Métodos de Canvas	**Función que realiza**
Canvas()	Es el único constructor de esta clase
paint(Graphics g);	Dibuja un rectángulo con el color de background. Lo normal es que las sub-clases de Canvas redefinan este método.

Tabla 5.14. Métodos de la clase Canvas.

Desde los objetos de la clase ***Canvas*** se puede llamar a los métodos ***paint()*** y ***repaint()*** de la super-clase ***Component***. Con frecuencia conviene redefinir los siguientes métodos de ***Component***: ***getPreferredSize()***, ***getMinimumSize()*** y ***getMaximumSize()***, que devuelven un objeto de la clase ***Dimension***. El ***LayoutManager*** se encarga de utilizar estos valores.

La clase ***Canvas*** no tiene eventos propios, pero puede recibir los eventos ***ComponentEvent*** de su super-clase ***Component***.

5.2.17 Component Checkbox y clase CheckboxGroup

Los objetos de la clase ***Checkbox*** son ***botones de opción o de selección*** con dos posibles valores: ***on*** y ***off***. Al cambiar la selección de un ***Checkbox*** se produce un ***ItemEvent***.

Métodos de Checkbox	**Función que realizan**
Checkbox(), Checkbox(String), Checkbox(String, boolean), Checkbox(String, boolean, CheckboxGroup), Checkbox(String, CheckboxGroup, boolean)	Constructores de Checkbox. Algunos permiten establecer la etiqueta, si está o no seleccionado y si pertenece a un grupo
addItemListener(ItemListener), removeItemListener(ItemListener)	Registra o elimina los objetos que gestionarán los eventos ItemEvent
setLabel(String), String getLabel()	Establece u obtiene la etiqueta del componente
setState(boolean), boolean getState()	Establece u obtiene el estado (true o false, según esté seleccionado o no)
setCheckboxGroup(CheckboxGroup), CheckboxGroup getCheckboxGroup()	Establece u obtiene el grupo al que pertenece el Checkbox
Métodos de CheckboxGroup	**Función que realizan**
CheckboxGroup()	Constructores de CheckboxGroup:
Checkbox getSelectedCheckbox(), setSelectedCheckbox(Checkbox box)	Obtiene o establece el componente seleccionado de un grupo:

Tabla 5.15. Métodos de las clases Checkbox y CheckboxGroup.

La clase ***CheckboxGroup*** permite la opción de agrupar varios ***Checkbox*** de modo que uno y sólo uno esté en ***on*** (al comienzo puede que todos estén en ***off***). Se corresponde con los botones de opción de Visual Basic. La Tabla 5.15 muestra los métodos más importantes de estas clases.

Cuando el usuario actúa sobre un objeto ***Checkbox*** se ejecuta el método ***itemStateChanged()***, que es el único método de la interface ***ItemListener***. Si hay varias ***checkboxes*** cuyos eventos se gestionan en un mismo objeto y se quiere saber cuál es la que ha recibido el evento, se puede utilizar el método ***getSource()*** del evento ***EventObject***. También se puede utilizar el método ***getItem()*** de ***ItemEvent***, cuyo valor de retorno es un ***Object*** que contiene el ***label*** del componente (para convertirlo a ***String*** habría que hacer un ***cast***).

Para crear un ***grupo*** o conjunto de botones de opción (de forma que uno y sólo uno pueda estar activado), se debe crear un objeto de la clase ***CheckboxGroup***. Este objeto no tiene datos: simplemente sirve como identificador del grupo. Cuando se crean los objetos ***Checkbox*** se pasa a los ***constructores*** el objeto ***CheckboxGroup*** del grupo al que se quiere que pertenezcan.

Cuando se selecciona un ***Checkbox*** de un grupo se producen dos eventos: uno por el elemento que se ha seleccionado y otro por haber perdido la selección el elemento que estaba seleccionado anteriormente. Al hablar de evento ***ItemEvent*** y del método ***itemStateChanged()*** se verán métodos para determinar los ***checkboxes*** que han sido seleccionados o que han perdido la selección.

5.2.18 Clase ItemEvent

Se produce un ***ItemEvent*** cuando ciertos componentes (***Checkbox***, ***CheckboxMenuItem***, ***Choice*** y ***List***) cambian de estado (on/off). Estos componentes son los que implementan la interface ***ItemSelectable***. La Tabla 5.16 muestra algunos métodos de esta clase.

Métodos de la clase ItemEvent	**Función que realizan**
Object getItem()	Devuelve el objeto donde se originó el evento
ItemSelectable getItemSelectable()	Devuelve el objeto ItemSelectable donde se originó el evento
int getStateChange()	Devuelve una de las constantes SELECTED o DESELECTED definidas en la clase ItemEvent

Tabla 5.16. Métodos de la clase ItemEvent.

La clase ***ItemEvent*** define las constantes enteras SELECTED y DESELECTED, que se pueden utilizar para comparar con el valor devuelto por el método ***getStateChange()***.

5.2.19 Clase Choice

La clase ***Choice*** permite elegir un item de una lista desplegable. Los objetos ***Choice*** ocupan menos espacio en pantalla que los ***Checkbox***. Al elegir un item se genera un ***ItemEvent***. Un ***index*** permite determinar un elemento de la lista (se empieza a contar desde 0). La Tabla 5.17 muestra los métodos más habituales de esta clase.

Métodos de Choice	**Función que realizan**
Choice()	Constructor de Choice
addItemListener(ItemListener), removeItemListener(ItemListener)	Establece o elimina un ItemListener
add(String), addItem(String)	Añade un elemento a la lista
insert(String label, int index)	Inserta un elemento con un label an la posición incicada
int getSelectedIndex(), String getSelectedItem()	Obtiene el index o el label del elemento elegido de la lista
int getItemCount()	Obtiene el número de elementos
String getItem(int)	Obtiene el label a partir del index
select(int), select(String)	Selecciona un elemento por el index o el label
removeAll(), remove(int), remove(String)	Elimina todos o uno de los elementos de la lista

Tabla 5.17. Métodos de la clase Choice.

La clase ***Choice*** genera el evento ***ItemEvent*** al seleccionar un item de la lista. En este sentido es similar a las clases ***Checkbox*** y ***CheckboxGroup***, así como a los menús de selección.

5.2.20 Clase Label

La clase ***Label*** introduce en un container un ***texto no seleccionable y no editable***, que por defecto se alinea por la izquierda. La clase ***Label*** define las constantes Label.CENTER, Label.LEFT y Label.RIGHT para determinar la alineación del texto. La Tabla 5.18 muestra algunos métodos de esta clase

Métodos de Label	**Función que realizan**
Label(String lbl), Label(String lbl, int align)	Constructores de Label
setAlignement(int align), int getAlignement()	Establecer u obtener la alineación del texto
setText(String txt), String getText()	Establecer u obtener el texto del Label

Tabla 5.18. Métodos de la clase Label.

La elección del *font*, de los *colores*, del tamaño y posición de ***Label*** se realiza con los métodos heredados de la clase ***Component***: ***setFont(Font f)***, ***setForeground(Color)***, ***setBackground(Color)***, ***setSize(int, int)***, ***setLocation(int, int)***, ***setVisible(boolean)***, etc.

La clase ***Label*** no tiene más ***eventos*** que los de su super-clase ***Component***.

5.2.21 Clase List

La clase ***List*** viene definida por una zona de pantalla con varias líneas, de las que se muestran sólo algunas, y entre las que se puede hacer una ***selección simple o múltiple***. Las ***List*** generan eventos de la clase ***ActionEvents*** (al ***clicar dos veces*** sobre un item o al pulsar ***return***) e ***ItemEvents*** (al seleccionar o deseleccionar un item). Al gestionar el evento ***ItemEvent*** se puede preguntar si el usuario estaba pulsando a la vez alguna tecla (***Alt***, ***Ctrl***, ***Shift***), por ejemplo para hacer una selección múltiple.

Las ***List*** se diferencian de las ***Choices*** en que muestran varios items a la vez y que permiten hacer selecciones múltiples. La Tabla 5.19 muestra los principales métodos de la clase ***List***.

Métodos de List	**Función que realiza**
List(), List(int nl), List(int nl, boolean mult)	Constructor: por defecto una línea y selección simple
add(String), add(String, int), addItem(String), addItem(String, int)	Añadir un item. Por defecto se añaden al final
addActionListener(ActionListener), addItemListener(ItemListener)	Registra los objetos que gestionarán los dos tipos de eventos soportados
insert(String, int)	Inserta un nuevo elemento en la lista
replaceItem(String, int)	Sustituye el item en posición int por el String
delItem(int), remove(int), remove(String), removeAll()	Eliminar uno o todos los items de la lista
int getItemCount(), int getRows()	Obtener el número de items o el número de items visibles
String getItem(int), String[] getItems()	Obtiene uno o todos los elementos de la lista
int getSelectedIndex(), String getSelectedItem(), int[] getSelectedIndexes(), String[] getSelectedItems()	Obtiene el/los elementos seleccionados
select(int), deselect(int)	Selecciona o elimina la selección de un elemento
boolean isIndexSelected(int), boolean isItemSelected(String)	Indica si un elemento está seleccionado o no
boolean isMultipleMode(), setMultipleMode(boolean)	Pregunta o establece el modo de selección múltiple
int getVisibleIndex(), makeVisible(int)	Indicar o establecer si un item es visible

Tabla 5.19. Métodos de la clase List.

5.2.22 Clase Scrollbar

Una ***Scrollbar*** es una barra de desplazamiento con un cursor que permite introducir y modificar valores, entre unos valores mínimo y máximo, con pequeños y grandes incrementos. Las ***Scrollbars*** de ***Java*** se utilizan tanto como "sliders" o barras de desplazamiento aisladas (al estilo de Visual

Basic), como unidas a una ventana en posición vertical y/u horizontal para mostrar una cantidad de información superior a la que cabe en la ventana.

La clase ***Scrollbar*** tiene dos constantes, Scrollbar.HORIZONTAL y Scrollbar.VERTICAL, que indican la posición de la barra. El cambiar el valor de la ***Scrollbar*** produce un ***AdjustementEvent***. La Tabla 5.20 muestra algunos métodos de esta clase.

Métodos de Scrollbar	**Función que realizan**
Scrollbar(), Scrollbar(int pos), Scrollbar(int pos, int val, int vis, int min, int max)	Constructores de Scrollbar
addAdjustmentListener(AdjustmentListener)	egistra el objeto que gestionará los eventos
int getValue(), setValue(int)	Permiten obtener y fijar el valor
setMaximum(int), setMinimum(int)	Establecen los valores máximo y mínimo
setVisibleAmount(int), int getVisibleAmount()	Establecen y obtienen el tamaño del área visble
setUnitIncrement(int), int getUnitIncrement()	Establecen y obtienen el incremento pequeño
setBlockIncrement(int), int getBlockIncrement()	Establecen y obtienen el incremento grande
setOrientation(int), int getOrientation()	Establecen y obtienen la orientación
setValues(int value, int vis, int min, int max)	Establecen los parámetros de la barra

Tabla 5.20. Métodos de la clase Scrollbar.

En el constructor general, el parámetro ***pos*** es la constante que indica la posición de la barra (horizontal o vertical); el ***rango*** es el intervalo entre los valores mínimo ***min*** y máximo ***max***; el parámetro ***vis*** (de ***visibleAmount***) es el tamaño del área visible en el caso en que las ***Scrollbars*** se utilicen en ***TextAreas***. En ese caso, el tamaño del cursor representa la relación entre el ***área visible*** y el ***rango***, como es habitual en ***Netscape***, ***Word*** y tantas aplicaciones de ***Windows***. El valor seleccionado viene dado por la variable ***value***. Cuando ***value*** es igual a ***min*** el área visible comprende el inicio del rango; cuando ***value*** es igual a ***max*** el área visble comprende el final del ***rango***. Cuando la ***Scrollbar*** se va a utilizar aislada (como ***slider***), se debe hacer ***visibleAmount*** igual a cero.

Las variables ***Unit Increment*** y ***Block Increment*** representan los incrementos pequeño y grande, respectivamente. Por defecto, ***Unit Increment*** es "1" y ***Block Increment*** es "10", mientras que ***min*** es "0" y ***max*** es "100".

Cada vez que cambia el valor de una ***Scrollbar*** se genera un evento ***AdjustementEvent*** y se ejecuta el único método de la interface ***AdjustmentListener***, que es ***adjustmentValueChanged()***.

5.2.23 Clase AdjustmentEvent

Se produce un evento ***AdjustementEvent*** cada vez que se cambia el valor (entero) de una ***Scrollbar***. Hay cinco tipos de ***AdjustementEvent***:

1. ***track***: se arrastra el cursor de la ***Scrollbar***.
2. ***unit increment***, ***unit decrement***: se clica en las flechas de la ***Scrollbar***.
3. ***block increment***, ***block decrement***: se clica encima o debajo del cursor.

Métodos de la clase AdjustementEvent	**Función que realizan**
Adjustable getAdjustable()	Devuelve el Component que generó el evento (implementa la interface Adjustable)
int getAdjustementType()	Devuelve el tipo de adjustement
int getValue()	Devuelve el valor de la Scrollbar después del cambio

Tabla 5.21. Métodos de la clase AdjustmentEvent.

La Tabla 5.21 muestra algunos métodos de esta clase. Las constantes UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK permiten saber el tipo de acción producida, comparando con el valor de retorno de ***getAdjustementType()***.

5.2.24 Clase ScrollPane

Un ***ScrollPane*** es como una ventana de tamaño limitado en la que se puede mostrar un componente de mayor tamaño con dos ***Scrollbars***, una horizontal y otra vertical. El componente puede ser una imagen, por ejemplo. Las ***Scrollbars*** son visibles sólo si son necesarias (por defecto). Las constantes de la clase ***ScrollPane*** son (su significado es evidente): SCROLLBARS_AS_NEEDED, SCROLLBARS_ALWAYS, SCROLLBARS_NEVER. Los ***ScrollPanes*** no generan eventos. La Tabla 5.22 muestra algunos métodos de esta clase.

Métodos de ScrollPane	**Función que realizan**
ScrollPane(), ScrollPane(int scbs)	Constructores que pueden incluir las ctes.
Dimension getViewportSize(), int getHScrollbarHeight(), int getVScrollbarWidth()	Obtiene el tamaño del ScrollPane y la altura y anchura de las barras de desplazamiento
setScrollPosition(int x, int y), setScrollPosition(Point p), Point getScrollPosition()	Permiten establecer u obtener la posición del componente
setSize(int, int), add(Component)	Heredados de Container, permiten establecer el tamaño y añadir un componente

Tabla 5.22. Métodos de la clase ScrollPane.

En el caso en que no aparezcan scrollbars (SCROLLBARS_NEVER) será necesario desplazar el componente (hacer ***scrolling***) desde programa, con el método ***setScrollPosition()***.

5.2.25 Clases TextArea y TextField

Ambas componentes heredan de la clase ***TextComponent*** y muestran texto seleccionable y editable. La diferencia principal es que ***TextField*** sólo puede tener una línea, mientras que ***TextArea*** puede tener varias líneas. Además, ***TextArea*** ofrece posibilidades de edición de texto adicionales.

Se pueden especificar el *font* y los *colores de foreground y background*. Sólo la clase ***TextField*** genera ***ActionEvents***, pero como las dos heredan de la clase ***TextComponent*** ambas pueden recibir ***TextEvents***. La Tabla 5.23 muestra algunos métodos de las clases ***TextComponent***, ***TextField*** y ***TextArea***. No se pueden crear objetos de la clase ***TextComponent*** porque su ***constructor*** no es ***public***; por eso su constructor no aparece en la Tabla 5.23.

Métodos heredados de TextComponent	**Función que realizan**
String getText() y setText(String str)	Permiten establecer u obtener el texto del componente
setEditable(boolean b), boolean isEditable()	Hace que el texto sea editable o pregunta por ello
setCaretPosition(int n), int getCaretPosition()	Fija la posición del punto de inserción o la obtiene
String getSelectedText(), int getSelectionStart() y int getSelectionEnd()	Obtiene el texto seleccionado y el comienzo y el final de la selección
selectAll(), select(int start, int end)	Selecciona todo o parte del texto
Métodos de TextField	**Función que realizan**
TextField(), TextField(int ncol), TextField(String s), TextField(String s, int ncol)	Constructores de TextField
int getColumns(), setColoumns(int)	Obtiene o establece el número de columnas del TextField
setEchoChar(char c), char getEchoChar(), boolean echoCharIsSet()	Establece, obtiene o pregunta por el carácter utilizado para passwords, de forma que no se pueda leer lo tecleado por el usuario
Métodos de TextArea	**Función que realizan**
TextArea(), TextArea(int nfil, int ncol), TextArea(String text), TextArea(String text, int nfil, int ncol)	Constructores de TextArea
setRows(int), setColumns(int), int getRows(), int getColumns()	Establecer y/u obtener los números de filas y de columnas
append(String str), insert(String str, int pos), replaceRange(String s, int i, int f)	Añadir texto al final, insertarlo en una posición determinada y reemplazar un texto determinado

Tabla 5.23. Métodos de las clases TextComponent, TextField y TextArea.

La clase ***TextComponent*** recibe eventos ***TextEvent***, y por lo tanto también los reciben sus clases derivadas ***TextField*** y ***TextAreas***. Este evento se produce cada vez que se modifica el texto del componente. La caja ***TextField*** soporta también el evento ***ActionEvent***, que se produce cada vez que el usuario termina de editar la única línea de texto pulsando ***Intro***.

Como es natural, las cajas de texto pueden recibir también los eventos de sus ***super-clases***, y más en concreto los eventos de ***Component***: ***FocusEvent***, ***MouseEvent*** y sobre todo ***KeyEvent***. Estos eventos permiten capturar las teclas pulsadas por el usuario y tomar las medidas adecuadas. Por ejemplo, si el usuario debe teclear un número en un ***TextField***, se puede crear una función que vaya capturando los caracteres tecleados y que rechace los que no sean numéricos.

Cuando se cambia desde programa el número de filas y de columnas de un ***TextField*** o ***TextArea***, hay que llamar al método ***validate()*** de la clase ***Component***, para que vuelva a aplicar el ***LayoutManager*** correspondiente. De todas formas, los tamaños fijados por el usuario tienen el carácter de "recomendaciones" o tamaños "preferidos", que el ***LayoutManager*** puede cambiar si es necesario.

5.2.26 Clase TextEvent

Se produce un ***TextEvent*** cada vez que cambia algo en un ***TextComponent*** (***TextArea*** y ***TextField***). Se puede desear evitar ciertos caracteres y para eso hay que gestionar los eventos correspondientes.

La interface ***TextListener*** tiene un único método: ***void textValueChanged(TextEvent te)***.

No hay métodos propios de la clase ***TextEvent***. Se puede utilizar el método ***Object getSource()***, que es heredado de ***EventObject***.

5.2.27 Clase KeyEvent

Se produce un ***KeyEvent*** al pulsar sobre el teclado. Como el teclado no es un componente del AWT, es el ***objeto que tiene el focus*** en ese momento quien genera los eventos ***KeyEvent*** (el objeto que es el *event source*).

Hay dos tipos de ***KeyEvents***:

1. ***key-typed***, que representa la introducción de un carácter Unicode.
2. ***key-pressed*** y ***key-released***, que representan pulsar o soltar una tecla. Son importantes para teclas que no representan caracteres, como por ejemplo F1.

Para estudiar estos eventos son muy importantes las ***Virtual KeyCodes*** (VKC). Las VKC son unas constantes que se corresponden con las ***teclas físicas*** del teclado, sin considerar minúsculas (que no tienen VKC). Se indican con el prefijo VK_, como VK_SHIFT o VK_A. La clase ***KeyEvent*** (en el package ***java.awt.event***) define constantes VKC para todas las teclas del teclado.

Por ejemplo, para escribir la letra "A" mayúscula se generan 5 eventos: *key-pressed VK_SHIFT*, *key-pressed VK_A*, *key-typed "A"*, *key-released VK_A*, y *key-released VK_SHIFT*. La Tabla 5.24 muestra algunos métodos de la clase ***KeyEvent*** y otros heredados de ***InputEvent***.

Métodos de la clase KeyEvent	**Función que realizan**
int getKeyChar()	Obtiene el carácter Unicode asociado con el evento
int getKeyCode()	Obtiene el VKC de la tecla pulsada o soltada
boolean isActionKey()	Indica si la tecla del evento es una ActionKey (HOME, END, …)
String getKeyText(int keyCode)	Devuelve un String que describe el VKC, tal como "HOME", "F1" o "A". Estos Strings se definen en el fichero awt.properties
String getKeyModifiersText(int modifiers)	Devuelve un String que describe las teclas modificadoras, tales como"Shift" o "Ctrl+Shift" (fichero awt.properties)
Métodos heredados de InputEvent	**Función que realizan**
boolean isShiftDown(), boolean isControlDown(), boolean isMetaDown(), boolean isAltDown(), int getModifiers()	Permiten identificar las teclas modificadoras

Tabla 5.24. Métodos de la clase KeyEvent.

Las constantes de ***InputEvent*** permiten identificar las teclas modificadoras y los botones del ratón. Estas constantes son SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK. A estas constantes se pueden aplicar las operaciones lógicas de bits para detectar combinaciones de teclas o pulsaciones múltiples.

5.3 MENUS

Los ***Menus*** de ***Java*** no descienden de ***Component***, sino de ***MenuComponent***, pero tienen un comportamiento similar, pues aceptan ***Events***. La Figura 5.4 muestra la jerarquía de clases de los ***Menus*** de ***Java 1.1***.

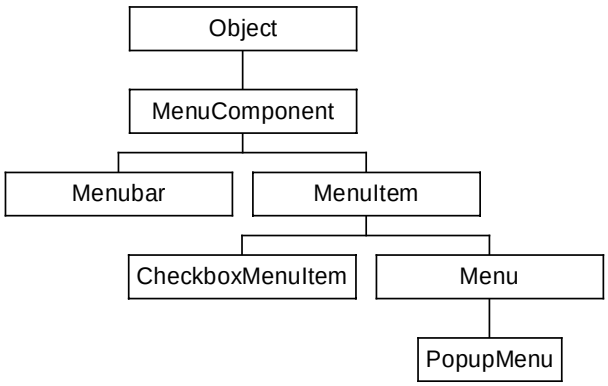

Figura 5.4. Jerarquía de clases para los menús.

Para crear un ***Menú*** se debe crear primero una ***MenuBar***; después se crean los ***Menus*** y los ***MenuItem***. Los ***MenuItems*** se añaden al ***Menu*** correspondiente; los ***Menus*** se añaden a la ***MenuBar*** y la ***MenuBar*** se añade a un ***Frame***. Una ***MenuBar*** se añade a un ***Frame*** con el método ***setMenuBar()***, de la clase ***Frame***. También puede añadirse un ***Menu*** a otro **Menu** para crear un ***sub-menú***, del modo que es habitual en ***Windows***.

La clase ***Menu*** es ***sub-clase*** de ***MenuItem***. Esto es así precisamente para permitir que un ***Menu*** sea añadido a otro ***Menu***.

5.3.1 Clase MenuShortcut

La clase ***java.awt.MenuShortcut*** (derivada de ***Object***) representa las teclas aceleradoras que pueden utilizarse para activar los menús desde teclado, sin ayuda del ratón. Se establece un ***Shortcut*** creando un objeto de esta clase con el constructor ***MenuShortcut(int vk_key)***, y pasándoselo al constructor adecuado de ***MenuItem***. Para más información, consúltese la adocumentación on-line de ***Java***. La Tabla 5.25 muestra algunos métodos de esta clase. Los ***MenuShortcut*** de ***Java*** están restringidos al uso la tecla control (CTRL). Al definir el ***MenuShortcut*** no hace falta incluir dicha tecla.

Métodos de la clase MenuShortcut	Función que realizan
MenuShortcut(int key)	Constructor
int getKey()	Obtiene el código virtual de la tecla utilizada como shortcut

Tabla 5.25. Métodos de la clase MenuShortcut.

5.3.2 Clase MenuBar

La Tabla 5.26 muestra algunos métodos de la clase ***MenuBar***. A una ***MenuBar*** sólo se pueden añadir objetos ***Menu***.

Métodos de MenuBar	Función que realizan
MenuBar()	Constructor
add(Menu), int getMenuCount(), Menu getMenu(int i)	Añade un manú, obtiene el número de menús y el menú en una posición determinada
MenuItem getShortcutMenuItem(MenuShortcut), deleteShortcut(MenuShortcut)	Obtiene el objeto MenuItem relacionado con un Shortcut y elimina el Shortcut especificado
remove(int index), remove(MenuComponent m)	Elimina un objeto Menu a partir de un índice o una referencia.
Enumeration shortcuts()	Obtiene un objeto Enumeration con todos los Shortcuts

Tabla 5.26. Métodos de la clase MenuBar.

5.3.3 Clase Menu

El objeto ***Menu*** define las opciones que aparecen al seleccionar uno de los menús de la barra de menús. En un ***Menu*** se pueden introducir objetos ***MenuItem***, otros objetos ***Menu*** (para crear sub-menús), objetos ***CheckboxMenuItem***, y ***separadores***. La Tabla 5.27 muestra algunos métodos de la clase ***Menu***. Obsérvese que como ***Menu*** desciende de ***MenuItem***, el método ***add(MenuItem)*** permite añadir objetos ***Menu*** a otro ***Menu***.

Métodos de Menu	**Función que realizan**
Menu(String)	Constructor a partir de una etiqueta
int getItemCount()	Obtener el número de items
MenuItem getItem(int)	Obtener el MenuItem a partir de un índice
add(String), add(MenuItem), addSeparator(), insertSeparator(int index)	Añadir un MenuItem o un separador
remove(int index), remove(MenuComponent), removeAll()	Eliminar uno o todos los componentes
insert(String lbl, int index), insert(MenuItem mnu, int index)	Insertar items en una posición dada
String getLabel(), setLabel(String)	Obtener y establecer las etiquetas de los items

Tabla 5.27. Métodos de la clase Menu.

5.3.4 Clase MenuItem

Los objetos de la clase ***MenuItem*** representan las distintas opciones de un menú. Al seleccionar, en la ejecución del programa, un objeto ***MenuItem*** se generan eventos del tipo ***ActionEvents***. Para cada item de un ***Menu*** se puede definir un ***ActionListener***, que define el método ***actionPerformed()***. La Tabla 5.28 muestra algunos métodos de la clase ***MenuItem***.

Métodos de MenuItem	**Función que realizan**
MenuItem(String lbl), MenuItem(String, MenuShortcut)	Constructores. El carácter (-) es el label de los separators
boolean isEnabled(), setEnabled(boolean)	Pregunta y determina si en item está activo
String getLabel(), setLabel(String)	Obtiene y establece la etiqueta del item
MenuShortcut getShortcut(), setShortcut(MenuShortcut), deleteShortcut(MenuShortcut)	Permiten obtener, establecer y borrar los MenuShortcuts
String getActionCommand(), setActionCommand(String)	Para obtener y establecer un identificador distinto del label

Tabla 5.28. Métodos de la clase MenuItem.

El método ***getActionCommand()***, asociado al ***getSource()*** del evento correspondiente, no permite identificar correctamente al ***item*** cuando éste se ha activado mediante el ***MenuShortcut*** (en ese caso devuelve ***null***).

5.3.5 Clase CheckboxMenuItem

Son ***items*** de un ***Menu*** que pueden estar activados o no activados. La clase ***CheckboxMenuItem*** no genera un ***ActionEvent***, sino un ***ItemEvent***, de modo similar a la clase ***Checkbox*** (ver Apartado 5.2.17 en la página 95). En este caso hará registrar un ***ItemListener***. La Tabla 5.29 muestra algunos métodos de esta clase.

Métodos de la clase CheckboxMenuItem	**Función que realizan**
CheckboxMenuItem(String lbl), CheckboxMenuItem(String lbl, boolean state)	Constructores
boolean getState(), setState(boolean)	Permiten obtener y establecer el estado del ChecboxMenuItem

Tabla 5.29. Métodos de la clase CheckboxMenuItem.

5.3.6 Menús pop-up

Los menús ***pop-up*** son menús que aparecen en cualquier parte de la pantalla al clicar con el botón derecho del ratón (***pop-up trigger***) sobre un componente determinado (***parent Component***). El menú ***pop-up*** se muestra en unas coordenadas relativas al ***parent Component***, que debe estar visible.

Métodos de PopupMenu	**Función que realizan**
PopupMenu(), PopupMenu(String title)	Constructores de PopupMenu:
show(Component origin, int x, int y)	Muestra el pop-up menú en la posición indicada

Tabla 5.30. Métodos de la clase PopupMenu.

Además, se pueden utilizar los métodos de la clase ***Menu*** (ver página 103), de la que deriva ***PopupMenu***. Para hacer que aparezca el ***PopupMenu*** habrá que registrar el ***MouseListener*** y definir el método ***mouseClicked()***.

5.4 LAYOUT MANAGERS

La portabilidad de ***Java*** a distintas plataformas y distintos sistemas operativos necesita flexibilidad a la hora de situar los ***Components*** (*Buttons*, *Canvas*, *TextAreas*, etc.) en un ***Container*** (*Window*, *Panel*, …). Un ***Layout Manager*** es un objeto que controla cómo los ***Components*** se sitúan en un ***Container***.

5.4.1 Concepto y Ejemplos de LayoutsManagers

El AWT define cinco ***Layout Managers***: dos muy sencillos (***FlowLayout*** y ***GridLayout***), dos más especializados (***BorderLayout*** y ***CardLayout***) y uno muy general (***GridBagLayout***). Además, los usuarios pueden escribir su propio ***Layout Manager***, implementando la interface ***LayoutManager***, que especifica 5 métodos. ***Java*** permite también posicionar los ***Components*** de ***modo absoluto***, sin ***Layout Manager***, pero de ordinario puede perderse la portabilidad y algunas otras características.

Todos los ***Containers*** tienen un ***Layout Manager*** por defecto, que se utiliza si no se indica otra cosa: Para ***Panel***, el defecto es un objeto de la clase ***FlowLayout***. Para ***Window*** (***Frame*** y ***Dialog***), el defecto es un objeto de la clase ***BorderLayout***.

La Figura 5.5 muestra un ejemplo de ***FlowLayout***: Los componentes se van añadiendo de izda a dcha y de arriba hacia abajo. Se puede elegir alineación por la izquierda, centrada o por la derecha, respecto al container.

La Figura 5.6 muestra un ejemplo de ***BorderLayout***: el container se divide en 5 zonas: ***North***, ***South***, ***East***, ***West*** y ***Center*** (que ocupa el resto de espacio).

El ejemplo de ***GridLayout*** se muestra en la Figura 5.7. Se utiliza una ***matriz de celdas*** que se numeran como se muestra en dicha figura (de izda a dcha y de arriba a abajo).

La Figura 5.8 muestra un ejemplo de uso del ***GridBagLayout***. Se utiliza también una matriz de celdas, pero permitiendo que algunos componentes ocupen más de una celda.

Finalmente, la Figura 5.9 y las dos Figuras siguientes muestran un ejemplo de ***CardLayout***. En este caso se permite que el mismo espacio sea utilizado sucesivamente por contenidos diferentes.

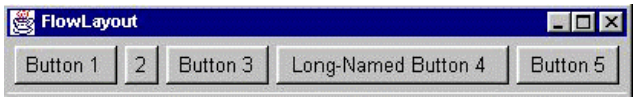

Figura 5.5. Ejemplo de FlowLayout.

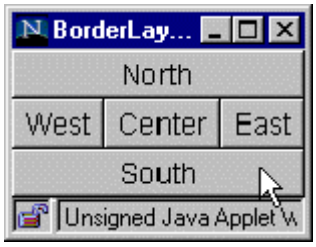

Figura 5.6. Ejemplo de BorderLayout

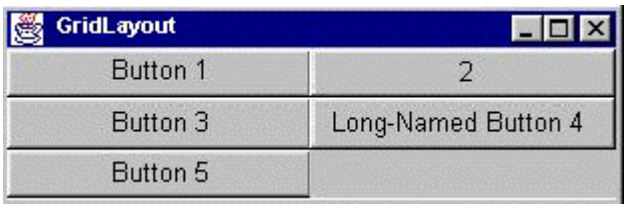

Figura 5.7. Ejemplo de GridLayout.

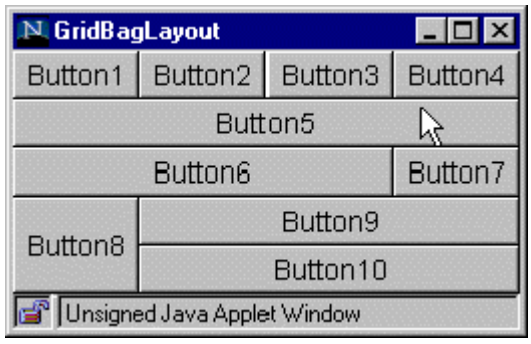

Figura 5.8. Ejemplo de GridBagLayout.

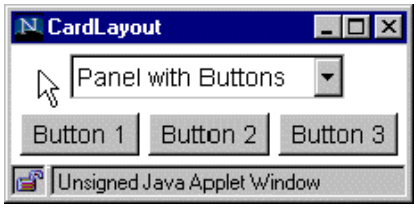

Figura 5.9. CardLayout: pantalla 1.

Figura 5.10. CardLayout: pantalla 2

Figura 5.11. CardLayout: pantalla 3.

5.4.2 Ideas generales sobre los LayoutManagers

Se debe elegir el ***Layout Manager*** que mejor se adecúe a las necesidades de la aplicación que se desea desarrollar. Recuérdese que cada ***Container*** tiene un ***Layout Manager*** por defecto. Si se desea utilizar el ***Layout Manager*** por defecto basta crear el ***Container*** (su constructor crea un objeto del ***Layout Manager*** por defecto e inicializa el ***Container*** para hacer uso de él).

Para utilizar un ***Layout Manager*** diferente hay que crear un objeto de dicho ***Layout Manager*** y pasárselo al constructor del container o decirle a dicho container que lo utilice por medio del método ***setLayout()***, en la forma:

```
unContainer.setLayout(new GridLayout());
```

La clase ***Container*** dispone de métodos para manejar el ***Layout Manager*** (ver Tabla 5.31):

Si se cambia de modo indirecto el tamaño de un ***Component*** (por ejemplo cambiando el tamaño del ***Font***), hay que llamar al método ***invalidate()*** del ***Component*** y luego al método ***validate()*** del ***Container***, lo que hace que se ejecute el método ***doLayout()*** para reajustar el espacio disponible.

Métodos de Container para manejar Layout Managers	Función que realizan
add()	Permite añadir Components a un Container
remove() y removeAll()	Permiten eliminar Components de un Container
doLayout(), validate()	doLayout() se llama automáticamente cada vez que hay que redibujar el Container y sus Components. Se llama también cuando el usuario llama al método validate()

Tabla 5.31. Métodos de Container para manejar los Layout Managers.

5.4.3 FlowLayout

FlowLayout es el ***Layout Manager*** por defecto para ***Panel***. ***FlowLayout*** coloca los componentes uno detrás deotro, en una fila, de izda a dcha y de arriba a abajo, en la misma forma en que procede un procesador de texto con las palabras de un párrafo. Los componentes se añaden en el mismo orden en que se ejecutan los métodos ***add()***. Si se cambia el tamaño de la ventana los componentes se redistribuyen de modo acorde, ocupando más filas si es necesario.

La clase ***FlowLayout*** tiene tres constructores:

```
FlowLayout();
FlowLayout(int alignement);
FlowLayout(int alignement, int horizontalGap, int verticalGap);
```

Se puede establecer la alineación de los componentes (centrados, por defecto), por medio de las constantes FlowLayout.LEFT, FlowLayout.CENTER y FlowLayout.RIGHT.

Es posible también establecer una distancia horizontal y vertical entre componentes (el ***gap***, en pixels). El valor por defecto son 5 pixels.

5.4.4 BorderLayout

BorderLayout es el ***Layout Manager*** por defecto para ***Windows*** y ***Frames***. ***BorderLayout*** define cinco áreas: ***North***, ***South***, ***East***, ***West*** y ***Center***. Si se aumenta el tamaño de la ventana todas las zonas se mantienen en su mínimo tamaño posible excepto ***Center***, que absorbe casi todo el crecimiento. Los componentes añadidos en cada zona tratan de ocupar todo el espacio disponible. Por ejemplo, si se añade un botón, el botón se hará tan grande como la celda, lo cual puede producir efectos muy extraños. Para evitar esto se puede introducir en la celda un panel con ***FlowLayout*** y añadir el botón al panel y el panel a la celda.

Los constructores de ***BorderLayout*** son los siguientes:

```
BorderLayout();
BorderLayout(int horizontalGap, int verticalGap);
```

Por defecto ***BorderLayout*** no deja espacio entre componentes. Al añadir un componente a un ***Container*** con ***BorderLayout*** se debe especificar la zona como segundo argumento:

```
miContainer.add(new Button("Norte"), "North");
```

5.4.5 GridLayout

Con ***GridLayout*** las componentes se colocan en una matriz de celdas. Todas las celdas tienen el mismo tamaño. Cada componente utiliza todo el espacio disponible en su celda, al igual que en ***BorderLayout***.

GridLayout tiene dos constructores:

```
GridLayout(int nfil, int ncol);
GridLayout(int nfil, int ncol, int horizontalGap, int verticalGap);
```

Al menos uno de los parámetros ***nfil*** y ***ncol*** debe ser distinto de cero. El valor por defecto para el espacio entre filas y columnas es cero pixels.

5.4.6 CardLayout

CardLayout permite disponer distintos componentes (de ordinario ***Panels***) que comparten la misma ventana para ser mostrados sucesivamente. Son como transparencias, diapositivas o cartas de baraja que van apareciendo una detrás de otra.

El ***orden*** de las "cartas" se puede establecer de los siguientes modos:

1. Yendo a la primera o a la última, de acuerdo con el orden en que fueron añadidas al container.
2. Recorriendo las cartas hacia delante o hacia atrás, de una en una.
3. Mostrando una carta con un nombre determinado.

Los ***constructores*** de esta clase son:

```
CardLayout()
CardLayout(int horizGap, int vertGap)
```

Para añadir componentes a un container con ***CardLayout*** se utiliza el método:

```
Container.add(Component comp, int index)
```

donde ***index*** indica la posición en que hay que insertar la carta. Los siguientes métodos de ***CardLayout*** permiten controlar el orden en que aparecen las cartas:

```
void first(Container cont);
void last(Container cont);
void previous(Container cont);
void next(Container cont);
void show(Container cont, String nameCard);
```

5.4.7 GridBagLayout

El ***GridBagLayout*** es el ***Layout Manager*** más completo y flexible, aunque también el más complicado de entender y de manejar. Al igual que el ***GridLayout***, el ***GridBagLayout*** parte de una matriz de celdas en la que se sitúan los componentes. La diferencia está en que las filas pueden tener distinta altura, las columnas pueden tener distinta anchura, y además en el ***GridBagLayout*** un componente puede ocupar varias celdas contiguas.

La posición y el tamaño de cada componente se especifican por medio de unas “restricciones” o ***constraints***. Las restricciones se establecen creando un objeto de la clase ***GridBagConstraints***, dando valor a sus propiedades (variables miembro) y asociando ese objeto con el componente por medio del método ***setConstraints()***.

Las ***variables miembro*** de ***GridBagConstraints*** son las siguientes:

- ***gridx*** y ***gridy***. Especifican la fila y la columna en la que situar la esquina superior izquierda del componente (se empieza a contar de cero). Con la constante GridBagConstraints.RELATIVE se indica que el componente se sitúa relativamente al anterior componente situado (es la condición por defecto).

- ***gridwidth*** y ***gridheight***. Determinan el número de columnas y de filas que va a ocupar el componente. El valor por defecto es una columna y una fila. La constante GridBagConstraints.REMAINDER indica que el componente es el último de la columna o de la fila, mientras que GridBagConstraints.RELATIVE indica que el componente se sitúa respecto al anterior componente de la fila o columna.
- ***fill***. En el caso en que el componente sea más pequeño que el espacio reservado, esta variable indica si debe ocupar o no todo el espacio disponible. Los posibles valores son: GridBagConstraints.NONE (no lo ocupa; defecto), GridBagConstraints.HORIZONTAL (lo ocupa en dirección horizontal), GridBagConstraints.VERTICAL (lo ocupa en vertical) y GridBagConstraints.BOTH (lo ocupa en ambas direcciones).
- ***ipadx*** y ***ipady***. Especifican el espacio a añadir en cada dirección al tamaño interno del componente. Los valores por defecto son cero. El tamaño del componente será el tamaño mínimo más dos veces el ***ipadx*** o el ***ipady***.
- ***insets***. Indican el espacio mínimo entre el componente y el espacio disponible. Se establece con un objeto de la clase ***java.awt.Insets***. Por defecto es cero.
- ***anchor***. Se utiliza para determinar dónde se coloca el componente, cuando éste es menor que el espacio disponible. Sus posibles valores vienen dados por las constantes de la clase ***GridBagConstraints***: CENTER (el valor por defecto), NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST y NORTHWEST.
- ***weightx*** y ***weighty***. Son unos coeficientes entre 0.0 y 1.0 que sirven para dar más o menos "peso" a las distintas filas y columnas. A más "peso" más probabilidades tienen de que se les dé más anchura o más altura.

A continuación se muestra una forma típica de crear un container con ***GridBagLayout*** y de añadirle componentes:

```
GridBagLayout unGBL = new GridBagLayout();
GridBagConstraints unasConstr = new GridBagConstraints();
unContainer.setLayout(unGBL);

// Ahora ya se pueden añadir los componentes
//...Se crea un componente unComp
//...Se da valor a las variables del objeto unasConstr
// Se asocian las restricciones con el componente
unGBL.setConstraints(unComp, unasConstr);
// Se añade el componente al container
unContainer.add(unComp);
```

5.5 GRÁFICOS, TEXTO E IMÁGENES

En esta parte final del AWT se van a describir, también muy sucintamente, algunas clases y métodos para realizar dibujos y añadir texto e imágenes a la interface gráfica de usuario.

5.5.1 Capacidades gráficas del AWT: Métodos paint(), repaint() y update()

La clase ***Component*** tiene tres métodos muy importantes relacionados con gráficos: ***paint()***, ***repaint()*** y ***update()***. Cuando el usuario llama al método ***repaint()*** de un componente, el AWT llama al método ***update()*** de ese componente, que por defecto llama al método ***paint()***.

5.5.1.1 Método paint(Graphics g)

El método ***paint()*** está definido en la clase ***Component***, pero ese método no hace nada y hay que redefinirlo en una de sus clases derivadas. El programador no tiene que preocuparse de llamar a este método: el sistema operativo lo llama al dibujar por primera vez una ventana, y luego lo vuelve a llamar cada vez que entiende que la ventana o una parte de la ventana debe ser re-dibujada (por ejemplo, por haber estado tapada por otra ventana y quedar de nuevo a la vista).

5.5.1.2 Método update(Graphics g)

El método ***update()*** hace dos cosas: primero re-dibuja la ventana con el color de fondo y luego llama al método ***paint()***. Este método también es llamado por el AWT, y también puede ser llamado por el programador, quizás porque ha realizado algún cambio en la ventana y necesita que se dibuje de nuevo.

La propia estructura de este método -el comenzar pintando de nuevo con el color de fondo- hace que se produzca ***parpadeo*** (***flicker***) en las animaciones. Una de las formas de evitar este efecto es redefinir este método de una forma diferente, cambiando de una imagen a otra sólo lo que haya que cambiar, en vez de re-dibujar todo otra vez desde el principio. Este método no siempre proporciona los resultados buscados y hay que recurrrir al método del ***doble buffer***.

5.5.1.3 Método repaint()

Este es el método que con más frecuencia es llamado por el programador. El método ***repaint()*** llama "lo antes posible" al método ***update()*** del componente. Se puede también especificar un número de milisegundos para que el método ***update()*** se llame transcurrido ese tiempo. El método ***repaint()*** tiene las cuatro formas siguientes:

```
repaint()
repaint(long time)
repaint(int x, int y, int w, int h)
repaint(long time, int x, int y, int w, int h)
```

Las formas tercera y cuarta permiten definir una ***zona rectangular*** de la ventana a la que aplicar el método.

5.5.2 Clase Graphics

El único argumento de los métodos ***update()*** y ***paint()*** es un objeto de esta clase. La clase ***Graphics*** dispone de métodos para soportar dos tipos de gráficos:

1. Dibujo de ***primitivas gráficas*** (*texto, líneas, círculos, rectángulos, polígonos, …*).
2. Presentación de ***imágenes*** en formatos ****.gif*** y ****.jpeg***.

Además, la clase ***Graphics*** mantiene un ***contexto gráfico***: un área de dibujo actual, un color de dibujo del background y otro del foreground, un font con todas sus propiedades, etc. La Figura 5.12 muestra el sistema de coordenadas utilizado en ***Java***. Como es habitual en Informática, los ejes

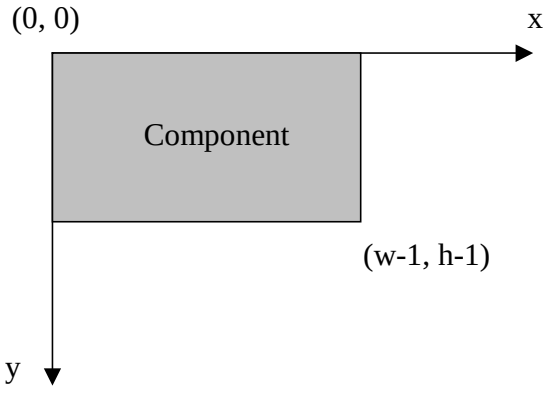

Figura 5.12. Coordenadas de los gráficos de Java.

están situados en la esquina superior izquierda, con la orientación indicada en la Figura 5.12 (eje de ordenadas descendente). Las coordenadas se miden siempre en ***pixels***.

5.5.3 Primitivas gráficas

Java dispone de métodos para realizar dibujos sencillos, llamados a veces "primitivas" gráficas. Como se ha dicho, las coordenadas se miden en ***pixels***, empezando a contar desde cero. La clase ***Graphics*** dispone de los métodos para primitivas gráficas reseñados en la Tabla 5.32.

Excepto los polígonos y las líneas, todas las formas geométricas se determinan por el rectángulo que las comprende, cuyas dimensiones son ***w*** y ***h***. Los polígonos admiten un argumento de la clase ***java.awt.Polygon***.

Método gráfico	**Función que realizan**
drawLine(int x1, int y1, int x2, int y2)	Dibuja una línea entre dos puntos
drawRect(int x1, int y1, int w, int h)	Dibuja un rectángulo (w-1, h-1)
fillRect(int x1, int y1, int w, int h)	Dibuja un rectángulo y lo rellena con el color actual
clearRect(int x1, int y1, int w, int h)	Borra dibujando con el background color
draw3DRect(int x1, int y1, int w, int h, boolean raised)	Dibuja un rectángulo resaltado (w+1, h+1)
fill3DRect(int x1, int y1, int w, int h, boolean raised)	Rellena un rectángulo resaltado (w+1, h+1)
drawRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Dibuja un rectángulo redondeado
fillRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Rellena un rectángulo redondeado
drawOval(int x1, int y1, int w, int h)	Dibuja una elipse
fillOval(int x1, int y1, int w, int h)	Dibuja una elipse y la rellena de un color
drawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Dibuja un arco de elipse (ángulos en grados)
fillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Rellena un arco de elipse
drawPolygon(int x[], int y[], int nPoints)	Dibuja y cierra el polígono de modo automático
drawPolyline(int x[], int y[], int nPoints)	Dibuja un polígono pero no lo cierra
fillPolygon(int x[], int y[], int nPoints)	Rellena un polígono

Tabla 5.32. Métodos de la clase Graphics para dibujo de primitivas gráficas.

Los métodos ***draw3DRect()***, ***fill3DRect()***, ***drawOval()***, ***fillOval()***, ***drawArc()*** y ***fillArc()*** dibujan objetos cuyo tamaño total es (w+1, h+1) pixels.

5.5.4 Clases Graphics y Font

La clase ***Graphics*** permite "dibujar" texto, como alternativa al texto mostrado en los componentes ***Label***, ***TextField*** y ***TextArea***. Los métodos de esta clase para dibujar texto son los siguientes:

```
drawBytes(byte data[], int offset, int length, int x, int y);
drawChars(char data[], int offset, int length, int x, int y);
drawString(String str, int x, int y);
```

En estos métodos, los argumentos ***x*** e ***y*** representan las coordenadas de la ***línea base*** (ver Figura 5.13). El argumento ***offset*** indica el elemento del array que se empieza a imprimir.

Cada tipo de letra está representado por un objeto de la clase ***Font***. Las clases ***Component*** y ***Graphics*** disponen de métodos ***setFont()*** y ***getFont()***. El constructor de ***Font*** tiene la forma:

```
Font(String name, int style, int size)
```

donde el ***style*** se puede definir con las constantes Font.PLAIN, Font.BOLD y Font.ITALIC. Estas constantes se pueden combinar en la forma: Font.BOLD | Font.ITALIC.

La clase ***Font*** tiene tres variables *protected*, llamadas ***name***, ***style*** y ***size***. Además tiene tres constantes enteras: PLAIN, BOLD e ITALIC. Esta clase dispone de los métodos ***String getName()***, ***int getStyle()***, ***int getSize()***, ***boolean isPlain()***, ***boolean isBold()*** y ***boolean isItalic()***, cuyo significado es inmediato.

Figura 5.13. Líneas importantes en un tipo de letra.

Para mayor portabilidad se recomienda utilizar nombres lógicos de fonts, tales como ***Serif*** (*Times New Roman*), ***SansSerif*** (*Arial*) y ***Monospaced*** (*Courier*).

5.5.5 Clase FontMetrics

La clase ***FontMetrics*** permite obtener información sobre una ***font*** y sobre el espacio que ocupa un ***char*** o un ***String*** utilizando esa ***font***. Esto es muy útil cuando se pretende rotular algo de modo que quede siempre centrado y bien dimensionado.

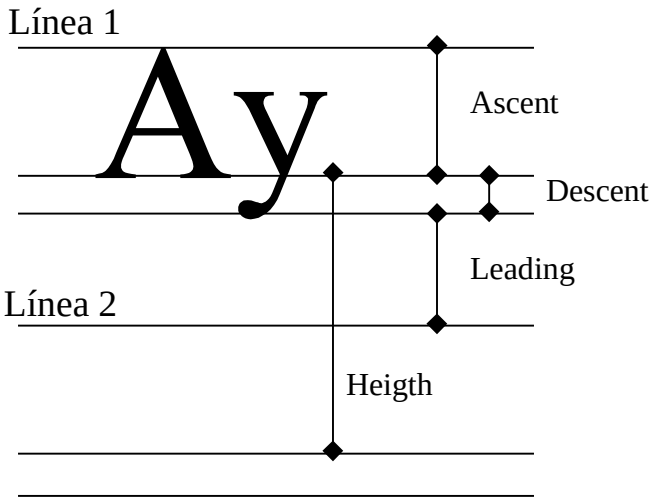

Figura 5.14. Nomenclatura para la clase FontMetrics.

La clase ***FontMetrics*** es una clase ***abstract***. Esto quiere decir que no se pueden crear directamente objetos de esta clase ni llamar a su constuctor. La forma habitual de soslayar esta dificultad es creando una subclase. En la práctica ***Java*** resuelve esta dificultad para el usuario, ya que la clase ***FontMetrics*** tiene como variable miembro un objeto de la clase ***Font***. Por ello, un objeto de la clase ***FontMetrics*** contiene información sobre la ***font*** que se le ha pasado como argumento al constructor. De todas formas, el camino más habitual para obtener esa información es a partir de un objeto de la clase ***Graphics*** que ya tiene un ***font*** definido. A partir de un objeto ***g*** de la clase ***Graphics*** se puede obtener una referencia ***FontMetrics*** en la forma:

```
FontMetrics miFontMet = g.getFontMetrics();
```

donde está claro que se está utilizando una referencia de la clase *abstract* ***FontMetrics*** para refererirse a un objeto de una clase derivada creada dentro del API de ***Java***. Con una referencia de tipo ***FontMetrics*** se pueden utilizar todos los métodos propios de dicha clase.

La Tabla 5.33 muestra algunos métodos de la clase ***FontMetrics***, para los que se debe tener en cuenta la terminología introducida en la Figura 5.14.

Métodos de la clase FontMetrics	**Función que realizan**
FontMetrics(Font font)	Constructor
int getAscent(), int getMaxAscent()	Permiten obtener el "Ascent" actual y máximo para esa font
int getDescent(), int getMaxDescent()	Permiten obtener el "Descent" actual y máximo para esa font
int getHeight(), int getLeading()	Permiten ontener la distancia entre líneas y la distancia entre el descender de una línea y el ascender de la siguiente
int getMaxAdvance()	Da la máxima anchura de un carácter de esa font, incluyendo el espacio hasta el siguiente carácter
int charWidth(int ch), int charWidth(char ch), int stringWidth(String str)	Dan la anchura de un carácter (incluyendo el espacio hasta el siguiente carácter) o de toda una cadena de caracteres
int charsWidth(char data[], int start, int len), int bytesWidth(byte data[], int start, int len)	Dan la anchura de un array de caracteres o de bytes. Permiten definir la posición del comienzo y el número de caracteres

Tabla 5.33. Métodos de la clase FontMetrics.

5.5.6 Clase Color

La clase ***java.awt.Color*** encapsula colores utilizando el formato RGB (Red, Green, Blue). Las componentes de cada color primario en el color resultante se expresan con números enteros entre 0 y 255, siendo 0 la intensidad mínima de ese color, y 255 la máxima.

En la clase ***Color*** existen constantes para colores predeterminados de uso frecuente: ***black***, ***white***, ***green***, ***blue***, ***red***, ***yellow***, ***magenta***, ***cyan***, ***orange***, ***pink***, ***gray***, ***darkGray***, ***lightGray***. La Tabla 5.34 muestra algunos métodos de la clase ***Color***.

Metodos de la clase Color	**Función que realizan**
Color(int), Color(int,int,int), Color(float,float,float)	Constructores de Color, con tres bytes en un int (del bit 0 al 23), con enteros entre 0 y 255 y float entre 0.0 y 1.0
Color brighter(), Color darker()	Obtienen una versión más o menos brillante de un color
Color getColor(), int getRGB()	Obtiene un color en los tres primeros bytes de un int
int getGreen(), int getRed(), int getBlue()	Obtienen las componentes de un color
Color getHSBColor()	Obtiene un color a partir de los valores de "hue", "saturation" y "brightness" (entre 0.0 y 1.0)
float[] RGBtoHSB(int,int,int,float[]), int HSBtoRGB(float,float,float)	Métodos static para convertir colores de un sistema de definición de colores a otro

Tabla 5.34. Métodos de la clase Color.

5.5.7 Imágenes

Java permite incorporar imágenes de tipo GIF y JPEG definidas en ficheros. Se dispone para ello de la clase ***java.awt.Image***. Para cargar una imagen hay que indicar la localización del fichero (URL) y cargarlo mediante los métodos ***Image getImage(String)*** o ***Image getImage(URL, String)***. Estos métodos existen en las clases ***java.awt.Toolkit*** y ***java.applet.Applet***. El argumento de tipo ***String*** representa una variable conteniendo el nombre del fichero.

Cuando estas imágenes se cargan en ***applets***, para obtener el URL pueden ser útiles las funciones ***getDocumentBase()*** y ***getCodeBase()***, que devuelven el URL del fichero HTML que llama al ***applet***, y el directorio que contiene el ***applet*** (en forma de ***String***).

Para cargar una imagen hay que comenzar creando un objeto ***Image***, y llamar al método ***getImage()***, pasándole como argumento el URL. Por ejemplo:

```
Image miImagen = getImage(getCodeBase(), "imagen.gif")
```

Una vez cargada la imagen, hay que representarla, para lo cual se redefine el método ***paint()*** para llamar al método ***drawImage()*** de la clase ***Graphics***. Dicho método admite varias formas, aunque casi siempre hay que incluir el nombre del objeto imagen creado, las dimensiones de dicha imagen y un objeto ***ImageObserver***.

ImageObserver es una interface que declara métodos para observar el estado de la carga y visualización de la imagen. Si se está programando un ***applet***, basta con poner como ***ImageObserver*** la referencia ***this***, ya que en la mayoría de los casos, la implementación de esta interface en la clase ***Applet*** proporciona el comportamiento deseado. Para más información sobre dicho método dirigirse a la referencia de la API.

La clase ***Image*** define ciertas constantes para controlar los algoritmos de cambio de escala:

SCALE_DEFAULT, SCALE_FAST, SCALE_SMOOTH,

SCALE_REPLICATE, SCALE_AVERAGE.

La Tabla 5.35 muestra algunos métodos de la clase ***Image***.

Métodos de la clase Image	**Función que realizan**
Image()	Constructor
int getWidth(ImageObserver) int getHeight(ImageObserver)	Determinan la anchura y la altura de la imagen. Si no se conocen todavía, este método devuelve -1 y el objeto ImageObserver especificado será notificado más tarde
Graphics getGraphics()	Crea un contexto gráfico para poder dibujar en una imagen no visible en pantalla. Este método sólo se puede llamar para objetos no visibles en pantalla
Object getProperty(String, ImageObserver)	Obtiene una propiedad de una imagen a partir del nombre de la propiedad
Image getScaledInstance(int w, int h, int hints)	Crea una versión de la imagen a otra escala. Si w o h son negativas se utiliza la otra dimensión manteniendo la proporción. El último argumento es información para el algoritmo de cambio de escala

Tabla 5.35. Métodos de la clase Image.

5.6 ANIMACIONES

Las animaciones tienen un gran interés desde diversos puntos de vista. Una imagen vale más que mil palabras y una imagen en movimiento es todavía mucho más útil. Para presentar o describir ciertos conceptos el movimiento animado es fundamental. Además, las animaciones o mejor dicho, la forma de hacer animaciones en ***Java*** ilustran mucho la forma en que dicho lenguaje realiza los gráficos. En estos apartados se va a seguir el esquema del ***Tutorial*** de ***Sun*** sobre el AWT.

Se pueden hacer animaciones de una forma muy sencilla: se define el método ***paint()*** de forma que cada vez que sea llamado dibuje algo diferente de lo que ha dibujado la vez anterior. De todas formas, recuérdese que el programador no llama directamente a este método. El programador llama al método ***repaint()***, quizás dentro de un bucle ***while*** que incluya una llamada al método ***sleep()*** de

la clase ***Thread*** (ver Capítulo 6, a partir de la página 116), para esperar un cierto número de milisegundos entre dibujo y dibujo (entre *frame* y *frame*, utilizando la terminología de las animaciones). Recuérdese que ***repaint()*** llama a ***update()*** lo antes posible, y que ***update()*** borra todo redibujando con el color de fondo y llama a ***paint()***.

La forma de proceder descrita da buenos resultados para animaciones muy sencillas, pero produce ***parpadeo*** o ***flicker*** cuando los gráficos son un poco más complicados. La razón está en el propio proceso descrito anteriormente, combinado con la velocidad de refresco del monitor. La velocidad de refresco vertical de un monitor suele estar entre 60 y 75 herzios. Eso quiere decir que la imagen se actualiza unas 60 ó 75 veces por segundo. Cuando el refresco se realiza después de haber borrado la imagen anterior pintando con el color de fondo y antes de que se termine de dibujar de nuevo toda la imagen, se obtiene una imagen incompleta, que sólo aparecerá terminada en uno de los siguientes pasos de refresco del monitor. Ésta es la causa del ***flicker***. A continuación se verán dos formas de reducirlo o eliminarlo.

5.6.1 Eliminación del parpadeo o flicker redefiniendo el método update()

El problema del ***flicker*** se localiza en la llamada al método ***update()***, que borra todo pintando con el color de fondo y después llama a ***paint()***. Una forma de resolver esta dificultad es ***re-definir*** el método ***update()***, de forma que se adapte mejor al problema que se trata de resolver.

Una posibilidad es no re-pintar todo con el color de fondo, no llamar a ***paint()*** e introducir en ***update()*** el código encargado de realizar los dibujos, cambiando sólo aquello que haya que cambiar. A pesar de esto, es necesario re-definir ***paint()*** pues es el método que se llama de forma automática cuando la ventana de ***Java*** es tapada por otra que luego se retira. Una posible solución es hacer que ***paint()*** llame a ***update()***, terminando por establecer un orden de llamadas opuesto al de defecto. Hay que tener en cuenta que, al no borrar todo pintando con el color de fondo, el programador tiene que preocuparse de borrar de forma selectiva entre *frame* y *frame* lo que sea necesario. Los métodos ***setClip()*** y ***clipRect()*** de la clase ***Graphics*** permiten hacer que las operaciones gráficas no surtan efecto fuera de un área rectangular previamente determinada. Al ser dependiente del tipo de gráficos concretos de que se trate, este método no siempre proporciona soluciones adecuadas.

5.6.2 Técnica del doble buffer

JGJLa técnica del ***doble buffer*** proporciona la mejor solución para el problema de las animaciones, aunque requiere una programación algo más complicada. La idea básica del ***doble buffer*** es realizar los dibujos en una imagen invisible, distinta de la que se está viendo en la pantalla, y hacerla visible cuando se ha terminado de dibujar, de forma que aparezca instantáneamente.

Para crear el segundo buffer o imagen invisible hay que crear un objeto de la clase ***Image*** del mismo tamaño que la imagen que se está viendo y crear un contexto gráfico u objeto de la clase ***Graphics*** que permita dibujar sobre la imagen invisible. Esto se hace con las sentencias,

```
Image imgInv;
Graphics graphInv;
Dimension dimInv;
Dimension d = size(); // se obtiene la dimensión del panel
```

en la clase que controle el dibujo (por ejemplo en una clase que derive de ***Panel***). En el método ***update()*** se modifica el código de modo que primero se dibuje en la imagen invisible y luego ésta se haga visible: