Aunque la interface ***Map*** no deriva de ***Collection***, es posible ver los ***Maps*** como colecciones de ***claves***, de ***valores*** o de parejas ***clave/valor***. A continuación se muestran los métodos de la interface ***Map*** (comando **> *javap java.util.Map***):

```
Compiled from Map.java
public interface java.util.Map
{
    public abstract void clear();
    public abstract boolean containsKey(java.lang.Object);
    public abstract boolean containsValue(java.lang.Object);
    public abstract java.util.Set entrySet();
    public abstract boolean equals(java.lang.Object);
    public abstract java.lang.Object get(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Set keySet();
    public abstract java.lang.Object put(java.lang.Object, java.lang.Object);
    public abstract void putAll(java.util.Map);
    public abstract java.lang.Object remove(java.lang.Object);
    public abstract int size();
    public abstract java.util.Collection values();
    public static interface java.util.Map.Entry
    {
        public abstract boolean equals(java.lang.Object);
        public abstract java.lang.Object getKey();
        public abstract java.lang.Object getValue();
        public abstract int hashCode();
        public abstract java.lang.Object setValue(java.lang.Object);
    }
}
```

Muchos de estos métodos tienen un significado evidente, pero otros no tanto. El método ***entrySet()*** devuelve una "vista" del ***Map*** como ***Set***. Los elementos de este ***Set*** son referencias de la interface ***Map.Entry***, que es una ***interface interna*** de ***Map***. Esta "vista" del ***Map*** como ***Set*** permite modificar y eliminar elementos del ***Map***, pero no añadir nuevos elementos.

El método ***get(key)*** permite obtener el valor a partir de la clave. El método ***keySet()*** devuelve una "vista" de las claves como ***Set***. El método ***values()*** devuelve una "vista" de los valores del ***Ma***p como ***Collection*** (porque puede haber elementos repetidos). El método ***put()*** permite añadir una pareja clave/valor, mientras que ***putAll()*** vuelca todos los elementos de un ***Map*** en otro ***Map*** (los pares con clave nueva se añaden; en los pares con clave ya existente los valores nuevos sustituyen a los antiguos). El método ***remove()*** elimina una pareja clave/valor a partir de la clave.

La interface ***SortedMap*** añade los siguientes métodos, similares a los de ***SortedSet***:

```
Compiled from SortedMap.java
public interface java.util.SortedMap extends java.util.Map
{
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object firstKey();
    public abstract java.util.SortedMap headMap(java.lang.Object);
    public abstract java.lang.Object lastKey();
    public abstract java.util.SortedMap subMap(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedMap tailMap(java.lang.Object);
}
```

La clase ***HashMap*** implementa la interface ***Map*** y está basada en una hash table, mientras que ***TreeMap*** implementa ***SortedMap*** y está basada en un árbol binario.

4.5.4.8 Algoritmos y otras características especiales: Clases Collections y Arrays

La clase ***Collections*** (no confundir con la interface ***Collection***, en singular) es una clase que define un buen número de métodos static con diversas finalidades. No se detallan o enumeran aquí porque exceden del espacio disponible. Los más interesantes son los siguientes:

- Métodos que definen algoritmos:

  Ordenación mediante el método mergesort
  ```
  public static void sort(java.util.List);
  public static void sort(java.util.List, java.util.Comparator);
  ```
  Eliminación del orden de modo aleatorio
  ```
  public static void shuffle(java.util.List);
  public static void shuffle(java.util.List, java.util.Random);
  ```
  Inversión del orden establecido
  ```
  public static void reverse(java.util.List);
  ```
  Búsqueda en una lista
  ```
  public static int binarySearch(java.util.List, java.lang.Object);
  public static int binarySearch(java.util.List, java.lang.Object,
                                 java.util.Comparator);
  ```
  Copiar una lista o reemplazar todos los elementos con el elemento especificado
  ```
  public static void copy(java.util.List, java.util.List);
  public static void fill(java.util.List, java.lang.Object);
  ```
  Cálculo de máximos y mínimos
  ```
  public static java.lang.Object max(java.util.Collection);
  public static java.lang.Object max(java.util.Collection, java.util.Comparator);
  public static java.lang.Object min(java.util.Collection);
  public static java.lang.Object min(java.util.Collection, java.util.Comparator);
  ```
- Métodos de utilidad

  Set inmutable de un único elemento
  ```
  public static java.util.Set singleton(java.lang.Object);
  ```
  Lista inmutable con n copias de un objeto
  ```
  public static java.util.List nCopies(int, java.lang.Object);
  ```
  Constantes para representar el conjunto y la lista vacía
  ```
  public static final java.util.Set EMPTY_SET;
  public static final java.util.List EMPTY_LIST;
  ```

Además, la clase ***Collections*** dispone de dos conjuntos de métodos "factory" que pueden ser utilizados para convertir objetos de distintas colecciones en objetos "***read only***" y para convertir distintas colecciones en objetos "***synchronized***" (por defecto las clases vistas anteriormente no están sincronizadas), lo cual quiere decir que se puede acceder a la colección desde distintas ***threads*** sin que se produzcan problemas. Los métodos correspondientes son los siguientes:

```
public static java.util.Collection synchronizedCollection(java.util.Collection);
public static java.util.List synchronizedList(java.util.List);
public static java.util.Map synchronizedMap(java.util.Map);
public static java.util.Set synchronizedSet(java.util.Set);
public static java.util.SortedMap synchronizedSortedMap(java.util.SortedMap);
public static java.util.SortedSet synchronizedSortedSet(java.util.SortedSet);

public static java.util.Collection unmodifiableCollection(java.util.Collection);
public static java.util.List unmodifiableList(java.util.List);
public static java.util.Map unmodifiableMap(java.util.Map);
public static java.util.Set unmodifiableSet(java.util.Set);
public static java.util.SortedMap unmodifiableSortedMap(java.util.SortedMap);
public static java.util.SortedSet unmodifiableSortedSet(java.util.SortedSet);
```

Estos métodos se utilizan de una forma muy sencilla: se les pasa como argumento una referencia a un objeto que no cumple la característica deseada y se obtiene como valor de retorno una referencia a un objeto que sí la cumple.

4.5.4.9 Desarrollo de clases por el usuario: clases abstract

Las clases ***abstract*** indicadas en la Figura 4.2, en la página 69, pueden servir como base para que los programadores, con necesidades no cubiertas por las clases vistas anteriormente, desarrollen sus propias clases.

4.5.4.10 Interfaces Cloneable y Serializable

Las clases ***HashSet***, ***TreeSet***, ***ArrayList***, ***LinkedList***, ***HashMap*** y ***TreeMap*** (al igual que ***Vector*** y ***Hashtable***) implementan las interfaces ***Cloneable*** y ***Serializable***, lo cual quiere decir que es correcto sacar copias bit a bit de sus objetos con el método ***Object.clone()***, y que se pueden convertir en cadenas o flujos (***streams***) de caracteres.

Una de las ventajas de implementar la interface ***Serializable*** es que los objetos de estas clases pueden ser impresos con los métodos ***System.Out.print()*** y ***System.Out.println()***.

4.6 OTRAS CLASES DEL PACKAGE JAVA.UTIL

El package ***java.util*** tiene otras clases interesantes para aplicaciones de distinto tipo, entre ellas algunas destinadas a considerar todo lo relacionado con fechas y horas. A continuación se consideran algunas de dichas clases.

4.6.1 Clase Date

La clase ***Date*** representa un instante de tiempo dado con precisión de milisegundos. La información sobre fecha y hora se almacena en un entero ***long*** de 64 bits, que contiene los milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970 GMT (*Greenwich mean time*). Ya se verá que otras clases permiten a partir de un objeto ***Date*** obtener información del año, mes, día, hora, minuto y segundo. A continuación se muestran los métodos de la clase ***Date***, habiéndose eliminado los métodos declarados obsoletos (*deprecated*) en el JDK 1.2:

```
Compiled from Date.java
public class java.util.Date extends java.lang.Object implements
             java.io.Serializable, java.lang.Cloneable, java.lang.Comparable {
    public java.util.Date();
    public java.util.Date(long);
    public boolean after(java.util.Date);
    public boolean before(java.util.Date);
    public java.lang.Object clone();
    public int compareTo(java.lang.Object);
    public int compareTo(java.util.Date);
    public boolean equals(java.lang.Object);
    public long getTime();
    public int hashCode();
    public void setTime(long);
    public java.lang.String toString();
}
```

El constructor por defecto ***Date()*** crea un objeto a partir de la fecha y hora actual del ordenador. El segundo constructor crea el objeto a partir de los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT. Los métodos ***after()*** y ***before()*** permiten saber si la fecha indicada como argumento implícito (***this***) es posterior o anterior a la pasada como argumento explícito. Los métodos ***getTime()*** y ***setTime()*** permiten obtener o establecer los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT para un determinado objeto ***Date***. Otros métodos son consecuencia de las interfaces implementadas por la clase ***Date***.

Los objetos de esta clase no se utilizan mucho directamente, sino que se utilizan en combinación con las clases que se vana ver a continuación.

4.6.2 Clases Calendar y GregorianCalendar

La clase ***Calendar*** es una clase ***abstract*** que dispone de métodos para convertir objetos de la clase ***Date*** en enteros que representan fechas y horas concretas. La clase ***GregorianCalendar*** es la única clase que deriva de ***Calendar*** y es la que se utilizará de ordinario.

Java tiene una forma un poco particular para representar las fechas y horas:

1. Las horas se representan por enteros de 0 a 23 (la hora "0" va de las 00:00:00 hasta la 1:00:00), y los minutos y segundos por enteros entre 0 y 59.
2. Los días del mes se representan por enteros entre 1 y 31 (lógico).
3. Los meses del año se representan mediante enteros de 0 a 11 (no tan lógico).
4. Los años se representan mediante enteros de cuatro dígitos. Si se representan con dos dígitos, se resta 1900. Por ejemplo, con dos dígitos el año 2000 es para Java el año 00.

La clase ***Calendar*** tiene una serie de variables miembro y constantes (variables ***final***) que pueden resultar muy útiles:

- La variable ***int*** AM_PM puede tomar dos valores: las constantes enteras AM y PM.
- La variable ***int*** DAY_OF_WEEK puede tomar los valores ***int*** SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY y SATURDAY.
- La variable ***int*** MONTH puede tomar los valores ***int*** JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER. Para hacer los programas más legibles es preferible utilizar estas constantes simbólicas que los correspondientes números del 0 al 11.
- La variable miembro HOUR se utiliza en los métodos ***get()*** y ***set()*** para indicar la hora de la mañana o de la tarde (en relojes de 12 horas, de 0 a 11). La variable HOUR_OF_DAY sirve para indicar la hora del día en relojes de 24 horas (de 0 a 23).
- Las variables DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_MONTH (o bien DATE), DAY_OF_YEAR, WEEK_OF_MONTH, WEEK_OF_YEAR tienen un significado evidente.
- Las variables ERA, YEAR, MONTH, HOUR, MINUTE, SECOND, MILLISECOND tienen también un significado evidente.
- Las variables ZONE_OFFSET y DST_OFFSET indican la zona horaria y el desfase en milisegundos respecto a la zona GMT.

La clase ***Calendar*** dispone de un gran número de métodos para establecer u obtener los distintos valores de la fecha y/u hora. Algunos de ellos se muestran a continuación. Para más información, se recomienda utilizar la documentación de JDK 1.2.

```
Compiled from Calendar.java
public abstract class java.util.Calendar extends java.lang.Object implements
java.io.Serializable, java.lang.Cloneable {
    protected long time;
    protected boolean isTimeSet;
    protected java.util.Calendar();
    protected java.util.Calendar(java.util.TimeZone,java.util.Locale);
    public abstract void add(int, int);
    public boolean after(java.lang.Object);
    public boolean before(java.lang.Object);
    public final void clear();
    public final void clear(int);
    protected abstract void computeTime();
```

```
    public boolean equals(java.lang.Object);
    public final int get(int);
    public int getFirstDayOfWeek();
    public static synchronized java.util.Calendar getInstance();
    public static synchronized java.util.Calendar getInstance(java.util.Locale);
    public static synchronized java.util.Calendar getInstance(java.util.TimeZone);
    public static synchronized java.util.Calendar getInstance(java.util.TimeZone,
                                                               java.util.Locale);
    public final java.util.Date getTime();
    protected long getTimeInMillis();
    public java.util.TimeZone getTimeZone();
    public final boolean isSet(int);
    public void roll(int, int);
    public abstract void roll(int, boolean);
    public final void set(int, int);
    public final void set(int, int, int);
    public final void set(int, int, int, int, int);
    public final void set(int, int, int, int, int, int);
    public final void setTime(java.util.Date);
    public void setFirstDayOfWeek(int);
    protected void setTimeInMillis(long);
    public void setTimeZone(java.util.TimeZone);
    public java.lang.String toString();
}
```

La clase ***GregorianCalendar*** añade las constante BC y AD para la ERA, que representan respectivamente antes y después de Jesucristo. Añade además varios constructores que admiten como argumentos la información correspondiente a la fecha/hora y –opcionalmente– la zona horaria.

A continuación se muestra un ejemplo de utilización de estas clases. Se sugiere al lector que cree y ejecute el siguiente programa, observando los resultados impresos en la consola.

```
import java.util.*;

public class PruebaFechas {
    public static void main(String arg[]) {
        Date d = new Date();
        GregorianCalendar gc = new GregorianCalendar();
        gc.setTime(d);
        System.out.println("Era:             "+gc.get(Calendar.ERA));
        System.out.println("Year:            "+gc.get(Calendar.YEAR));
        System.out.println("Month:           "+gc.get(Calendar.MONTH));
        System.out.println("Dia del mes:     "+gc.get(Calendar.DAY_OF_MONTH));
        System.out.println("D de la S en mes:"
                           +gc.get(Calendar.DAY_OF_WEEK_IN_MONTH));
        System.out.println("No de semana:    "+gc.get(Calendar.WEEK_OF_YEAR));
        System.out.println("Semana del mes:  "+gc.get(Calendar.WEEK_OF_MONTH));
        System.out.println("Fecha:           "+gc.get(Calendar.DATE));
        System.out.println("Hora:            "+gc.get(Calendar.HOUR));
        System.out.println("Tiempo del dia:  "+gc.get(Calendar.AM_PM));
        System.out.println("Hora del dia:    "+gc.get(Calendar.HOUR_OF_DAY));
        System.out.println("Minuto:          "+gc.get(Calendar.MINUTE));
        System.out.println("Segundo:         "+gc.get(Calendar.SECOND));
        System.out.println("Dif. horaria:    "+gc.get(Calendar.ZONE_OFFSET));
    }
}
```

4.6.3 Clases DateFormat y SimpleDateFormat

DateFormat es una clase ***abstract*** que pertenece al package ***java.text*** y no al package ***java.util***, como las vistas anteriormente. La razón es para facilitar todo lo referente a la ***internacionalización***, que es un aspecto muy importante en relación con la conversión, que permite dar formato a fechas y horas de acuerdo con distintos criterios locales. Esta clase dispone de métodos ***static*** para convertir ***Strings*** representando fechas y horas en objetos de la clase ***Date***, y viceversa.

La clase ***SimpleDateFormat*** es la única clase derivada de ***DateFormat***. Es la clase que conviene utilizar. Esta clase se utiliza de la siguiente forma: se le pasa al constructor un ***String*** definiendo el formato que se desea utilizar. Por ejemplo:

```
import java.util.*;
import java.text.*;

class SimpleDateForm {
    public static void main(String arg[]) throws ParseException {
        SimpleDateFormat sdf1 = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
        SimpleDateFormat sdf2 = new SimpleDateFormat("dd-MM-yy");
        Date d = sdf1.parse("12-04-1968 11:23:45");
        String s = sdf2.format(d);
        System.out.println(s);
    }
}
```

La documentación de la clase ***SimpleDateFormat*** proporciona abundante información al respecto, incluyendo algunos ejemplos.

4.6.4 Clases TimeZone y SimpleTimeZone

La clase ***TimeZone*** es también una clase ***abstract*** que sirve para definir la zona horaria. Los métodos de esta clase son capaces de tener en cuenta el cambio de la hora en verano para ahorrar energía. La clase ***SimpleTimeZone*** deriva de ***TimeZone*** y es la que conviene utilizar.

El valor por defecto de la zona horaria es el definido en el ordenador en que se ejecuta el programa. Los objetos de esta clase pueden ser utilizados con los constructores y algunos métodos de la clase ***Calendar*** para establecer la zona horaria.

5. EL AWT (ABSTRACT WINDOWS TOOLKIT)

5.1 QUÉ ES EL AWT

El AWT (*Abstract Windows Toolkit*) es la parte de ***Java*** que se ocupa de construir interfaces gráficas de usuario. Aunque el AWT ha estado presente en ***Java*** desde la versión 1.0, la versión 1.1 representó un cambio notable, sobre todo en lo que respecta al ***modelo de eventos***. La versión 1.2 ha incorporado un modelo distinto de componentes llamado ***Swing***, que también está disponible en la versión 1.1 como package adicional. En este Capítulo se seguirá el AWT de ***Java 1.1***, también soportado por la versión 1.2.

5.1.1 Creación de una Interface Gráfica de Usuario

Para construir una interface gráfica de usuario hace falta:

1. Un "contenedor" o ***container***, que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos. Se correspondería con un ***formulario*** o una ***picture box*** de ***Visual Basic***.
2. Los ***componentes***: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc. Se corresponderían con los ***controles*** de ***Visual Basic***.
3. El ***modelo de eventos***. El usuario controla la aplicación actuando sobre los componentes, de ordinario con el ratón o con el teclado. Cada vez que el usuario realiza una determinada acción, se produce el ***evento*** correspondiente, que el sistema operativo transmite al AWT. El AWT crea un ***objeto*** de una determinada clase de evento, derivada de ***AWTEvent***. Este ***evento*** es transmitido a un determinado ***método*** para que lo gestione. En ***Visual Basic*** el entorno de desarrollo crea automáticamente el procedimiento que va a gestionar el evento (uniendo el nombre del control con el tipo del evento mediante el carácter _) y el usuario no tiene más que introducir el código. En ***Java*** esto es un poco más complicado: el componente u objeto que recibe el evento debe "registrar" o indicar previamente qué objeto se va a hacer cargo de gestionar ese evento.

En los siguientes apartados se verán con un cierto detalle estos tres aspectos del AWT. Hay que considerar que el AWT es una parte muy extensa y complicada de ***Java***, sobre la que existen libros con muchos cientos de páginas.

5.1.2 Objetos "event source" y objetos "event listener"

El modelo de eventos de ***Java*** está basado en que los objetos sobre los que se producen los eventos (***event sources***) "registran" los objetos que habrán de gestionarlos (***event listeners***), para lo cual los ***event listeners*** habrán de disponer de los ***métodos*** adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los ***event listeners*** disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface ***Listener***. Las interfaces ***Listener*** se corresponden con los tipos de ***eventos*** que se pueden producir. En los apartados siguientes se verán con más detalle los ***componentes*** que pueden recibir ***eventos***, los distintos tipos de ***eventos*** y los ***métodos*** de las interfaces ***Listener*** que hay que definir para gestionarlos. En este punto es muy importante ser capaz de buscar la información correspondiente en la documentación de ***Java***.

Las capacidades gráficas del AWT resultan pobres y complicadas en comparación con lo que se puede conseguir con ***Visual Basic***, pero tienen la ventaja de poder ser ejecutadas casi en cualquier ordenador y con cualquier sistema operativo.

5.1.3 Proceso a seguir para crear una aplicación interactiva (orientada a eventos)

Para avanzar un paso más, se resumen a continuación los pasos que se pueden seguir para construir una aplicación orientada a eventos sencilla, con interface gráfica de usuario:

1. Determinar los ***componentes*** que van a constituir la interface de usuario (botones, cajas de texto, menús, etc.).
2. Crear una ***clase*** para la aplicación que contenga la función ***main()***.
3. Crear una clase ***Ventana***, sub-clase de ***Frame***, que responda al evento ***WindowClosing()***.
4. La función ***main()*** deberá crear un objeto de la clase ***Ventana*** (en el que se van a introducir las componentes seleccionadas) y mostrarla por pantalla con el tamaño y posición adecuados.
5. Añadir al objeto ***Ventana*** todos los ***componentes*** y ***menús*** que deba contener. Se puede hacer en el constructor de la ventana o en el propio método ***main()***.
6. Definir los objetos ***Listener*** (objetos que se ocuparán de responder a los eventos, cuyas clases implementan las distintas interfaces ***Listener***) para cada uno de los eventos que deban estar soportados. En aplicaciones pequeñas, el propio objeto ***Ventana*** se puede ocupar de responder a los eventos de sus componentes. En programas más grandes se puede crear uno o más objetos de clases especiales para ocuparse de los eventos.
7. Finalmente, se deben implementar los métodos de las interfaces ***Listener*** que se vayan a hacer cargo de la gestión de los eventos.

5.1.4 Componentes y eventos soportados por el AWT de Java

5.1.4.1 Jerarquía de Componentes

Como todas las clases de ***Java***, los componentes utilizados en el AWT pertenecen a una determinada jerarquía de clases, que es muy importante conocer. Esta jerarquía de clases se muestra en la Figura 5.1. Todos los componentes descienden de la clase ***Component***, de la que pueden ya heredar algunos métodos interesantes. El ***package*** al que pertenecen estas clases se llama ***java.awt***.

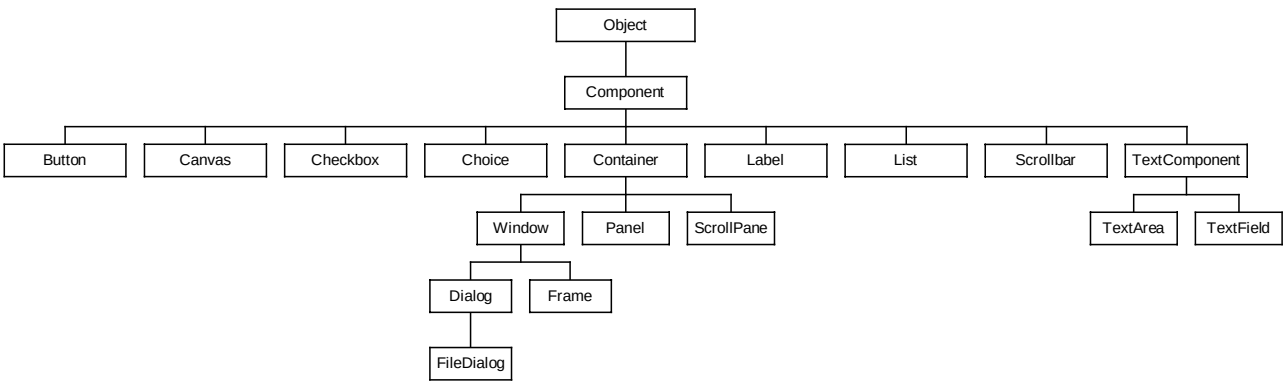

Figura 5.1. Jerarquía de clases para los componentes del AWT.

A continuación se resumen algunas características importantes de los componentes mostrados en la Figura 5.2:

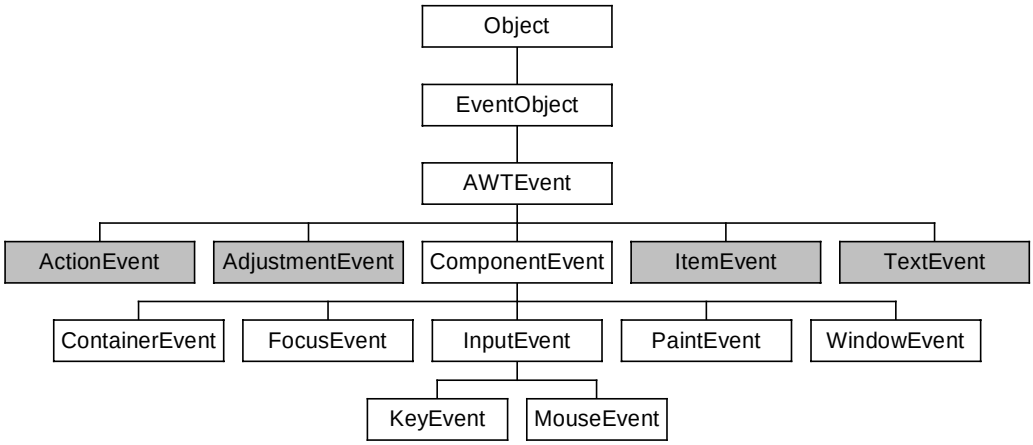

Figura 5.2. Jerarquía de eventos de Java.

1. Todos los ***Components*** (excepto ***Window*** y los que derivan de ella) deben ser añadidos a un ***Container***. También un ***Container*** puede ser añadido a otro ***Container***.
2. Para añadir un ***Component*** a un ***Container*** se utiliza el método ***add()*** de la clase ***Container***:

   ```
   containerName.add(componentName);
   ```

3. Los ***Containers*** de máximo nivel son las ***Windows*** (***Frames*** y ***Dialogs***). Los ***Panels*** y ***ScrollPanes*** deben estar siempre dentro de otro ***Container***.
4. Un ***Component*** sólo puede estar en un ***Container***. Si está en un ***Container*** y se añade a otro, deja de estar en el primero.
5. La clase ***Component*** tiene una serie de funcionalidades básicas comunes (variables y métodos) que son heredadas por todas sus ***sub-clases***.

5.1.4.2 Jerarquía de eventos

Todos los eventos de ***Java 1.1*** y ***Java 1.2*** son objetos de clases que pertenecen a una determinada jerarquía de clases. La super-clase ***EventObject*** pertenece al package ***java.util***. De ***EventObject*** deriva la clase ***AWTEvent***, de la que dependen todos los eventos de AWT. La **¡Error! No se encuentra el origen de la referencia.** muestra la jerarquía de clases para los eventos de ***Java***. Por conveniencia, estas clases están agrupadas en el package ***java.awt.event***.

Los eventos de ***Java*** pueden ser de alto y bajo nivel. Los ***eventos de alto nivel*** se llaman también *eventos semánticos*, porque la acción de la que derivan tiene un significado en sí misma, en el contexto de las interfaces gráficas de usuario. Los ***eventos de bajo nivel*** son las acciones elementales que hacen posible los eventos de alto nivel. Son ***eventos de alto nivel*** los siguientes eventos: los cuatro que tienen que ver con clicar sobre botones o elegir comandos en menús (***ActionEvent***), cambiar valores en barras de desplazamiento (***AdjustmentEvent***), elegir valores (***ItemEvents***) y cambiar el texto (***TextEvent***). En la Figura 5.2 los eventos de alto nivel aparecen con fondo gris.

Los ***eventos de bajo nivel*** son los que se producen con las operaciones elementales con el ratón, teclado, containers y windows. Las seis clases de eventtos de bajo nivel son los eventos

relacionados con componentes (***ComponentEvent***), con los containers (***ContainerEvent***), con pulsar teclas (***KeyEvent***), con mover, arrastrar, pulsar y soltar con el ratón (***MouseEvent***), con obtener o perder el focus (***FocusEvent***) y con las operaciones con ventanas (***WindowEvent***).

El modelo de eventos se complica cuando se quiere connstruir un tipo de componente propio, no estándar del AWT. En este caso hay que interceptar los eventos de bajo nivel de ***Java*** y adecuarlos al problema que se trata de resolver. Éste es un tema que no se va a tratar en este manual.

5.1.4.3 Relación entre Componentes y Eventos

La Tabla 5.1 muestra los componentes del AWT y los eventos específicos de cada uno de ellos, así como una breve explicación de en qué consiste cada tipo de evento.

Component	**Eventos generados**	**Significado**
Button	ActionEvent	Clicar en el botón
Checkbox	ItemEvent	Seleccionar o deseleccionar un item
CheckboxMenuItem	ItemEvent	Seleccionar o deseleccionar un item
Choice	ItemEvent	Seleccionar o deseleccionar un item
Component	ComponentEvent	Mover, cambiar tamaño, mostrar u ocultar un componente
	FocusEvent	Obtener o perder el focus
	KeyEvent	Pulsar o soltar una tecla
	MouseEvent	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón (tener en cuenta que este evento tiene dos Listener)
Container	ContainerEvent	Añadir o eliminar un componente de un container
List	ActionEvent	Hacer doble click sobre un item de la lista
	ItemEvent	Seleccionar o deseleccionar un item de la lista
MunuItem	ActionEvent	Seleccionar un item de un menú
Scrollbar	AdjustementEvent	Cambiar el valor de la scrollbar
TextComponent	TextEvent	Cambiar el texto
TextField	ActionEvent	Terminar de editar un texto pulsando Intro
Window	WindowEvent	Acciones sobre una ventana: abrir, cerrar, iconizar, restablecer e iniciar el cierre

Tabla 5.1. Componentes del AWT y eventos específicos que generan.

La relación entre componentes y eventos indicada en la Tabla 5.1 pueden inducir a engaño si no se tiene en cuenta que los eventos propios de una ***super-clase*** de componentes pueden afectar también a los componentes de sus ***sub-clases***. Por ejemplo, la clase ***TextArea*** no tiene ningún evento específico o propio, pero puede recibir los de su ***super-clase TextComponent***.

La Tabla 5.2 muestra los ***componentes*** del AWT y ***todos los tipos de eventos*** que se pueden producir sobre cada uno de ellos, teniendo en cuenta también los que son específicos de sus ***super-clases***. Entre ambas tablas se puede sacar una idea bastante precisa de qué tipos de eventos están soportados en ***Java*** y qué eventos concretos puede recibir cada componente del AWT. En la práctica, no todos los tipos de evento tienen el mismo interés.

AWT Components	**Eventos que se pueden generar**									
	AtionEvent	**AdjustementEvent**	**ComponentEvent**	**ContainerEvent**	**FocusEvent**	**ItemEvent**	**KeyEvent**	**MouseEvent**	**TextEvent**	**WindowEvent**
Button	✔		✔		✔		✔	✔		
Canvas			✔		✔		✔	✔		
Checkbox			✔		✔	✔	✔	✔		
Checkbox-MenuItem						✔				
Choice			✔		✔	✔	✔	✔		
Component			✔		✔		✔	✔		
Container			✔	✔	✔		✔	✔		
Dialog			✔	✔	✔		✔	✔		✔
Frame			✔	✔	✔		✔	✔		✔
Label			✔		✔		✔	✔		
List	✔		✔		✔	✔	✔	✔		
MenuItem	✔									
Panel			✔	✔	✔		✔	✔		
Scrollbar		✔	✔		✔		✔	✔		
TextArea			✔		✔		✔	✔	✔	
TextField	✔		✔		✔		✔	✔	✔	
Window			✔	✔	✔		✔	✔		✔

Tabla 5.2. Eventos que generan los distintos componentes del AWT.

5.1.5 Interfaces Listener

Una vez vistos los distintos eventos que se pueden producir, conviene ver cómo se deben gestionar estos eventos. A continuación se detalla cómo se gestionan los eventos según el modelo de ***Java***:

1. Cada objeto que puede recibir un evento (***event source***), "registra" uno o más objetos para que los gestionen (***event listener***). Esto se hace con un método que tiene la forma,

   ```
   eventSourceObject.addEventListener(eventListenerObject);
   ```

   donde ***eventSourceObject*** es el objeto en el que se produce el evento, y ***eventListenerObject*** es el objeto que deberá gestionar los eventos. La relación entre ambos se establece a través de una interface ***Listener*** que la clase del ***eventListenerObject*** debe implementar. Esta interface proporciona la declaración de los métodos que serán llamados cuando se produzca el evento. La interface a implementar depende del tipo de evento. La Tabla 5.3 relaciona los distintos tipos de eventos, con la interface que se debe implementar para gestionarlos. Se indican también los métodos declarados en cada interface.

   Es importante observar la correspondencia entre ***eventos*** e ***interfaces Listener***. Cada evento tiene su interface, excepto el ratón que tiene dos interfaces ***MouseListener*** y ***MouseMotionListener***. La razón de esta duplicidad de interfaces se encuentra en la peculiaridad de los eventos que se producen cuando el ratón se mueve. Estos eventos, que se producen con muchísima más frecuencia que los simples clicks, por razones de eficiencia son gestionados por una interface especial: ***MouseMotionListener***.

Obsérvese que el ***nombre de la interface*** coincide con el ***nombre del evento***, sustituyendo la palabra ***Event*** por ***Listener***.

2. Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente la correspondiente interface ***Listener***, se deben definir los métodos de dicha interface. Siempre hay que definir ***todos los métodos*** de la interface, aunque algunos de dichos métodos puedan estar "vacíos". Un ejemplo es la implementación de la interface ***WindowListener*** vista en el Apartado 1.3.9 (en la página 17), en el que todos los métodos estaban vacíos excepto ***windowClosing()***.

Evento	Interface Listener	Métodos de Listener
ActionEvent	ActionListener	actionPerformed()
AdjustementEvent	AdjustementListener	adjustementValueChanged()
ComponentEvent	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
ContainerEvent	ContainerListener	componentAdded(), componentRemoved()
FocusEvent	FocusListener	focusGained(), focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
	MouseMotionListener	mouseDragged(), mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

Tabla 5.3. Métodos relacionados con cada evento a través de una interface Listener.

5.1.6 Clases Adapter

Java proporciona ayudas para definir los métodos declarados en las interfaces ***Listener***. Una de estas ayudas son las clases ***Adapter***, que existen para cada una de las interfaces ***Listener*** que tienen más de un método. Su nombre se construye a partir del nombre de la interface, sustituyendo la palabra "*Listener*" por "*Adapter*". Hay 7 clases ***Adapter***: *ComponentAdapter*, *ContainerAdapter*, *FocusAdapter*, *KeyAdapter*, *MouseAdapter*, *MouseMotionAdapter* y *WindowAdapter*.

Las clases ***Adapter*** derivan de ***Object***, y son clases predefinidas que contienen ***definiciones vacías*** para todos los métodos de la interface. Para crear un objeto que responda al evento, en vez de crear una clase que implemente la interface ***Listener***, basta crear una clase que derive de la clase ***Adapter*** correspondiente, y redefina sólo los métodos de interés. Por ejemplo, la clase ***VentanaCerrable*** del Apartado 1.3.9 (página 17) se podía haber definido de la siguiente forma:

```
1.    // Fichero VentanaCerrable2.java

2.    import java.awt.*;
3.    import java.awt.event.*;

4.    class VentanaCerrable2 extends Frame {

5.        // constructores
6.        public VentanaCerrable2() { super(); }
7.        public VentanaCerrable2(String title) {
8.            super(title);
9.            setSize(500,500);
10.           CerrarVentana cv = new CerrarVentana();
11.           this.addWindowListener(cv);
12.       }
```

```
13.    } // fin de la clase VentanaCerrable2

14.    // definición de la clase CerrarVentana
15.    class CerrarVentana extends WindowAdapter {
16.        void windowClosing(WindowEvent we) { System.exit(0); }
17.    } // fin de la clase CerrarVentana
```

Las sentencias 15-17 definen una clase auxiliar (***helper class***) que deriva de la clase ***WindowAdapter***. Dicha clase hereda definiciones vacías de todos los métodos de la interface ***WindowListener***. Lo único que tiene que hacer es redefinir el único método que se necesita para cerrar las ventanas. El constructor de la clase ***VentanaCerrable*** crea un objeto de la clase ***CerrarVentana*** en la sentencia 10 y lo registra como ***event listener*** en la sentencia 11. En la sentencia 11 la palabra ***this*** es opcional: si no se incluye, se supone que el ***event source*** es el objeto de la clase en la que se produce el evento, en este caso la propia ventana.

Todavía hay otra forma de responder al evento que se produce cuando el usuario desea cerrar la ventana. Las ***clases anónimas*** de ***Java*** son especialmente útiles en este caso. En realidad, para gestionar eventos sólo hace falta un objeto que sea registrado como ***event listener*** y contenga los métodos adecuados de la interface ***Listener***. Las ***clases anónimas*** son útiles cuando sólo se necesita un objeto de la clase, como es el caso. La nueva definición de la clase ***VentanaCerrable*** podría ser como sigue:

```
1.     // Fichero VentanaCerrable3.java

2.     import java.awt.*;
3.     import java.awt.event.*;

4.     class VentanaCerrable3 extends Frame {

5.         // constructores
6.         public VentanaCerrable3() { super(); }
7.         public VentanaCerrable3(String title) {
8.             super(title);
9.             setSize(500,500);
10.            this.addWindowListener(new WindowAdapter() {
11.                                       public void windowClosing() {System.exit(0);}
12.                                   });
13.        }

14.    } // fin de la clase VentanaCerrable
```

Obsérvese que el objeto ***event listener*** se crea justamente en el momento de pasárselo como argumento al método ***addWindowListener()***. Se sabe que se está creando un nuevo objeto porque aparece la palabra ***new***. Debe tenerse en cuenta que no se está creando un nuevo objeto de ***WindowAdapter*** (entre otras cosas porque dicha clase es ***abstract***), sino extendiendo la clase ***WindowAdapter***, aunque la palabra ***extends*** no aparezca. Esto se sabe por las ***llaves*** que se abren al final de la línea 10. Los ***paréntesis vacíos*** de la línea 10 podrían contener los argumentos para el constructor de ***WindowAdapter***, en el caso de que dicho constructor necesitara argumentos. En la sentencia 11 se redefine el método ***windowClosing()***. En la línea 12 se cierran las llaves de la ***clase anónima***, se cierra el paréntesis del método ***addWindowListener()*** y se pone el ***punto y coma*** de terminación de la sentencia que empezó en la línea 10. Para más información sobre las ***clases anónimas*** ver el Apartado 3.10.4, en la página 56.

5.2 COMPONENTES Y EVENTOS

Error! Unknown switch argument.

Figura 5.3. Algunos componentes del AWT.

En este Apartado se van a ver los ***componentes gráficos*** de ***Java***, a partir de los cuales se pueden construir interfaces gráficas de usuario. Se verán también, en paralelo y lo más cerca posible en el texto, las diversas clases de ***eventos*** que pueden generar cada uno de esos componentes.

La Figura 5.3, tomada de uno de los ejemplos de ***Java Tutorial*** de ***Sun***, muestra algunos componentes del AWT. En ella se puede ver un *menú*, una superficie de dibujo o *canvas* en la que se puede dibujar y escribir texto, una *etiqueta*, una *caja de texto* y un *área de texto*, un *botón de comando* y un *botón de selección*, una *lista* y una *caja de selección desplegable*.

5.2.1 Clase Component

Métodos de Component	**Función que realizan**
boolean isVisible(), void setVisible(boolean)	Permiten chequear o establecer la visibilidad de un componente
boolean isShowing()	Permiten saber si un componente se está viendo. Para ello tanto el componente debe ser visible, y su container debe estar mostrándose
boolean isEnabled(), void setEnabled(boolean)	Permiten saber si un componente está activado y activarlo o desactivarlo
Point getLocation(), Point getLocationOnScreen()	Permiten obtener la posición de la esquina superior izquierda de un componente respecto al componente-padre o a la pantalla
void setLocation(Point), void setLocation(int x, int y)	Desplazan un componente a la posición especificada respecto al container o componente-padre
Dimension getSize(), void setSize(int w, int h), void setSize(Dimension d)	Permiten obtener o establecer el tamaño de un componente
Rectangle getBounds(), void setBounds(Rectangle), void setBounds(int x, int y, int width, int height)	Obtienen o establecen la posición y el tamaño de un componente
invalidate(), validate(), doLayout()	invalidate() marca un componente y sus contenedores para indicar que se necesita volver a aplicar el Layout Manager. validate() se asegura que el Layout Manager está bien aplicado. doLayout() hace que se aplique el Layout Manager
paint(Graphics), repaint() y update(Graphics)	Métodos gráficos para dibujar en la pantalla
setBackground(Color), setForeground(Color)	Métodos para establecer los colores por defecto

Tabla 5.4. Métodos de la clase Component.

La clase ***Component*** es una clase ***abstract*** de la que derivan todas las clases del AWT, según el diagrama mostrado previamente en la Figura 5.1, en la página 82. Los métodos de esta clase son importantes porque son heredados por todos los componentes del AWT. La Tabla 5.4 muestra algunos de los métodos más utilizados de la clase ***Component***. En las declaraciones de los métodos de dicha clase aparecen las clases ***Point***, ***Dimension*** y ***Rectangle***. La clase ***java.awt.Point*** tiene dos variables miembro *int* llamadas ***x*** e ***y***. La clase ***java.awt.Dimension*** tiene dos variables miembro *int*: ***height*** y ***width***. La clase ***java.awt.Rectangle*** tiene cuatro variables *int*: ***height***, ***width***, ***x*** e ***y***. Las tres son sub-clases de ***Object***.

Además de los métodos mostrados en la Tabla 5.4, la clase ***Component*** tiene un gran número de métodos básicos cuya funcionalidad puede estudiarse mediante la documentación on-line de ***Java***. Entre otras funciones, permiten controlar los ***colores***, las ***fonts*** y los ***cursores***.

A continuación se describen las clases de ***eventos*** más generales, relacionados bien con la clase ***Component***, bien con diversos tipos de componentes que también se presentan a continuación.

5.2.2 Clases EventObject y AWTEvent

Todos los métodos de las interfaces ***Listener*** relacionados con el AWT tienen como argumento único un objeto de alguna clase que desciende de la clase ***java.awt.AWTEvent*** (ver Figura 5.2, en la página 83).

La clase ***AWTEvent*** desciende de ***java.util.EventObject***. La clase ***AWTEvent*** no define ningún método, pero hereda de ***EventObject*** el método ***getSource()***:

```
Object getSource();
```

que devuelve una referencia al objeto que generó el evento. Las clases de eventos que descienden de ***AWTEvent*** definen métodos similares a ***getSource()*** con unos valores de retorno menos genéricos. Por ejemplo, la clase ***ComponentEvent*** define el método ***getComponent()***, cuyo valor de retorno es un objeto de la clase ***Component***.

5.2.3 Clase ComponentEvent

Los eventos ***ComponentEvent*** se generan cuando un ***Component*** de cualquier tipo se muestra, se oculta, o cambia de posición o de tamaño. Los eventos de *mostrar* u *ocultar* ocurren cuando se llama al método ***setVisible(boolean)*** del ***Component***, pero no cuando se *minimiza* la ventana.

Otro método útil de la clase ***ComponentEvent*** es ***Component getComponent()*** que devuelve el componente que generó el evento. Se puede utilizar en lugar de ***getSource()***.

5.2.4 Clases InputEvent y MouseEvent

De la clase ***InputEvent*** descienden los eventos del ratón y el teclado. Esta clase dispone de métodos pera detectar si los botones del ratón o las teclas especiales han sido pulsadas. Estos botones y estas teclas se utilizan para cambiar o modificar el significado de las acciones del usuario. La clase ***InputEvent*** define unas constantes que permiten saber qué teclas especiales o botones del ratón estaban pulsados al producirse el evento, como son: SHIFT_MASK, ALT_MASK, CTRL_MASK, BUTTON1_MASK, BUTTON2_MASK y BUTTON3_MASK, cuyo significado es evidente. La Tabla 5.5 muestra algunos métodos de esta clase.

Métodos heredados de la clase InputEvent	**Función que realizan**
boolean isShiftDown(), boolean isAltDown(), boolean isControlDown()	Devuelven un boolean con información sobre si esa tecla estaba pulsada o no
int getModifiers()	Obtiene información con una máscara de bits sobre las teclas y botones pulsados
long getWhen()	Devuelve la hora en que se produjo el evento

Tabla 5.5. Métodos de la clase InputEvent.

Se produce un ***MouseEvent*** cada vez que el cursor movido por el ratón entra o sale de un componente visible en la pantalla, al clicar, o cuando se pulsa o se suelta un botón del ratón. Los métodos de la interface ***MouseListener*** se relacionan con estas acciones, y son los siguientes (ver Tabla 5.3, en la página 86): ***mouseClicked()***, ***mouseEntered()***, ***mouseExited()***, ***mousePressed()*** y ***mouseReleased()***. Todos son *void* y reciben como argumento un objeto ***MouseEvent***. La Tabla 5.6 muestra algunos métodos de la clase ***MouseEvent***.

Métodos de la clase MouseEvent	**Función que realizan**
int getClickCount()	Devuelve el número de clicks en ese evento
Point getPoint(), int getX(), int getY()	Devuelven la posición del ratón al producirse el evento
boolean isPopupTrigger()	Indica si este evento es el que dispara los menús popup

Tabla 5.6. Métdos de la clase MouseEvent.

La clase ***MouseEvent*** define una serie de constantes ***int*** que permiten identificar los tipos de eventos que se han producido: MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_DRAGGED.

Además, el método ***Component getComponent()***, heredado de ***ComponentEvent***, devuelve el componente sobre el que se ha producido el evento.

Los eventos ***MouseEvent*** disponen de una segunda interface para su gestión, la interface ***MouseMotionListener***, cuyos métodos reciben también como argumento un evento de la clase ***MouseEvent***. Estos eventos están relacionados con el ***movimiento del ratón***. Se llama a un método de la interface ***MouseMotionListener*** cuando el usuario utiliza el ratón (o un dispositivo similar) para mover el cursor o arrastrarlo sobre la pantalla. Los métodos de la interface ***MouseMotionListener*** son ***mouseMoved()*** y ***mouseDragged()***.

5.2.5 Clase FocusEvent

El ***Focus*** está relacionado con la posibilidad de sustituir al ratón por el teclado en ciertas operaciones. De los componentes que aparecen en pantalla, en un momento dado hay sólo uno que puede recibir las acciones del teclado y se dice que ese componente tiene el ***Focus***. El componente que tiene el ***Focus*** aparece diferente de los demás (resaltado de alguna forma). Se cambia el elemento que tiene el ***Focus*** con la tecla ***Tab*** o con el ratón. Se produce un ***FocusEvent*** cada vez que un componente gana o pierde el ***Focus***.

El método ***requestFocus()*** de la clase ***Component*** permite hacer desde el programa que un componente obtenga el ***Focus***.

El método ***boolean isTemporary()***, de la clase ***FocusEvent***, indica si la pérdida del ***Focus*** es o no temporal (puede ser temporal por haberse ocultado o dejar de estar activa la ventana, y recuperarse al cesar esta circunstancia).

El método ***Component getComponent()*** es heredado de ***ComponentEvent***, y permite conocer el componente que ha ganado o perdido el ***Focus***. Las constantes de esta clase ***FOCUS_GAINED*** y ***FOCUS_LOST*** permiten saber el tipo de evento ***FocusEvent*** que se ha producido.

5.2.6 Clase Container

La clase ***Container*** es también una clase muy general. De ordinario, nunca se crea un objeto de esta clase, pero los métodos de esta clase son heredados por las clases ***Frame*** y ***Panel***, que sí se utilizan con mucha frecuencia para crear objetos. La Tabla 5.7 muestra algunos métodos de ***Container***.

Los containers mantienen una ***lista de los objetos*** que se les han ido añadiendo. Cuando se añade un nuevo objeto se incorpora al final de la lista, salvo que se especifique una posición determinada. En esta clase tiene mucha importancia todo lo que tiene que ver con los ***Layout Managers***, que se explicarán más adelante. La Tabla 5.7 muestra algunos métodos de la clase ***Container***.

5.2.7 Clase ContainerEvent

Métodos de Container	**Función que realizan**
Component add(Component)	Añade un componente al container
doLayout()	Ejecuta el algoritmo de ordenación del layout manager
Component getComponent(int)	Obtiene el n-ésimo componente en el container
Component getComponentAt(int, int), Component getComponentAt(Point)	Obtiene el componente que contine un determinado punto
int getComponentCount()	Obtiene el número de componentes en el container
Component[] getComponents()	Obtiene los componentes en este container.
remove(Component), remove(int), removeAll()	Elimina el componente especificado.
setLayout(LayoutManager)	Determina el layout manager para este container

Tabla 5.7. Métodos de la clase Container.

Los ***ContainerEvents*** se generan cada vez que un ***Component*** se añade o se retira de un ***Container***. Estos eventos sólo tienen un *papel de aviso* y no es necesario gestionarlos para que se realize la operación.

Los métodos de esta clase son ***Component getChild()***, que devuelve el ***Component*** añadido o eliminado, y ***Container getContainer()***, que devuelve el ***Container*** que generó el evento.

5.2.8 Clase Window

Los objetos de la clase ***Window*** son ventanas de máximo nivel, pero ***sin bordes*** y ***sin barra de menús***. En realidad son más interesantes las clases que derivan de ella: ***Frame*** y ***Dialog***. Los métodos más útiles, por ser heredados por las clases ***Frame*** y ***Dialog***, se muestran en la Tabla 5.8.

Métodos de Window	**Función que realizan**
toFront(), toBack()	Para desplazar la ventana hacie adelante y hacia atrás en la pantalla
show()	Muestra la ventana y la trae a primer plano
pack()	Hace que los componentes se reajusten al tamaño preferido

Tabla 5.8. Métodos de la clase Window.

5.2.9 Clase WindowEvent

Se produce un ***WindowEvent*** cada vez que se *abre*, *cierra*, *iconiza*, *restaura*, *activa* o *desactiva* una ventana. La interface ***WindowListener*** contiene los siete métodos siguientes, con los que se puede responder a este evento:

```
void windowOpened(WindowEvent we);  // antes de mostrarla por primera vez
void windowClosing(WindowEvent we); // al recibir una solicitud de cierre
void windowClosed(WindowEvent we);  // después de cerrar la ventana
void windowIconified(WindowEvent we);
void windowDeiconified(WindowEvent we);
void windowActivated(WindowEvent we);
```

```
void windowDeactivated(WindowEvent we);
```

El uso más frecuente de ***WindowEvent*** es para cerrar ventanas (por defecto, los objetos de la clase ***Frame*** no se pueden cerrar más que con ***Ctrl+Alt+Supr***). También se utiliza para detener ***threads*** y liberar recursos al iconizar una ventana (que contiene por ejemplo animaciones) y comenzar de nuevo al restaurarla.

La clase ***WindowEvent*** define la siguiente serie de constantes que permiten identificar el tipo de evento:

WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED,
WINDOW_ICONIFIED, WINDOW_DEICONIFIED,
WINDOW_ACTIVATED, WINDOW_DEACTIVATED

En la clase ***WindowEvent*** el método ***Window getWindow()*** devuelve la ***Window*** que generó el evento. Se utiliza en lugar de ***getSource()***.

5.2.10 Clase Frame

Es una ventana ***con un borde*** y que puede tener una ***barra de menús***. Si una ventana depende de otra ventana, es mejor utilizar una ***Window*** (ventana sin borde ni barra de menús) que un ***Frame***. La Tabla 5.9 muestra algunos métodos más utilizados de la clase ***Frame***.

Métodos de Frame	**Función que realiza**
Frame(), Frame(String title)	Constructores de Frame
String getTitle(), setTitle(String)	Obtienen o determinan el título de la ventana
MenuBar getMenuBar(), setMenuBar(MenuBar), remove(MenuComponent)	Permite obtener, establecer o eliminar la barra de menús
Image getIconImage(), setIconImage(Image)	Obtienen o determinan el icono que apareceré en la barra de títulos
setResizable(boolean), boolean isResizable()	Determinan o chequean si se puede cambiar el tamaño
dispose()	Método que libera los recursos utilizados en una ventana. Todos los componentes de la ventana son destruidos.

Tabla 5.9. Métodos de la clase Frame.

Además de los métodos citados, se utilizan mucho los métodos ***show()***, ***pack()***, ***toFront()*** y ***toBack()***, heredados de la super-clase ***Window***.

5.2.11 Clase Dialog

Un ***Dialog*** es una ventana que depende de otra ventana (de una ***Frame***). Si una ***Frame*** se cierra, se cierran también los ***Dialog*** que dependen de ella; si se iconifica, sus ***Dialog*** desaparecen; si se restablece, sus ***Dialog*** aparecen de nuevo. Este comportamiento se obtiene de forma automática.

Las ***Applets*** estándar no soportan ***Dialogs*** porque no son ***Frames*** de ***Java***. Las ***Applets*** que abren ***Frames*** sí pueden soportar ***Dialogs***.

Un ***Dialog modal*** require la atención inmediata del usuario: no se puede hacer ninguna otra cosa hasta no haber cerrado el ***Dialog***. Por defecto, los ***Dialogs*** son ***no modales***. La Tabla 5.10 muestra los métodos más importantes de la clase ***Dialog***. Se pueden utilizar también los métodos heredados de sus super-clases.

Métodos de Dialog	**Función que realiza**
Dialog(Frame fr), Dialog(Frame fr, boolean mod), Dialog(Frame fr, String title), Dialog(Frame fr, String title, boolean mod)	Constructores
String getTitle(), setTitle(String)	Permite obtener o determinar el título
boolean isModal(), setModal(boolean)	Pregunta o determina si el Dialog es modal o no
boolean isResizable(), setResizable(boolean)	Pregunta o determina si se puede cambiar el tamaño
show()	Muestra y trae a primer plano el Dialog

Tabla 5.10. Métodos de la clase Dialog.

5.2.12 Clase FileDialog

La clase ***FileDialog*** muestra una ventana de diálogo en la cual se puede seleccionar un fichero. Esta clase deriva de ***Dialog***. Las constantes enteras LOAD (abrir ficheros para lectura) y SAVE (abrir ficheros para escritura) definen el ***modo*** de apertura del fichero. La Tabla 5.11 muestra algunos métodos de esta clase.

Métodos de la clase FileDialog	**Función que realizan**
FileDialog(Frame parent), FileDialog(Frame parent, String title), public FileDialog(Frame parent, String title, int mode)	Constructores
int getMode(),setMode(int mode)	Modo de apertura (SAVE o LOAD)
String getDirectory(), String getFile()	Obtiene el directorio o fichero elegido
setDirectory(String dir), setFile(String file)	Determina el directorio o fichero elegido
FilenameFilter getFilenameFilter(), setFilenameFilter(FilenameFilter filter)	Determina o establece el filtro para los ficheros

Tabla 5.11. Métodos de la clase FileDialog.

Las clases que implementan la interface ***java.io.FilenameFilter*** permiten filtrar los ficheros de un directorio. Para más información, ver la docomentación on-line.

5.2.13 Clase Panel

Un ***Panel*** es un ***Container*** de propósito general. Se puede utilizar tal cual para contener otras componentes, y también crear una sub-clase para alguna finalidad más específica. Por defecto, el ***Layout Manager*** de ***Panel*** es ***FlowLayout***. Los ***Applets*** son sub-clases de ***Panel***. La Tabla 5.12 muestra los métodos más importantes que se utilizan con la clase ***Panel***, que son algunos métodos heredados de ***Componet*** y ***Container***, pues la clase ***Panel*** no tiene métodos propios.

Métodos de Panel	**Función que realiza**
Panel(), Panel(LayoutManager miLM)	Constructores de Panel
Métodos heredados de Container y Component	
add(Component), add(Component, int)	Añade componentes al panel
setLayout(), getLayout()	Establece o permite obtener el layout manager utilizado
validate(), doLayout()	Para reorganizar los componentes después de algún cambio. Es mejor utilizar validate()
remove(int), remove(Component), removeAll()	Para eliminar componentes
Dimension getMaximumSize(), Dimension getMinimumSize(), Dimension getPreferredSize()	Permite obtener los tamaños máximo, mínimo y preferido
Insets getInsets()	

Tabla 5.12. Métodos de la clase Panel.

Un ***Panel*** puede contener otros ***Panel***. Esto es una gran ventaja respecto a los demás tipos de containers, que son containers de máximo nivel y no pueden introducirse en otros containers.

Insets es una clase que deriva de ***Object***. Sus variables son ***top***, ***left***, ***botton***, ***right***. Representa el *espacio que se deja libre* en los bordes de un ***Container***. Se establece mediante la llamada al adecuado constructor del ***Layout Manager***.

5.2.14 Clase Button

Aunque según la Tabla 5.2, en la página 85, un ***Button*** puede recibir seis tipos de eventos, lo más importante es que al clicar sobre él se genera un evento de la clase ***ActionEvent***.

El aspecto de un ***Button*** depende de la plataforma (PC, Mac, Unix), pero la funcionalidad siempre es la misma. Se puede cambiar el ***texto*** y la ***font*** que aparecen en el ***Button***, así como el ***foreground*** y ***backgroung color***. También se puede establecer que esté activado o no. La Tabla 5.13 muestra los métodos más importantes de la clase ***Button***.

Métodos de la clase Button	**Función que realiza**
Button(String label) y Button()	Constructores
setLabel(String str), String getLabel()	Permite establecer u obtener la etiqueta del Button
addActionListener(ActionListener al), removeActionListener(ActionListener al)	Permite registrar el objeto que gestionará los eventos, que deberá implementar ActionListener
setActionCommand(String cmd), String getActionCommand()	Establece y recupera un nombre para el objeto Button independiente del label y del idioma

Tabla 5.13. Métodos de la clase Button.

5.2.15 Clase ActionEvent

Los eventos ***ActionEvent*** se producen al clicar con el ratón en un botón (***Button***), al elegir un comando de un menú (***MenuItem***), al hacer doble clic en un elemento de una lista (***List***) y al pulsar ***Intro*** para introducir un texto en una caja de texto (***TextField***).

El método ***String getActionCommand()*** devuelve el texto asociado con la acción que provocó el evento. Este texto se puede fijar con el método ***setActionCommand(String str)*** de las clases ***Button*** y ***MenuItem***. Si el texto no se ha fijado con este método, el método ***getActionCommand()*** devuelve el texto mostrado por el componente (su etiqueta). Para objetos con varios items el valor devuelto es el nombre del item seleccionado.

El método ***int getModifiers()*** devuelve un entero representando una constante definida en ***ActionEvent*** (SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK). Estas constantes sirven para determinar si se pulsó una de estas teclas modificadores mientras se clicaba. Por ejemplo, si se estaba pulsando la tecla CTRL la siguiente expresión es distinta de cero:

```
actionEvent.getModifiers() & ActionEvent.CTRL_MASK
```

5.2.16 Clase Canvas

Una ***Canvas*** es una zona rectangular de pantalla en la que se puede dibujar y en la que se pueden generar eventos. Las ***Canvas*** permiten realizar dibujos, mostrar imágenes y crear componentes a medida, de modo que muestren un aspecto similar en todas las plataformas. La Tabla 5.14 muestra los métodos de la clase ***Canvas***.