println("   <new>:  file with substrings to replace with");
293    System.out.println("   <base>: file to modify");
294    System.out.println("   <mod>:  modified file, OR");
295    System.out.println("Syntax: TeXFilter <del> <base> <mod>");
296    System.out.println("   <del>:  line delimiter string");
297    System.out.println("   <base>: file to modify");
298    System.out.println("   <mod>:  modified file");
299   }
300  }
301 }
```

8.2 Fichero: Sort.java

```
1 /**
```

Clase: Sort

La clase `Sort` implementa la interfaz `Filter` y usa el algoritmo Quick Sort sobre `String`s. Modifica las reglas normales de manera que 'del a' queda detrás de 'de lo'. Además ignora los caracteres '\' y '"'. De este modo produce el mismo resultado que el programa C correspondiente al generar los índices de "El problema aparente" Ramón Casares: *El problema aparente. Una teoría del conocimiento*, Visor Dis., Madrid, 1999. Isbn: 84-7774-877-2.

Ha sido adaptada de Sun QSortAlgorithm v. 1.6 96/12/06

`@author` © Ramón Casares 2000

`@version` 2000.05.02

```
16 */
17 package TexFiles;
18
19 import java.util.Locale;
20 import java.text.Collator;
21 import java.text.CollationKey;
22 import java.text.RuleBasedCollator;
23
24 public class Sort implements Filter {
25
26  /**
```

Constructor: `Sort()`

```
26 */
27  public Sort() {}
28
29  /**
```

Constructor: `Sort(String, String)`

Crea un objeto `Sort` y ordena el fichero `infile` y lo salva como `outfile`.

`@param infile` el fichero a leer para ordenar

`@param outfile` el fichero ordenado a escribir

```
36  */
37   public Sort(String infile, String outfile)
38    throws java.io.IOException { execute(infile, outfile); }
39
40   /**
```

Método: `execute(String, String)`

Ordena el fichero `infile` y lo salva como `outfile`. Con este método implementa la interfaz `Filter`.

`@param infile` el fichero a leer para ordenar

`@param outfile` el fichero ordenado a escribir

`@exception java.io.IOException` si hay un error al leer o escribir

```
48  */
49   public void execute(String infile, String outfile)
50    throws java.io.IOException {
51    try {
52     CharFile in = new CharFile(infile);
53     RuleBasedCollator esCollator =
54      (RuleBasedCollator) Collator.getInstance(new Locale("es","ES"));
55     String texRules = esCollator.getRules()
56      + "&'_'=' '&'_'='}'" // treat space and } as _
57      + "&\u0000='\\'&\u0000='\"'"; // ignore \ and "
58     RuleBasedCollator texCollator = new RuleBasedCollator(texRules);
59     CollationKey[] keys = new CollationKey[in.line.length];
60     for (int i=0; i<in.line.length; i++)
61      keys[i] = texCollator.getCollationKey(in.line[i]);
62     sort(keys);
63     for (int i=0; i<in.line.length; i++)
64      in.line[i] = keys[i].getSourceString();
65     in.store(outfile);
66    } catch (Exception e) { System.out.println(e); }
67   }
68
69   /**
```

Método: `QuickSort(CollationKey[], int, int)`

Esta es una versión genérica del algoritmo Quicksort de C.A.R HoareC.A.R. Hoare: *Quicksort* (1962). Reimpreso en *Great Papers in Computer Science*, editado por Ph. Laplante, IEEE Press, Piscataway NJ & West, St. Paul MN; 1996. ISBN: 0-7803-1112-4. Es capaz de tratar `arrays` ya ordenados y `arrays` con claves duplicadas. Usa el método auxiliar `swap`.

N: 0-7803-1112-4. .

`@param a` el array a ordenar

`@param lo0` límite inferior

`@param hi0` límite superior

`@exception Exception` si algo va mal

```
84  */
85   static void QuickSort(CollationKey a[], int lo0, int hi0)
86    throws Exception {
87
88    int lo = lo0;
89    int hi = hi0;
90    CollationKey mid;
91
92    if ( hi0 > lo0) {
93     mid = a[ ( lo0 + hi0 ) / 2 ];
94     while( lo <= hi ) {
95      while( ( lo < hi0 ) && ( a[lo].compareTo(mid) < 0 ) ) ++lo;
96      while( ( hi > lo0 ) && ( a[hi].compareTo(mid) > 0 ) ) --hi;
97      if( lo <= hi ) swap(a, lo++, hi--);
98     }
99     if( lo0 < hi ) QuickSort( a, lo0, hi );
100    if( lo < hi0 ) QuickSort( a, lo, hi0 );
101   }
102  }
103
104  private static void swap(CollationKey a[], int i, int j)
105   {  CollationKey T; T = a[i]; a[i] = a[j]; a[j] = T;  }
106
107
108  /**
```

Método: `sort(CollationKey[])`

Llama al algoritmo Quick Sort inicializando los parámetros.

`@param a` es el array a ordenar

```
113 */
114  public static void sort(CollationKey[] a) throws Exception
115   {  QuickSort(a, 0, a.length - 1);  }
116
117  /**
```

Método: `main(String[])`

La presente implementación del método principal (`main`) permite hacer dos usos directos de la clase `Sort`. Si se le llama sin argumentos, por ejemplo pinchando el fichero `Sort.class`, entonces se ejecuta dentro de la ventana gráfica proporcionada por la clase `FilterW`. Si se le llama con dos argumentos, entonces ordena las líneas del fichero cuyo nombre figura como primer argumento y deja el resultado en un fichero cuyo nombre es el segundo argumento. En cualquier otro caso muestra la sintaxis de llamada de esta clase.

`@param args` son los argumentos de la línea de comandos

```
130 */
131  public static void main(String[] args) {
132   if (args.length == 2) {
133    try { new Sort(args[0],args[1]); }
134    catch (java.io.IOException e) { System.out.println(e); }
135   } else if (args.length == 0) {
136    new FilterW("Sort",new Sort(),"","");
137   } else {
138    System.out.println("Syntax: Sort [<input filename> <output filename>]");
139   }
140  }
141
142 }
```

8.3 Fichero: Regex.java

```
1 /**
```

Clase: Regex

Implements a regular expressions processor.

@author © Ramón Casares 2003

@version 2003.11.03

```
7 */
8 package TexFiles;
9
10 import java.util.regex.Pattern;
11 import java.util.regex.Matcher;
12 import javax.swing.JOptionPane;
13
14 public class Regex implements Filter {
15
16  /**
```

Variable: p is the current pattern

```
16 */
17  public Pattern p;
18  public String sp;
19
20  /**
```

Variable: r is the replacement

```
20 */
21  public String r;
22
23  /**
```

Método: newPattern(String)

```
23 */
24  public Pattern newPattern(String regex) {
25   p = Pattern.compile(regex);
26   sp = new String(regex);
27   return(p);
28  }
```

```
29
30  public Pattern newPattern() {
31   String regex = JOptionPane.showInputDialog(null,"New Pattern:",sp);
32   if( regex != null ) {
33    p = Pattern.compile(regex);
34    sp = new String(regex);
35   }
36   return(p);
37  }
38
39  /**
```

Método: newReplacement(String)

```
39 */
40  public String newReplacement(String rep) {
41   r = new String(rep);
42   return(r);
43  }
44
45  public String newReplacement() {
46   String rep = JOptionPane.showInputDialog(null,"New Replacement for "+sp,r);
47   if( rep != null ) r = new String(rep);
48   return(r);
49  }
50
51
52  /**
```

Método: newPattern(String, String)

```
52 */
53  public Pattern newPattern(String regex, String rep) {
54   p = Pattern.compile(regex);
55   sp = new String(regex);
56   r = new String(rep);
57   return(p);
58  }
59
60  /**
```

Método: verboseExecute(String)

```
60 */
61  public String verboseExecute(String text) {
62   if( p == null || text == null ) return("");
63   Matcher m = p.matcher(text);
64   StringBuffer sb = new StringBuffer();
65   while (m.find()) m.appendReplacement(sb, r);
66   m.appendTail(sb);
67   return( sb.toString() );
68  }
69
70  public String execute(String text) {
71   return( text.replaceAll(sp,r) );
72  }
73
74  public void execute(String filein, String fileout)
```

```
75    throws java.io.IOException {
76    CharFile cf = new CharFile(filein);
77    for(int i=0; i<cf.line.length; i++)
78     if(cf.line[i] != null)
79      cf.line[i] = verboseExecute(cf.line[i]);
80    cf.store(fileout);
81   }
82
83
84  }
```

8.4 Fichero: Filter.java

```
1  /**
```

Interfaz: `Filter`

Un filtro es cualquier clase que implementa el método `execute`.

`@author` © Ramón Casares 2000

`@version` 2000.05.02

```
7  */
8  package TexFiles;
9
10 public interface Filter {
11  /**
```

Método: `execute(String, String)`

El método `execute` simplemente lee los datos de un fichero y escribe datos en otro fichero.

`@param infile` el fichero del que se leen los datos

`@param outfile` el fichero en el que se escriben los datos

`@exception java.io.IOException` si falla la lectura o la escritura

```
19 */
20  public void execute(String infile, String outfile)
21   throws java.io.IOException;
22 }
```

8.5 Fichero: FilterW.java

```
1  /**
```

Clase: `FilterW`

Proporciona una GUI (Interfaz Gráfica de Usuario) a las clases que implementan la interfaz `Filter` o a aquellos programas externos que aceptan la sintaxis `progname infile outfile`.

Por su parte, `FilterW` implementa las interfaces `ActionListener` y `WindowListener`.

Al arrancar `FilterW` aparece la ventana principal `w`. Para seleccionar el fichero desde el que se leen los datos se puede pulsar el botón `Search in`, y entonces aparece la ventana de entrada `wi`. Del mismo modo, pero apretando el botón `Search out`, se puede

seleccionar el fichero en el que se escribirá el resultado de aplicar el filtro. Las acciones que resultan de pulsar los botones de la ventana principal `w` están especificadas en el método `actionPerformed(ActionEvent)`.

`@author` © Ramón Casares 2000

`@version` 2000.05.02

`@see` `Filter` (en la página 15)

```
22 */
23 package TexFiles;
24
25 import java.awt.*;
26 import java.awt.event.*;
27
28 public class FilterW
29  implements ActionListener, WindowListener {
30
31  /**
```

Variable: `w` (ventana principal)

```
31 */
32  private Window w;
33  /**
```

Variable: `wi` (ventana de entrada)

```
33 */
34  private Window wi;
35  /**
```

Variable: `wo` (ventana de salida)

```
35 */
36  private Window wo;
37
38  /**
```

Variable: `namein` (contiene el nombre del fichero de entrada)

```
38 */
39  private TextField namein;
40  /**
```

Variable: `nameout` (contiene el nombre del fichero de salida)

```
40 */
41  private TextField nameout;
42
43  /**
```

Variable: `activefilter` (contiene el valor del filtro activo)

```
43 */
44  private Label activefilter;
45
46  /**
```

Variable: `filter` (objeto que implementa la interfaz `Filter`)

```
46 */
47  private Filter filter = null; // null si usa un programa externo
48
49  private Regex re = new Regex();
50
51  /**
```

Variable: `progname` (nombre del programa externo)

```
51 */
52  private String progname = "xcopy /i ";
53
54  /**
```

Constructor: `FilterW()`

```
54 */
55  public FilterW() {}
56
57  /**
```

Constructor: `FilterW(String, Filter, String, String)`

Crea un objeto `FilterW` que trabaja sobre un objeto Java que implementa la interfaz `Filter`.

`@param title` título que aparece en la ventana

`@param filter` objeto Java que determina la acción a ejecutar

`@param infile` fichero del que se lee

`@param outfile` fichero en el que se escribe

`@see Filter` (en la página 15)

```
67 */
68  public FilterW(String title, Filter filter,
69   String infile, String outfile) {
70   this.filter = filter;
71   this.progname = title;
72   openW(title, infile, outfile);
73  }
74
75  /**
```

Constructor: `FilterW(String, String, String, String)`

Crea un objeto `FilterW` que trabaja sobre un programa externo que acepta la sintaxis `progname infile outfile`.

`@param title` título que aparece en la ventana

`@param progname` programa externo que determina la acción a ejecutar

`@param infile` fichero del que se lee

`@param outfile` fichero en el que se escribe

```
84 */
85  public FilterW(String title, String progname,
86   String infile, String outfile) {
87   this.filter = null;
88   this.progname = progname;
89   openW(title, infile, outfile);
90  }
91
92  /**
```

Método: `openW(String, String, String)`

Crea la ventana principal, `w`. Es un panel de dos filas, 1) una para el fichero de entrada y 2) otra para el de salida, y cuatro columnas, 1) una etiqueta que explica de que fichero se trata, entrada o salida, 2) un cuadro para introducir el nombre del fichero, 3) un botón de búsqueda, que crea una ventana de búsqueda de ficheros, y 4) un botón para ejecutar el filtro o cancelarlo todo.

`@param title` título que aparece en la ventana

`@param infile` fichero del que se lee

`@param outfile` fichero en el que se escribe

```
107 */
108  public void openW(String title, String infile, String outfile) {
109   Frame f = new Frame(title); w = f;
110   f.setBackground(Color.lightGray);
111   // f.setForeground(Color.black);
112
113   MenuBar bar = new MenuBar(); ////////////////////////////////
114    Menu mFile = new Menu("File");
115     MenuItem miClose = new MenuItem("Close");
116      miClose.addActionListener(this);
117      mFile.add(miClose);
118    bar.add(mFile);
119
120    Menu mFilter = new Menu("Filters");
121     MenuItem miCopy = new MenuItem("Copy");
122      miCopy.addActionListener(this);
123      mFilter.add(miCopy);
124     MenuItem miReplace = new MenuItem("Replace");
125      miReplace.addActionListener(this);
126      mFilter.add(miReplace);
127     MenuItem miSort = new MenuItem("Sort");
128      miSort.addActionListener(this);
129      mFilter.add(miSort);
130    bar.add(mFilter);
131
132    Menu mHelp = new Menu("Help");
133     MenuItem miAbout = new MenuItem("About TexFiles");
134      miAbout.addActionListener(this);
135      mHelp.add(miAbout);
136    bar.add(mHelp);
137
138   f.setMenuBar(bar);
```

```
139
140   ////////////////////////////////////////////////////////////
141
142   GridBagLayout gridbag = new GridBagLayout();
143   GridBagConstraints c = new GridBagConstraints();
144   // setFont(new Font("Helvetica", Font.PLAIN, 14));
145   f.setLayout(gridbag);
146
147   Label textin = new Label("Input file:",Label.RIGHT);
148   namein = new TextField(infile,25);
149   Label textout = new Label("Output file:",Label.RIGHT);
150   nameout = new TextField(outfile,25);
151
152   Button searchin = new Button("Search in");
153   Button searchout = new Button("Search out");
154
155   //Button cancel = new Button("Cancel");
156   activefilter = new Label("Copy",Label.CENTER);
157   Button execute = new Button("Execute");
158
159   searchin.addActionListener(this);
160   execute.addActionListener(this);
161   searchout.addActionListener(this);
162   //cancel.addActionListener(this);
163
164   c.fill = GridBagConstraints.BOTH; // for all
165   c.gridheight = 1; c.weighty = 0.0; // for all
166   c.insets = new Insets(5,5,5,5);
167   c.gridwidth = 1; c.weightx = 0.0; // not expandable
168   gridbag.setConstraints(textin, c); f.add(textin);
169   c.gridwidth = 3; c.weightx = 1.0; // expandable
170   gridbag.setConstraints(namein, c); f.add(namein);
171   c.gridwidth = GridBagConstraints.RELATIVE; //last but one
172   c.weightx = 0.0; // not expandable
173   gridbag.setConstraints(searchin, c); f.add(searchin);
174   c.gridwidth = GridBagConstraints.REMAINDER; //end row
175   //gridbag.setConstraints(cancel, c); f.add(cancel);
176   gridbag.setConstraints(activefilter, c); f.add(activefilter);
177   c.gridwidth = 1;
178   gridbag.setConstraints(textout, c); f.add(textout);
179   c.gridwidth = 3; c.weightx = 1.0; // expandable
180   gridbag.setConstraints(nameout, c); f.add(nameout);
181   c.gridwidth = GridBagConstraints.RELATIVE; //last but one
182   c.gridwidth = 1; c.weightx = 0.0; // not expandable
183   gridbag.setConstraints(searchout, c); f.add(searchout);
184   c.gridwidth = GridBagConstraints.REMAINDER; //end row
185   gridbag.setConstraints(execute, c); f.add(execute);
186
187   f.pack(); f.setSize(f.getPreferredSize()); f.show();
188
189   f.addNotify();
190   f.addWindowListener(this);
191
192   }
193
194  /**
```

Método: takeinputname()

Crea la ventana de entrada para seleccionar el fichero de entrada, wi. El fichero elegido, cuyo nombre completo (con path) devuelve este método, es el que será leído.

La ventana wi tiene vida fuera de este método, y así puede ser vista por el WindowListener.

@return el nombre del fichero elegido para leer

```
205 */
206  private String takeinputname() {
207   Frame fi = new Frame("Input file"); wi = fi;
208   FileDialog fdi = new FileDialog(fi,"Input name",FileDialog.LOAD);
209   fdi.pack(); fdi.show(); fdi.addNotify();
210   fdi.addWindowListener(this);
211   return(fdi.getDirectory() + fdi.getFile());
212  }
213
214  /**
```

Método: takeoutputname()

Crea la ventana de salida para seleccionar el fichero de salida, wo. El fichero elegido, cuyo nombre completo (con path) devuelve este método, es el que será escrito.

La ventana wo tiene vida fuera de este método, y así puede ser vista por el WindowListener.

@return el nombre del fichero elegido para leer

```
225 */
226  private String takeoutputname() {
227   Frame fo = new Frame("Output file"); wo = fo;
228   FileDialog fdo = new FileDialog(fo,"Output name",FileDialog.SAVE);
229   fdo.pack(); fdo.show(); fdo.addNotify();
230   fdo.addWindowListener(this);
231   return(fdo.getDirectory() + fdo.getFile());
232  }
233
234  /**
```

Método: actionPerformed(ActionEvent)

Este es el único método definido en la interfaz ActionListener, que así queda implementada. La implementación determina que acción se ejecuta al pulsar cada uno de los botones definidos en la ventana principal, w.

@param e es la acción ejecutada

```
243 */
244  public void actionPerformed(ActionEvent e) {
245   String texto = e.getActionCommand();
246   if ("Close".equals(texto)) System.exit(0);
247   if ("Cancel".equals(texto)) System.exit(0);
248   if ("Execute".equals(texto)) {
249    try {
250     if (filter==null) {
251      String commandstring = progname+" \""+
```

```
252        namein.getText() + "\" \"" + nameout.getText() + "\"";
253       Runtime.getRuntime().exec(commandstring);
254       System.out.println("Executed: " + commandstring);
255      } else {
256       filter.execute(namein.getText(),nameout.getText());
257       System.out.println("Executed: " + progname +
258        "(" + namein.getText() + "," + nameout.getText() + ");");
259      }
260     } catch (java.io.IOException ioe) { System.out.println(ioe); }
261    }
262    if ("Copy".equals(texto)) {
263     filter = null;
264     progname = "xcopy /i ";
265     activefilter.setText("Copy");
266    }
267    if ("Sort".equals(texto)) {
268     filter = new Sort();
269     progname = "Sort";
270     activefilter.setText("Sort");
271    }
272    if ("Replace".equals(texto)) {
273     re.newPattern();
274     re.newReplacement();
275     filter = re;
276     progname = "Replace";
277     activefilter.setText("Replace");
278    }
279    if ("Search in".equals(texto)) {
280     String niaux = this.takeinputname();
281     if (!"nullnull".equals(niaux)) namein.setText(niaux);
282     w.show(); }
283    if ("Search out".equals(texto)) {
284     String noaux = this.takeoutputname();
285     if (!"nullnull".equals(noaux)) nameout.setText(noaux);
286     w.show(); }
287    if ("About TexFiles".equals(texto)) {
288     String[] message = new String[3];
289     message[0] = "TexFiles version 1.0";
290     message[1] = "ⓒ 2004 Ramón Casares";
291     message[2] = "r.casares@ieee.org";
292     javax.swing.JOptionPane.showMessageDialog(null,message,
293      "About TeXFiles",javax.swing.JOptionPane.INFORMATION_MESSAGE);
294    }
295   }
296
297   /**
```

Método: `windowClosing(WindowEvent)`

El único método no vacío de la interfaz `WindowListener`. Debe distinguir en cuál de las ventanas se ejecuta la acción. Al cerrarse la ventana principal, `w`, se termina el programa.

La implementación de los otros métodos de la interfaz `WindowListener` está vacía.

`@param e` es la acción ejecutada en la ventana

```
308 */
309  public void windowClosing(WindowEvent e) {
310   if(e.getWindow() == w) { w.dispose(); System.exit(0);}
311   if(e.getWindow() == wi) { wi.dispose(); }
312   if(e.getWindow() == wo) { wo.dispose(); }
313  }
314  public void windowOpened(WindowEvent e) {}
315  public void windowClosed(WindowEvent e) {}
316  public void windowIconified(WindowEvent e) {}
317  public void windowDeiconified(WindowEvent e) {}
318  public void windowActivated(WindowEvent e) {}
319  public void windowDeactivated(WindowEvent e) {}
320
321  /**
```

Método: `main(String[])`

Ejecuta la ventana gráfica tomando los nombres de los ficheros, si los hay, de la línea de comandos. Si hay uno, supone que es el del fichero a leer. Si hay dos, el primero es el de lectura, el segundo el de escritura. Si son más, se ignoran del tercero en adelante.

Como usa el constructor nulo, `FilterW()`, se aplica el programa externo definido por defecto, `xcopy /i`, lo que quiere decir que, si el sistema operativo es DOS, copia el fichero de entrada en el de salida.

`@param args` los argumentos de la línea de comandos

```
335 */
336  public static void main(String[] args) {
337   FilterW fw = new FilterW();
338   if (args.length == 0) fw.openW("Filter","","");
339   if (args.length == 1) fw.openW("Filter",args[0],"");
340   if (args.length  > 1) fw.openW("Filter",args[0],args[1]);
341   }
342
343 }
```

8.6 Fichero: TexFiles.bat

```
1  cd ..
2  javac TexFiles/CharFile.java
3  javac TexFiles/Filter.java
4  javac TexFiles/FilterW.java
5  javac TexFiles/Sort.java
6  javac TexFiles/Regex.java
7  jar cf TexFiles/TexFiles.jar TexFiles/CharFile.class TexFiles/CharFile.java
8  jar uf TexFiles/TexFiles.jar TexFiles/Filter.class TexFiles/Filter.java
9  jar uf TexFiles/TexFiles.jar TexFiles/FilterW.class TexFiles/FilterW.java
10 jar uf TexFiles/TexFiles.jar TexFiles/Sort.class TexFiles/Sort.java
11 jar uf TexFiles/TexFiles.jar TexFiles/Regex.class TexFiles/Regex.java
12 echo jar uf TexFiles/TexFiles.jar TexFiles/TexFiles.tex
13 echo jar uf TexFiles/TexFiles.jar TexFiles/TexFiles.pdf
14 jar uf TexFiles/TexFiles.jar TexFiles/TexFiles.bat
15 echo Main-Class: TexFiles/FilterW> TexFiles.MF
16 echo Class-Path: .\ TexFiles.jar>> TexFiles.MF
```

```
17 jar umf TexFiles.MF TexFiles/TexFiles.jar
18 del TexFiles.MF
```

Índice Java

Índice