MANUAL DE FORMACIÓN EN JAVA

PLAN DE FORMACIÓN DEL CENTRO DE FORMACIÓN DEL PROFESORADO DE SEVILLA.
IES POLÍGONO SUR

Centro de Profesorado de SEVILLA

Índice

Introducción al lenguaje Java

En el aprendizaje de todo lenguaje de programación, el primer paso suele ser entrar en conocimiento de los conceptos fundamentales, como son las variables, los tipos de datos, expresiones, flujo de control, etc. Por ello, durante varias secciones se estudiarán de estos conceptos. Lo básico resultará muy familiar a quienes tengan conocimientos de C/C++. Los programadores con experiencia en otros lenguajes procedurales reconocerán la mayor parte de las construcciones.

Cuando se programa en Java, se coloca todo el código en métodos, de la misma forma que se escriben funciones en lenguajes como C.

Comentarios

En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea

/*
 comentarios de una o más líneas
*/

/** comentario de documentación, de una o más líneas
 */
```

Los dos primeros tipos de comentarios son los que todo programador conoce y se utilizan del mismo modo. Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java, , no disponible en otros lenguajes de programación. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación de las clases que se va construyendo al mismo tiempo que se genera el código de la aplicación.

En este tipo de comentario para documentación, se permite la introducción de algunos tokens o palabras clave, que harán que la información que les sigue aparezca de forma diferente al resto, permitiendo la incorporación de información útil, que luego se podrá ver en formato html sobre cualquier navegador. Aunque posteriormente se verán en detalle las palabras clave que soporta **javadoc**, hay que tener en cuenta a la hora de utilizar este tipo de comentarios, que javadoc solamente procesará la documentación para miembros public y protected, los comentarios para miembros private y friendly, que se verán en secciones posteriores, serán ignorados (aunque se puede utilizar el parámetro -private para incluir los miembros privados en la salida). Esto tiene bastante sentido, ya que solamente los miembros public y protected están disponibles fuera del fichero, que es el punto de vista del programador externo que utiliza la documentación. Sin embargo, todos los comentarios que se incluyan en class figurarán en la salida.

El JDK 1.x permite la creación de doclets, que son pequeñas aplicaciones que la herramienta javadoc puede ejecutar para generar cualquier tipo de documentación especial que se desee. No obstante, la salida estándar es un fichero html con el mismo formato que el resto de la documentación de Java, por lo que resulta muy fácil navegar a través de las clases. Y quizá sea el momento de ver algunos de los tokens que se pueden introducir en

estos comentarios de documentación para que javadoc pueda generar la documentación del código que se está construyendo, porque la herramienta en sí no se va a estudiar en profundidad.

Javadoc permite también colocar códigos o tags de html embebidos en los comentarios. Estos códigos son procesados también a la hora de generar el documento html. Esta característica permite utilizar todo el poder del lenguaje html, aunque su utilización principal es solamente para formatear un poco la salida que se genera, como por ejemplo:

```
/**
 * <pre>
 * System.out.println( new Date() );
 * </pre>
 */
```

También se puede utilizar html como en cualquier otro documento Web para formatear el texto en descripciones, como por ejemplo:

```
/**
 * Aquí <em>debes</em> introducir la lista:
 * <ol>
 * <li> Primer Elemento
 * <li> Segundo Elemento
 * <li> Tercer Elemento
 * </ol>
 */
```

Dentro de los comentarios de documentación, los asteriscos al comienzo de las líneas y los espacios hasta la primera letra no se tienen en cuenta. La salida del ejemplo anterior sería:

```
Aqui debes introducir la lista:
  1. Primer Elemento
  2. Segundo Elemento
  3. Tercer Elemento
```

Javadoc lo reformatea todo para acomodarlo a las normas html. No se debería utilizar cabeceras como <h1> o separadores como <hr> ya que javadoc inserta los suyos propios y se podría interferir con ellos. Todos los tipos de documentación, sean de clases, variables o métodos, pueden soportar html embebido. Además se pueden colocar tokens específicos de javadoc, que son los que van precedidos por el símbolo @, y a continuación se van a ver los más interesantes.

@see: Referencia a otras clases

Permite referenciar a la documentación de otras clases. Se puede colocar en los tres tipos de documentación. javadoc convertirá estos tags en enlaces a la otra documentación. Las formas que se permiten colocar como referencia son:

```
@see nombre-de-clase
@see nombre-de-clase-completo
@see nombre-de-clase-completo#nombre-de-método
```

Cada una de ellas añade un enlace en la entrada "See Also" de la documentación generada. Hay que tener en cuenta que javadoc no chequea los enlaces para comprobar que sean válidos.

@version: Información de la versión

Como parámetro se puede indicar cualquier información que sea relevante para la versión que se está generando. Cuando el flag -version se indica en la lista de argumentos de llamada de javadoc, esta información aparecerá especialmente resaltada en la documentación html. Se utiliza en la documentación de clases e interfaces (que se verán en secciones posteriores de este Tutorial).

@author: Información del autor

Normalmente se suele indicar el nombre del autor y su dirección de correo electrónico, aunque se puede incluir cualquier otra información relevante. Cuando se utiliza -author como argumento de javadoc, esta información aparecerá especialmente resaltada en la documentación html. Se utiliza en la documentación de clases e interfaces.

Se pueden colocar múltiples tags de este tipo para indicar una lista de autores, en cuyo caso deben estar colocados consecutivamente, ya que toda la información sobre la autoría del código se colocará en un solo párrafo de la documentación html generada.

@param: Parámetro y descripción

Se utiliza en la documentación de métodos y permite la documentación de los parámetros. El formato consiste en indicar un parámetro y en las líneas siguientes colocar la descripción, que se considera concluida cuando aparece otro tag de documentación. Se suele colocar un tag @param por cada parámetro.

@return: Descripción

Se utiliza en la documentación de métodos para indicar el significado del valor de retorno del método. La descripción se puede extender las líneas que sean necesarias.

@exception: Nombre de la clase y descripción

Las excepciones, que se verán en un capítulo posterior, son objetos que pueden ser generados por un método cuando este método falla. Aunque solamente se puede producir una excepción cuando se llama a un método, un método puede generar cualquier número de excepciones diferentes.

El nombre de la clase debe ser completo para que no haya ninguna ambigüedad con ninguna otra excepción y en la descripción se debería indicar el porqué puede aparecer esa excepción cuando se llama al método. Se utiliza en la documentación de métodos.

@deprecated

Se utiliza en la documentación de métodos para anunciar que la característica que indica el método está obsoleta. Este tag es un estímulo para que la característica sobre la que se coloca no se utilice más, ya que en un futuro puede desaparecer. Los métodos marcados con este tag hacen que el compilador genere avisos, aunque se produzca código perfectamente ejecutable.

El ejemplo , muestra la fecha del sistema cuando se ejecuta y se han colocado tags de documentación, para ser utilizados posteriormente con javadoc.

```
import java.util.*;

/**
 * java400.java: Ejemplo de documentacion.
 * Presenta la fecha y hora del Sistema
 * @author Agustin Froufe
 * @author froufe@arrakis.es
 * @version 1.0
 */

public class java400 {
   /**
    * Método principal de la aplicacion
    * @param args cadena de argumentos de llamada
    * @return No devuelve ningun valor
    * @exception excepciones No lanza ninguna
    */
    public static void main( String args[] ) {
        System.out.println( new Date() );
        }
    }
```

Si ahora se ejecuta javadoc sobre este fichero con el comando:

```
% javadoc -version -author java400.java
```

se generan varios ficheros de documentación, o se actualizan si ya existían, como son los ficheros AllNames.html y packages.html, y se genera el fichero de documentación de la clase.

Identificadores

Los identificadores nombran variables, funciones, clases y objetos; cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de dólar ($). Los siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay una longitud máxima establecida para el identificador. La forma básica de una declaración de variable, por ejemplo, sería:

```
tipo identificador [ = valor][,identificador [= valor] ...];
```

Serían identificadores válidos:

```
identificador
nombre_usuario
Nombre_Usuario
_variable_del_sistema
$transaccion
```

y su uso sería, por ejemplo:

```
int contador_principal;
char _lista_de_ficheros;
float $cantidad_en_Ptas;
```

Palabras Clave

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como identificadores:

```
abstract   continue   for          new          switch
boolean    default    goto         null         synchronized
break      do         if           package      this
byte       double     implements   private      threadsafe
byvalue    else       import       protected    throw
case       extends    instanceof   public       transient
catch      false      int          return       true
char       final      interface    short        try
class      finally    long         static       void
const      float      native       super        while
```

Palabras Reservadas

Además, el lenguaje se reserva unas cuantas palabras más, pero que hasta ahora no tienen un cometido específico. Son:

```
cast       future     generic      inner
operator   outer      rest         var
```

Literales

Un valor constante en Java se crea utilizando una representación literal de él. Java utiliza cinco tipos de elementos: enteros, reales en coma flotante, booleanos, caracteres y cadenas, que se pueden poner en cualquier lugar del código fuente de Java. Cada uno de estos literales tiene un tipo correspondiente asociado con él.

Enteros:

byte 8 bits complemento a dos

short 16 bits complemento a dos

int 32 bits complemento a dos

long 64 bits complemento a dos

Por ejemplo: 2 21 077 0xDC00

Reales en coma flotante:

float 32 bits IEEE 754

double 64 bits IEEE 754

Por ejemplo: 3.14 2e12 3.1E12

Booleanos:

true

false

Caracteres:

Por ejemplo: a \t \u???? [????] número unicode

Cadenas:

Por ejemplo: "Esto es una cadena literal"

Cuando se inserta un literal en un programa, el compilador normalmente sabe exactamente de qué tipo se trata. Sin embargo, hay ocasiones en la que el tipo es ambiguo y hay que guiar al compilador proporcionándole información adicional para indicarle exactamente de qué tipo son los caracteres que componen el literal que se va a encontrar. En el ejemplo siguiente se muestran algunos casos en que resulta imprescindible indicar al compilador el tipo de información que se le está proporcionando:

```
class literales {
    char c = 0xffff;     // mayor valor de char en hexadecimal
    byte b = 0x7f;       // mayor valor de byte en hexadecimal
    short s = 0x7fff;    // mayor valor de short en hexadecimal
    int i1 = 0x2f;       // hexadecimal en minúsculas
    int i2 = 0X2F;       // hexadecimal en mayúsculas
    int i3 = 0177;       // octal (un cero al principio)
    long l1 = 100L;
    long l2 = 100l;
    long l3 = 200;
    // long l4(200);     // no está permitido
    float f1 = 1;
    float f2 = 1F;
    float f3 = 1f;
    float f4 = 1e-45f;  // en base 10
    float f5 = 1e+9f;
    double d1 = 1d;
    double d2 = 1D;
    double d3 = 47e47d; // en base 10
    }
```

Un valor hexadecimal (base 16), que funciona con todos los tipos enteros de datos, se indica mediante un 0x o 0X seguido por 0-9 y a-f, bien en mayúsculas o minúsculas. Si se intenta inicializar una variable con un valor mayor que el que puede almacenar, el compilador generará un error. Por ejemplo, si se exceden los valores para char, byte y short que se indican en el ejemplo, el compilador automáticamente convertirá el valor a un int e indicará que se necesita un moldeo estrecho para la asignación.

Los números octales (base 8), se indican colocando un cero a la izquierda del número que se desee. No hay representación para números binarios ni en en C, ni en C++, ni en Java.

Se puede colocar un carácter al final del literal para indicar su tipo, ya sea una letra mayúscula o minúscula. L se usa para indicar un long, F significa float y una D mayúscula o minúscula es lo que se emplea para indicar que el literal es un double.

La exponenciación se indica con la letra e, tomando como referente la base 10. Es decir, que hay que realizar una traslación mental al ver estos números de tal forma que 1.3e-45f en Java, en la realidad es 1.3 x 10-45.

No es necesario indicarle nada al compilador cuando se puede conocer el tipo sin ambiguedades. Por ejemplo, en

```
long l3 = 200;
```

no es necesario colocar la L después de 200 porque resultaría superflua. Sin embargo, en el caso:

```
float f4 = 1e-45f; // en base 10
```

sí es necesario indicar el tipo, porque el compilador trata normalmente los números exponenciales como double, por lo tanto, si no se coloca la f final, el compilador generará un error indicando que se debe colocar un moldeador para convertir el double en float.

Cuando se realizan operaciones matemáticas o a nivel de bits con tipos de datos básicos más pequeños que int (char, byte o short), esos valores son promocionados a int antes de realizar las operaciones y el resultado es de tipo int. Si se quiere seguir teniendo el tipo de dato original, hay que colocar un moldeo, teniendo en cuenta que al pasar de un tipo de dato mayor a uno menor, es decir, al hacer un moldeo estrecho, se puede perder información. En general, el tipo más grande en una expresión es el que determina el tamaño del resultado de la expresión; si se multiplica un float por un double, el resultado será un double, y si se suma un int y un long, el resultado será un long.

Separadores

Sólo hay un par de secuencias con otros caracteres que pueden aparecer en el código Java; son los separadores simples, que van a definir la forma y función del código. Los separadores admitidos en Java son:

() - **paréntesis**. Para contener listas de parámetros en la definición y llamada a métodos. También se utiliza para definir precedencia en expresiones, contener expresiones para control de flujo y rodear las conversiones de tipo.

{} - **llaves**. Para contener los valores de matrices inicializadas automáticamente. También se utiliza para definir un bloque de código, para clases, métodos y ámbitos locales.

[] - **corchetes**. Para declarar tipos matriz. También se utiliza cuando se referencian valores de matriz.

; - **punto y coma**. Separa sentencias.

, - **coma**. Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia for.

. - **punto**. Para separar nombres de paquete de subpaquetes y clases. También se utiliza para separar una variable o método de una variable de referencia.

Operadores

Los operadores de Java son muy parecidos en estilo y funcionamiento a los de C. Tanto C, como C++, como Java, proporcionan un conjunto de operadores para poder realizar acciones sobre uno o dos operandos. Un operador que actúa sobre un solo operando es un operador unario, y un operador que actúa sobre dos operandos es un operador binario.

Algunos operadores pueden funcionar como unarios y como binarios, el ejemplo más claro es el operador - (signo menos). Como operador binario, el signo menos hace que

el operando de la derecha sea sustraido al operando de la izquierda; como operador unario hace que el signo algebraico del operando que se encuentre a su derecha sea cambiado.

En la siguiente tabla aparecen los operadores que se utilizan en Java, por orden de precedencia:

```
.        []       ()
++       --
!        ~        instanceof
*        /        %
+        -
<<       >>       >>>
<        >        <=        >=       ==       !=
&        ^        |
&&       ||
?  :
=        op=      (*=       /=       %=       +=     -=    etc.) ,
```

Los operadores numéricos se comportan como esperamos:

```
int + int = int
```

Los operadores relacionales devuelven un valor booleano.

Para las cadenas, se pueden utilizar los operadores relacionales para comparaciones además de + y += para la concatenación:

```
String nombre = "nombre" + "Apellido";
```

El operador = siempre hace copias de objetos, marcando los antiguos para borrarlos, y ya se encargará el garbage collector de devolver al sistema la memoria ocupada por el objeto eliminado.

Java, a diferencia de C++, no soporta la sobrecarga de operadores. Esto significa que no es posible redefinir el entorno en el que actúa un operador con respecto a los objetos de un nuevo tipo que el programador haya definido como propios.

Un caso interesante, que se sale de la afirmación anterior, es el operador + (signo más), que se puede utilizar para realizar una suma aritmética o para concatenar cadenas (¡Java lo sobrecarga internamente!). Cuando el signo más se utiliza en esta última forma, el operando de la derecha se convierte automáticamente en una cadena de caracteres antes de ser concatenada con el operando que se encuentra a la izquierda del operador +. Esto asume que el compilador sabe que el operando de la derecha es capaz de soportar la conversión. Y ese conocimiento lo tiene porque comprueba todos los tipos primitivos y muchos de los tipos internos que se utilizan. Obviamente, el compilador no sabe absolutamente nada de los tipos que haya definido el programador.

A continuación se detallan algunos de los operadores que admite Java y que se suelen agrupar en operadores aritméticos, relacionales y condicionales, operadores lógicos y que actúan sobre bits y, finalmente, operadores de asignación.

Operadores Aritméticos

Java soporta varios operadores aritméticos que actúan sobre números enteros y números en coma flotante. Los operadores binarios soportados por Java son:

+ suma los operandos

- resta el operando de la derecha al de la izquierda

* multiplica los operandos

/ divide el operando de la izquierda entre el de la derecha

% resto de la división del operando izquierdo entre el derecho

Como se ha indicado anteriormente, el operador más (+), se puede utilizar para concatenar cadenas, como se observa en el ejemplo siguiente:

```
"miVariable tiene el valor " + miVariable + " en este programa"
```

Esta operación hace que se tome la representación como cadena del valor de miVariable para su uso exclusivo en la expresión. De ninguna forma se altera el valor que contiene la variable.

El operador módulo (%), que devuelve el resto de una división, a diferencia de C++, en Java funciona con tipos en coma flotante además de con tipos enteros. Cuando se ejecuta el programa que se muestra en el ejemplo .

La salida que se obtiene por pantalla tras la ejecución del ejemplo es la que se reproduce a continuación:

```
%java java401
x mod 10 = 3
y mod 10 = 3.299999999999997
```

Los operadores unarios que soporta Java son:

+ indica un valor positivo

- negativo, o cambia el signo algebraico

++ suma 1 al operando, como prefijo o sufijo

-- resta 1 al operando, como prefijo o sufijo

En los operadores de incremento (++) y decremento (--), en la versión prefijo, el operando aparece a la derecha del operador, ++x; mientras que en la versión sufijo, el operando aparece a la izquierda del operador, x++. La diferencia entre estas versiones es el momento en el tiempo en que se realiza la operación representada por el operador si éste y su operando aparecen en una expresión larga. Con la versión prefijo, la variable se incrementa (o decrementa) antes de que sea utilizada para evaluar la expresión en que se encuentre, mientras que en la versión sufijo, se utiliza la variable para realizar la evaluación de la expresión y luego se incrementa (o decrementa) en una unidad su valor.

Operadores Relacionales y Condicionales

Los operadores relacionales en Java devuelven un tipo booleano, true o false.

> el operando izquierdo es mayor que el derecho

>= el operando izquierdo es mayor o igual que el derecho

< el operando izquierdo es menor que el derecho

<= el operando izquierdo es menor o igual que el derecho

== el operando izquierdo es igual que el derecho

!= el operando izquierdo es distinto del derecho

Los operadores relacionales combinados con los operadores condicionales, se utilizan para obtener expresiones más complejas. Los operadores condicionales que soporta Java son:

&& expresiones izquierda y derecha son true

|| o la expresión izquierda o al expresión de la derecha son true

! la expresión de la derecha es false

Una característica importante del funcionamiento de los operadores && y ||, tanto en Java como en C++ (y que es pasado a veces por alto, incluso por programadores experimentados) es que las expresiones se evalúan de izquierda a derecha y que la evaluación de la expresión finaliza tan pronto como se pueda determinar el valor de la expresión. Por ejemplo, en la expresión siguiente:

```
( a < b ) || ( c < d )
```

si la variable a es menor que la variable b, no hay necesidad de evaluar el operando izquierdo del operador || para determinar el valor de la expresión entera. En casos de complicadas, complejas y largas expresiones, el orden en que se realizan estas comprobaciones puede ser fundamental, y cualquier error en la colocación de los operandos puede dar al traste con la evaluación que se desea realizar y, estos errores son harto difíciles de detectar, ya que se debe estudiar concienzudamente el resultado de las expresiones en tiempo de ejecución para poder detectar el problema.

Operadores a Nivel de Bits

Java y C, C++ comparten un conjunto de operadores que realizan operaciones sobre un solo bit cada vez. Java también soporta el operador >>>, que no existe en C o C++ y, además, el entorno en el que se realizan algunas de las operaciones no siempre es el mismo en Java que en los otros lenguajes.

Los operadores de bits que soporta Java son los que aparecen en la siguiente tablita:

Operador	Uso	Operación
>>	Operando >> Despl	Desplaza bits del operando hacia la derecha las posiciones indicadas (con signo)
<<	Operando << Despl	Desplaza bits del operando hacia la izquierda las posiciones indicadas
>>>	Operando >>> Despl	Desplaza bits del operando hacia la derecha las posiciones indicadas (sin signo)
&	Operando & Operando	Realiza una operación AND lógiga entre los dos operandos
|	Operando | Operando	Realiza una operación OR lógica entre los dos operandos
^	Operando ^ Operando	Realiza una operación lógica OR Exclusiva entre los dos operandos
~	~Operando	Complementario del operando (unario)
Operador	Uso	Operación
>>	Operando >> Despl	Desplaza bits del operando hacia la derecha las posiciones indicadas (con signo)
<<	Operando << Despl	Desplaza bits del operando hacia la izquierda las posiciones indicadas
>>>	Operando >>> Despl	Desplaza bits del operando hacia la derecha las posiciones indicadas (sin signo)
&	Operando & Operando	Realiza una operación AND lógiga entre los dos operandos
|	Operando | Operando	Realiza una operación OR lógica entre los dos operandos
^	Operando ^ Operando	Realiza una operación lógica OR Exclusiva entre los dos operandos
~	~Operando	Complementario del operando (unario)

En Java, el operador de desplazamiento hacia la derecha sin signo, rellena los bits que pueden quedar vacíos con ceros. Los bits que son desplazados fuera del entorno se pierden.

En el desplazamiento a la izquierda, hay que ser precavidos cuando se trata de desplazar enteros positivos pequeños, porque el desplazamiento a la izquierda tiene el efecto de multiplicar por 2 para cada posición de bit que se desplace; y esto es peligroso porque si se desplaza un bit 1 a la posición más alta, el bit 31 o el 63, el valor se convertirá en negativo.

Operadores de Asignación

El operador = es un operador binario de asignación de valores. El valor almacenado en la memoria y representado por el operando situado a la derecha del operador es copiado

en la memoria indicada por el operando de la izquierda. El operador de asignación (ni ningún otro) no se puede sobrecargar en Java.

Java soporta toda la panoplia de operadores de asignación que se componen con otros operadores para realizar la operación que indique ese operador y luego asignar el valor obtenido al operando situado a la izquierda del operador de asignación. De este modo se pueden realizar dos operaciones con un solo operador.

```
+=    -=    *=    /=    %=    &=    |=    ^=    <<=    >>=   >>>=
```

Por ejemplo, las dos sentencias que siguen realizan la misma función:

```
x += y;
x = x + y;
```

Operadores Ternario if-then-else

Java, lo mismo que C y C++, soporta este operador ternario. No obstante, la construcción utilizada por este operador es algo confusa, aunque se puede llegar a entender perfectamente cuando uno lee en el pensamiento lo que está escrito en el código. La forma general del operador es:

```
expresion ? sentencia1 : sentencia2
```

en donde expresion puede ser cualquier expresión de la que se obtenga como resultado un valor booleano, porque en Java todas las expresiones condicionales se evalúan a booleano; y si es true entonces se ejecuta la sentencia1 y en caso contrario se ejecuta la sentencia2. La limitación que impone el operador es que sentencia1 y sentencia2 deben devolver el mismo tipo, y éste no puede ser void.

Puede resultar útil para evaluar algún valor que seleccione una expresión a utilizar, como en la siguiente sentencia:

```
cociente = denominador == 0 ? 0 : numerador / denominador
```

Cuando Java evalúa la asignación, primero mira la expresión que está a la izquierda del interrogante. Si denominador es cero, entonces evalúa la expresión que está entre el interrogante y los dos puntos, y se utiliza como valor de la expresión completa. Si denominador no es cero, entonces evalúa la expresión que está después de los dos puntos y se utiliza el resultado como valor de la expresión completa, que se asigna a la variable que está a la izquierda del operador de asignación, cociente.

En el ejemplo se utiliza este operador para comprobar que el denominador no es cero antes de evaluar la expresión que divide por él, devolviendo un cero en caso contrario. Hay dos expresiones, una que tiene un denominados cero y otra que no.

```
class java402 {
    public static void main( String args[] ) {
        int a = 28;
        int b = 4;
        int c = 45;
        int d = 0;

        // Utilizamos el operador ternario para asignar valores a
        // las dos variables e y f, que son resultado de la
        // evaluación realizada por el operador
        int e = (b == 0) ? 0 : (a / b);
        int f = (d == 0) ? 0 : (c / d);
```

```
        // int f = c / d;
        System.out.println( "a = " + a );
        System.out.println( "b = " + b );
        System.out.println( "c = " + c );
        System.out.println( "d = " + d );
        System.out.println();
        System.out.println( "a / b = " + e );
        System.out.println( "c / d = " + f );
        }
    }
```

El programa se ejecuta sin errores, y la salida que genera por pantalla es la que se reproduce:

```
%java java402
a = 28
b = 4
c = 45
d = 0

a / b = 7
c / d = 0
```

Si ahora cambiamos la línea que asigna un valor a la variable f, y quitamos este operador, dejándola como:

```
int f = c / d;
```

se producirá una excepción en tiempo de ejecución de división por cero, deteniéndose la ejecución del programa en esa línea.

```
%java java402
java.lang.ArithmeticException:/ by zero at java402.main(java402.java:40)
```

Errores comunes en el uso de Operadores

Uno de los errores más comunes que se cometen cuando se utilizan operadores es el uso incorrecto, o no uso, de los paréntesis cuando la expresión que se va a evaluar es compleja. Y esto sigue siendo cierto en Java.

Cuando se programa en C/C++, el error más común es el siguiente:

```
while( x = y ) {
    // . . .
    }
```

EL programador intenta realmente hacer una equivalencia (==) en vez de una asignación. En C/C++ el resultado de esta asignación siempre será true si y no es cero, y lo más seguro es que esto se convierta en un bucle infinito. En Java, el resultado de esta expresión no es un booleano, y el compilador espera un booleano, que no puede conseguir a partir de un entero, así que aparecerá un error en tiempo de compilación que evitará que el programa se caiga al ejecutarlo. Es decir, este error tan común, para que se dé en Java tienen que ser x e y booleanos, lo cual no es lo más usual.

Otro error muy difícil de detectar en C++ es cuando se utiliza el operador unario & cuando la lógica requiere un operador &&; o utilizar el operador unario |, en lugar del operador ||. Este es otro problema difícil de detectar porque el resultado es casi siempre el mismo, aunque sólo por accidente. En Java, no ocurre esto y se puede utilizar perfectamente el operador unario & como sinónimo del operador &&, o el operador | como

sinónimo de ||; pero si se hace así, la evaluación de la expresión no concluirá hasta que todos los operandos hayan sido evaluados, perdiendo de esta forma las posibles ventajas que podría reportar la evaluación de izquierda a derecha.

Moldeo de Operadores

Es lo que se conoce como casting, refiriéndose a "colocar un molde". Java automáticamente cambia el tipo de un dato a otro cuando es pertinente. Por ejemplo, si se asigna un valor entero a una variable declarada como flotante, el compilador convierte automáticamente el int a float. El casting, o moldeo, permite hacer esto explícitamente, o forzarlo cuando normalmente no se haría.

Para realizar un moldeo, se coloca el tipo de dato que se desea (incluyendo todos los modificadores) dentro de paréntesis a la izquierda del valor. Por ejemplo:

```
void moldeos() {
    int i = 100;
    long l = (long)i;
    long l2 = (long)200;
    }
```

Como se puede ver, es posible realizar el moldeo de un valor numérico del mismo modo que el de una variable. No obstante, en el ejemplo el moldeo es superfluo porque el compilador ya promociona los enteros a flotantes cuando es necesario automáticamente, sin necesidad de que se le indique. Aunque hay otras ocasiones en que es imprescindible colocar el moldeo para que el código compile.

En Java, el moldeo es seguro, con la excepción de cuando se realiza una conversión estrecha, es decir, cuando se quiere pasar de un dato que contiene mucha información a otro que no puede contener tanta, en donde se corre el riesgo de perder información. Aquí es donde el compilador fuerza a que se coloque un moldeo expreso, indicando al programador que "esto puede ser algo peligroso, así que si quieres que lo haga, dímelo expresamente". Cuando se trata de una conversión ancha, no es necesario el moldeo explícito, ya que el nuevo tipo podrá contener más información que la que contiene el tipo original, no ocasionándose ninguna pérdida de información, así que el compilador puede realizar el moldeo explícitamente, sin la intervención del programador para confirmar la acción.

Java permite el moldeo de cualquier tipo primitivo en otro tipo primitivo, excepto en el caso de los booleanos, en que no se permite moldeo alguno. Tampoco se permite el moldeo de clases. Para convertir una clase en otra hay que utilizar métodos específicos. La clase String es un caso especial que se verá más adelante.

Variables, expresiones y Arrays

Variables

Las variables se utilizan en la programación Java para almacenar datos que varían durante la ejecución del programa. Para usar una variable, hay que indicarle al compilador el tipo y nombre de esa variable, declaración de la variable. El tipo de la variable determinará el conjunto de valores que se podrán almacenar en la variable y el tipo de operaciones que se podrán realizar con ella.

Por ejemplo, el tipo int solamente puede contener números completos (enteros). En Java, todas las variables de tipo int contienen valores con signo.

Expresiones

Los programas en Java, al igual que en C y C++, se componen de sentencias, que a su vez están compuestas en base a expresiones. Una expresión es una determinada combinación de operadores y operandos que se evalúan para obtener un resultado particular. Los operandos pueden ser variables, constantes o llamadas a métodos.

Una llamada a un método evalúa el valor devuelto por el método y el tipo de una llamada a un método es el tipo devuelto por ese método.

Java soporta constantes con nombre y la forma de crearlas es:

```
final float PI = 3.14159;
```

Esta línea de código produce un valor que se puede referenciar en el programa, pero no puede ser modificado. La palabra clave final es la que evita que esto suceda.

En ciertos casos, el orden en que se realizan las operaciones es determinante para el resultado que se obtenga. Al igual que en C++, Pascal y otros lenguajes, en Java se puede controlar el orden de evaluación de las expresiones mediante el uso de paréntesis. Si no se proporcionan estos paréntesis, el orden estará determinado por la precedencia de operadores, de tal modo que las operaciones en las que estén involucrados operadores con más alta precedencia serán los que primero se evalúen.

Arrays

Java dispone de un tipo array.

En Java, al ser un tipo de datos verdadero, se dispone de comprobaciones exhaustivas del correcto manejo del array; por ejemplo, de la comprobación de sobrepasar los límites definidos para el array, en evitación de desbordamiento o corrupción de memoria.

En Java hay que declarar un array antes de poder utilizarlo. Y en la declaración hay que incluir el nombre del array y el tipo de datos que se van a almacenar en él. La sintaxis general para declarar e instanciar un array es:

```
tipoDeElementos[] nombreDelArray = new tipoDeElemetos[tamañoDelArray];
```

Se pueden declarar en Java arrays de cualquier tipo:

```
char s[];
int iArray[];
```

Incluso se pueden construir arrays de arrays:

```
int tabla[][] = new int[4][5];
```

Al igual que los demás objetos en Java, la declaración del array no localiza, o reserva, memoria para contener los datos. En su lugar, simplemente localiza memoria para almacenar una referencia al array. La memoria necesaria para almacenar los datos que componen el array se buscará en memoria dinámica a la hora de instanciar y crear realmente el array.

Para crear un array en Java hay dos métodos básicos. Crear un array vacío:

```
int lista[] = new int[50];
```

O se puede crear ya el array con sus valores iniciales:

```
String nombres[] = {
    "Juan","Pepe","Pedro","Maria"
    };
```

Esto que es equivalente a:

```
String nombres[];
nombres = new String[4];
nombres[0] = new String( "Juan" );
nombres[1] = new String( "Pepe" );
nombres[2] = new String( "Pedro" );
nombres[3] = new String( "Maria" );
```

No se pueden crear arrays estáticos en tiempo de compilación:

```
int lista[50]; // generará un error en tiempo
               // de compilación
```

Tampoco se puede rellenar un array sin declarar el tamaño con el operador new:

```
int lista[];
for( int i=0; i < 9; i++ )
    lista[i] = i;
```

En las sentencias anteriores simultáneamente se declara el nombre del array y se reserva memoria para contener sus datos. Sin embargo, no es necesario combinar estos procesos. Se puede ejecutar la sentencia de declaración del array y posteriormente, otra sentencia para asignar valores a los datos.

Una vez que se ha instanciado un array, se puede acceder a los elementos de ese array utilizando un índice, de forma similar a la que se accede en otros lenguajes de programación. Sin embargo, Java no permite que se acceda a los elementos de un array utilizando punteros.

```
    miArray[5] = 6;
    miVariable = miArray[5];
```

Como en C y C++, los índices de un array siempre empiezan en 0.

Todos los arrays en Java tienen una función miembro: length(), que se puede utilizar para conocer la longitud del array.

```
int a[][] = new int[10][3];
a.length;    /* 10 */
a[0].length; /* 3 */
```

Además, en Java no se puede rellenar un array sin declarar el tamaño con el operador new. El siguiente código no sería válido:

```
int lista[];
for( int i=0; i < 9; i++ )
    lista[i] = i;
```

Es decir, todos los arrays en Java son estáticos. Para convertir un array en el equivalente a un array dinámico en C/C++, se usa la clase Vector, que permite operaciones de inserción, borrado, etc. en el array.

El ejemplo , intenta ilustrar un aspecto interesante del uso de arrays en Java. Se trata de que Java puede crear arrays multidimensionales, que se verían como arrays de

arrays. Aunque esta es una característica común a muchos lenguajes de programación, en Java, sin embargo, los arrays secundarios no necesitan ser todos del mismo tamaño. En el ejemplo, se declara e instancia un array de enteros, de dos dimensiones, con un tamaño inicial (tamaño de la primera dimensión) de 3. Los tamaños de las dimensiones secundarias (tamaño de cada uno de los subarrays) es 2, 3 y 4, respectivamente.

En este ejemplo, también se pueden observar alguna otra cosa, como es el hecho de que cuando se declara un array de dos dimensiones, no es necesario indicar el tamaño de la dimensión secundaria a la hora de la declaración del array, sino que puede declararse posteriormente el tamaño de cada uno de los subarrays. También se puede observar el resultado de acceder a elementos que se encuentran fuera de los límites del array; Java protege la aplicación contra este tipo de errores de programación.

En C/C++, cuando se intenta acceder y almacenar datos en un elemento del array que está fuera de límites, siempre se consigue, aunque los datos se almacenen en una zona de memoria que no forme parte del array. En Java, si se intenta hacer esto, el Sistema generará una excepción, tal como se muestra en el ejemplo. En él, la excepción simplemente hace que el programa termine, pero se podría recoger esa excepción e implementar un controlador de excepciones (exception handler) para corregir automáticamente el error, sin necesidad de abortar la ejecución del programa.

La salida por pantalla de la ejecución del ejemplo sería:

```
%java java403
00
012
0246
Acceso a un elemento fuera de limites
java.lang.ArrayIndexOutOfBoundsException:atjava403.main(java403.java:55)
```

El ejemplo , ilustra otro aspecto del uso de arrays en Java. Se trata, en este caso, de que Java permite la asignación de un array a otro. Sin embargo, hay que extremar las precauciones cuando se hace esto, porque lo que en realidad está pasando es que se está haciendo una copia de la referencia a los mismos datos en memoria. Y, tener dos referencias a los mismos datos no parece ser una buena idea, porque los resultados pueden despistar.

La salida por pantalla que se genera tras la ejecución del ejemplo es:

```
%java java404
Contenido del primerArray
0 1 2
Contenido del segundoArray
0 1 2
--> Cambiamos un valor en el primerArray
Contenido del primerArray
0 10 2
Contenido del segundoArray
0 10 2
```

Strings

Un String o cadena se considera a toda secuencia de caracteres almacenados en memoria y accesibles como una unidad. C/C++ no soportan un tipo String, sino que simplemente disponen de un formato de almacenamiento de caracteres que se puede tratar

como un String. Sin embargo, Java implementa cadenas a través de las clases String y StringBuffer, a las que se dedicará un amplio estudio.

Control de flujo

El control del flujo es la manera que tiene un lenguaje de programación de provocar que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos. Java, en este aspecto, no utiliza los principios de diseño orientado a objetos, sino que las sentencias de control del flujo del programa se han tomado del C/C++. A continuación se tratan todos los mecanismos que proporciona Java para conseguir este control y decidir qué partes del código ejecutar.

Sentencias de Salto

if/else

```
if( expresión-booleana ) {
    sentencias;
}
[else {
    sentencias;
}]
```

Esta construcción provoca que la ejecución atraviese un conjunto de estados booleanos que determinan que se ejecuten distintos fragmentos de código. La cláusula else es opcional. Cada una de las sentencias puede ser una sentencia compuesta y la expresión-booleana podría ser una variable simple declarada como boolean, o una expresión que utilice operadores relacionales para generar el resultado de una comparación.

En el ejemplo , se utilizan varias sentencias if-else para determinar en qué estación del año se encuentra un mes determinado.

Cuando se ejecuta el programa, la salida que se obtienes en pantalla es la que indica la estación en la que se encuentra el mes en que se está escribiendo el código de esta aplicación:

```
%java java405
Agosto esta en el verano.
```

Switch

```
switch( expresión ) {
case valor1:
    sentencias;
    break;
case valor2:
    sentencias;
    break;
[default:
    sentencias;]
}
```

La sentencia switch proporciona una forma limpia de enviar la ejecución a partes diferentes del código en base al valor de una única variable o expresión. La expresión puede devolver cualquier tipo básico, y cada uno de los valores especificados en las sentencias case debe ser de un tipo compatible.

La sentencia switch funciona de la siguiente manera: el valor de la expresión se compara con cada uno de los valores literales de las sentencias case. Si coincide con alguno, se ejecuta el código que sigue a la sentencia case. Si no coincide con ninguno de ellos, entonces se ejecuta la sentencia default (por defecto), que es opcional. Si no hay sentencia default y no coincide con ninguno de los valores, no hace nada. Al igual que en otros lenguajes, cada constante en sentencia case debe ser única.

El compilador de Java inspeccionará cada uno de los valores que pueda tomar la expresión en base a las sentencias case que se proporcionen, y creará una tabla eficiente que utiliza para ramificar el control del flujo al case adecuado dependiendo del valor que tome la expresión. Por lo tanto, si se necesita seleccionar entre un gran grupo de valores, una sentencia switch se ejecutará mucho más rápido que la lógica equivalente codificada utilizado sentencias if-else.

La palabra clave break se utiliza habitualmente en sentencias switch sin etiqueta, para que la ejecución salte tras el final de la sentencia switch. Si no se pone el break, la ejecución continuará en el siguiente case. Es un error habitual a la hora de programar el olvidar un break; dado que el compilador no avisa de dichas omisiones, es una buena idea poner un comentario en los sitios en los que normalmente se pondría el break, diciendo que la intención es que el case continúe en el siguiente, por convenio este comentario es simplemente, "Continúa". Otra diferencia de Java con C++ respecto a la sentencia switch es que Java soporta sentencias break etiquetadas. Esta característica puede hacer que el entorno de funcionamiento sea muy diferente entre ambos lenguajes cuando hay sentencias switch anidadas.

El ejemplo es el mismo ejemplo que se creó en la sección if-else, reescrito para utilizar switch.

Sentencias de Bucle

Bucles for

```
for( inicialización; terminación; iteración ) {
   sentencias;
   }
```

Un bucle for, normalmente involucra a tres acciones en su ejecución:

- Inicialización de la variable de control
- Comprobación del valor de la variable de control en una expresión condicional
- Actualización de la variable de control

La cláusula de inicio y la cláusula de incremento pueden estar compuestas por varias expresiones separadas mediante el operador coma (,), que en estos bucles Java también soporta.

```
for( a=0,b=0; a < 7; a++,b+=2 )
```

El operador coma garantiza que el operando de su izquierda se ejecutará antes que el operando de su derecha. Las expresiones de la cláusula de inicio se ejecutan una sola vez, cuando arranca el bucle. Cualquier expresión legal se puede emplear en esta cláusula, aunque generalmente se utiliza para inicialización. Las variables se pueden declarar e inicializar al mismo tiempo en esta cláusula:

```
for( int cnt=0; cnt < 2; cnt++ )
```

La segunda cláusula, de test, consiste en una única expresión que debe evaluarse a false para que el bucle concluya. En este caso, Java es mucho más restrictivo que C++, ya que en C, C++ cualquier expresión se puede evaluar a cero, que equivale a false. Sin embargo, en Java, esta segunda expresión debe ser de tipo booleano, de tal modo que se pueden utilizar únicamente expresiones relacionales o expresiones relacionales y condicionales.

El valor de la segunda cláusula es comprobado cuando la sentencia comienza la ejecución y en cada una de las iteraciones posteriores.

La tercera cláusula, de incremento, aunque aparece físicamente en la declaración de bucle, no se ejecuta hasta que se han ejecutado todas las sentencias que componen el cuerpo del bucle for; por ello, se utiliza para actualizar la variable de control. Es importante tener en cuenta que si utilizamos variables incluidas en esta tercera cláusula en las sentencias del cuerpo del bucle, su valor no se actualizará hasta que la ejecución de todas y cada una de las sentencias del cuerpo del bucle se haya completado. En esta cláusula pueden aparecer múltiples expresiones separadas por el operador coma, que serán ejecutadas de izquierda a derecha.

El siguiente trocito de código Java que dibuja varias líneas en pantalla alternando sus colores entre rojo, azul y verde, utiliza un bucle for para dibujar un número determinado de líneas y una sentencia switch para decidir el color de la línea. Este fragmento sería parte de una función Java (método):

```
int contador;
for( contador=1; contador <= 12; contador++ ) {
    switch( contador % 3 ) {
        case 0:
            setColor( Color.red );
            break;
        case 1:
            setColor( Color.blue );
            break;
        case 2:
            setColor( Color.green );
            break;
        }
    g.drawLine( 10,contador*10,80,contador*10 );
    }
```

También se soporta, como ya se ha indicado, el operador coma (,) en los bucles for, aunque su uso es una decisión de estilo, no es la única forma de codificar una sentencia lógica en particular. En ocasiones se utiliza como atajo, pero en otras se prefiere la utilización de sentencias múltiples dentro del cuerpo del bucle for.

```
for( a=0,b=0; a < 7; a++,b+=2 )
```

No se pueden definir variables con el mismo nombre en dos sitios diferentes de un método. El programa , no compila, generando un error que indica que la variable está declarada en el método cuando se intenta declararla en el bucle for. La salida de compilación es la que se muestra

```
C:\>javac java407.java
java407.java:32: Variable 'cnt' is already defined in this method.
for( int cnt=0; cnt < 2; cnt++ )
```

```
^
1 error
```

La primera y tercera cláusulas del bucle for pueden encontrarse vacías, pero deben estar separadas por punto y coma (;). Hay autores que sugieren incluso que la cláusula de testeo puede estar vacía, aunque para el programador que está escribiendo esto, salvando el caso de que se trate de implementar un bucle infinito, si esta cláusula de comprobación se encuentra vacía, el método de terminación del bucle no es nada obvio, al no haber una expresión condicional que evaluar, por lo que debería recurrirse a otro tipo de sentencia, en vez de utilizar un bucle for.

Bucles while

```
[inicialización;]
   while( terminación-expresión-booleana ) {
       sentencias;
       [iteración;]
       }
```

El bucle while es la sentencia de bucle más básica en Java. Ejecuta repetidamente una vez tras otra una sentencia, mientras una expresión booleana sea verdadera. Las partes de inicialización e iteración, que se presentan entre corchetes, son opcionales.

Esta sentencia while se utiliza para crear una condición de entrada. El significado de esta condición de entrada es que la expresión condicional que controla el bucle se comprueba antes de ejecutar cualquiera de las sentencias que se encuentran situadas en el interior del bucle, de tal modo que si esta comprobación es false la primera vez, el conjunto de las sentencias no se ejecutará nunca.

Un ejemplo típico de utilización de este bucle es el cálculo de la serie de números de Fibonacci, que es una de las formas de conseguir la serie, y eso es precisamente lo que hace el ejemplo , cuyo código se muestra a continuación:

```
class java408 {
    public static void main( String args[] ){
        int max = 20;
        int bajo = 1;
        int alto = 0;

        System.out.println( bajo );
        while( alto < 50 ) {
            System.out.println( alto );
            int temp = alto;
            alto = alto + bajo;
            bajo = temp;
            }
        }
    }
```

Bucles do/while

```
    [inicialización;]
    do {
        sentencias;
        [iteración;]
    }while( terminación-expresión-booleana );
```

A veces se puede desear el ejecutar el cuerpo de un bucle while al menos una vez, incluso si la expresión booleana de terminación tiene el valor false la primera vez. Es decir, si se desea evaluar la expresión de terminación al final del bucle en vez de al principio como en el bucle while. Esta construcción do-while hace eso exactamente.

Excepciones

try-catch-throw

```
try {
    sentencias;
} catch( Exception ) {
    sentencias;
}
```

Java implementa excepciones para facilitar la construcción de código robusto. Cuando ocurre un error en un programa, el código que encuentra el error lanza una excepción, que se puede capturar y recuperarse de ella. Java proporciona muchas excepciones predefinidas.

El manejo y control de excepciones es tan importante en Java que debe ser tratado en un capítulo aparte, aunque sirva este punto como mención previa de su existencia.

Control General del Flujo

break

```
break [etiqueta];
```

La sentencia break puede utilizarse en una sentencia switch o en un bucle. Cuando se encuentra en una sentencia switch, break hace que el control del flujo del programa pase a la siguiente sentencia que se encuentre fuera del entorno del switch. Si se encuentra en un bucle, hace que el flujo de ejecución del programa deje el ámbito del bucle y pase a la siguiente sentencia que venga a continuación del bucle.

Java incorpora la posibilidad de etiquetar la sentencia break, de forma que el control pasa a sentencias que no se encuentran inmediatamente después de la sentencia switch o del bucle, es decir, saltará a la sentencia en donde se encuentre situada la etiqueta. La sintaxis de una sentencia etiquetada es la siguiente:

```
etiqueta: sentencia;
```

continue

```
continue [etiqueta];
```

La sentencia continue no se puede utilizar en una sentencia switch, sino solamente en bucles. Cuando se encuentra esta sentencia en el discurrir normal de un programa Java, la iteración en que se encuentre el bucle finaliza y se inicia la siguiente.

Java permite el uso de etiquetas en la sentencia continue, de forma que el funcionamiento normal se ve alterado y el salto en la ejecución del flujo del programa se realizará a la sentencia en la que se encuentra colocada la etiqueta.

Por ejemplo, al encontrarse con bucles anidados, se pueden utilizar etiquetas para poder salir de ellos:

```
uno: for( ) {
         dos: for( ) {
              continue;       // seguiría en el bucle interno
              continue uno;   // seguiría en el bucle principal
              break uno;      // se saldría del bucle principal
              }
```

```
}
```

Hay autores que sugieren que el uso de sentencias break y continue etiquetadas proporciona una alternativa al infame goto (que C++ todavía soporta, pero Java no, aunque reserva la palabra). Quizá sea así, pero el entorno de uso de las sentencias etiquetadas break y continue es mucho más restrictivo que un goto. Concretamente, parece que un break o continue con etiqueta, compila con éxito solamente si la sentencia en que se encuentra colocada la etiqueta es una sentencia a la que se pueda llegar con un break o continue normal.

return

```
return expresión;
```

La sentencia return se utiliza para terminar un método o función y opcionalmente devolver un valor al método de llamada.

En el código de una función siempre hay que ser consecuentes con la declaración que se haya hecho de ella. Por ejemplo, si se declara una función para que devuelva un entero, es imprescindible colocar un return final para salir de esa función, independientemente de que haya otros en medio del código que también provoquen la salida de la función. En caso de no hacerlo se generará un Warning, y el código Java no se puede compilar con Warnings.

```
int func() {
    if( a == 0 )
        return 1;
    return 0; // es imprescindible porque se retorna un entero
    }
```

Si el valor a retornar es void, se puede omitir ese valor de retorno, con lo que la sintaxis se queda en un sencillo:

```
return;
```

y se usaría simplemente para finalizar el método o función en que se encuentra, y devolver el control al método o función de llamada.

Almacenamiento de Datos

En esta sección se va a tratar las Colecciones, o Estructuras de Datos, que Java aporta para la manipulación y Almacenamiento de Datos e información. Se presentan en primer lugar las Colecciones que están disponibles en la versión 1.1 del JDK, para luego tratar las nuevas Colecciones que se incorporan al lenguaje con el JDK 1.2 y que representan un tremendo avance en la potencia y facilidad de uso de las estruturas de datos orientadas al almacenamiento de información.

Arrays

Mucho de lo que se podría decir de los arrays ya se cuenta en otra sección, aquí sólo interesa el array como almacén de objetos. Hay dos características que diferencian a los arrays de cualquier otro tipo de colección: eficiencia y tipo. El array es la forma más eficiente que Java proporciona para almacenar y acceder a una secuencia de objetos. El array es una simple secuencia lineal, que hace que el acceso a los elementos sea muy rápido, pero el precio que hay que pagar por esta velocidad es que cuando se crea un array

su tamaño es fijado y no se puede cambiar a lo largo de la vida del objeto. Se puede sugerir la creación de un array de tamaño determinado y luego, ya en tiempo de ejecución, crear otro más grande, mover todos los objetos al nuevo y borrar el antiguo. Esto es lo que hace la clase Vector, que se verá posteriormente, pero debido a la carga que supone esta flexibilidad, un Vector es menos eficiente que un array, en cuestiones de velocidad.

La clase vector en C++ no sabe el tipo de los objetos que contiene, pero tiene un inconveniente cuando se la compara con los arrays en Java: el operador [] de la clase vector de C++ no realiza un chequeo de límites, así que uno se puede pasar del final (aunque es posible saber el tamaño del vector invocando al método at() para realizar la comprobación de límites, si se desea). En Java, independientemente de que se esté utilizando un array o una colección, siempre hay comprobación de límites, y el sistema lanzará una excepción en caso de que se intente el acceso a un elemento que se encuentra fuera de los límites. Así pues, la razón de que C++ no compruebe límites es la velocidad, y con esta sobrecarga hay que vivir siempre en Java, porque todas las veces realizará la comprobación de que no se acceda a un lugar fuera del array o la colección.

Los otros tipos de colecciones disponibles en Java: Vector, Stack y Hashtable; pueden contener cualquier tipo de objeto, sin necesidad de que sea de un tipo definido. Esto es así porque tratan a sus elementos como si fuesen Object, la clase raíz de todas las clases Java. Esto es perfecto desde el punto de vista de que se construye solamente una colección, y cualquier tipo de objeto puede ser almacenado en ella. Pero aquí es donde los arrays vuelven a ser más eficientes que las colecciones genéricas, porque cuando se crea un array hay que indicar el tipo de objetos que va a contener. Esto significa que ya en tiempo de compilación se realizan comprobaciones para que no se almacene en el array ningún objeto de tipo diferente al que está destinado a contener, ni que se intente extraer un objeto diferente. Desde luego, Java controlará de que no se envíe un mensaje inadecuado a un objeto, ya sea en tiempo de compilación como en tiempo de ejecución.

Así que, tanto por eficiencia como por comprobación de tipos, es mejor utilizar un array siempre que se pueda. Sin embargo, cuando se trata de resolver un problema más general, los arrays pueden ser muy restrictivos y es entonces cuando hay que recurrir a otro tipo de almacenamiento de datos.

Las colecciones de clases manejan solamente los identificadores, handles, de los objetos. Un array, sin embargo, puede crearse para contener tipos básicos directamente, o también identificadores de objetos. Es posible utilizar las clases correspondientes a los tipos básicos, como son Integer, Double, etc., para colocar tipos básicos dentro de una colección. El colocar una cosa u otra es cuestión de eficiencia, porque es mucho más rápida la creación y acceso en un array de tipos básicos que en uno de objetos del tipo básico.

Desde luego, si se está utilizando un tipo básico y se necesita la flexibilidad que ofrece una colección de expandirse cuando sea preciso, el array no sirve y habrá que recurrir a la colección de objetos del tipo básico. Quizás se podría pensar que un Vector especializado en cada uno de los tipos básicos podría ser casi igual de eficiente que un array, pero por desgracia, Java no proporciona mas que un tipo de genérico de Vector, en el que se puede meter de todo. Este es otro de las cuestiones que Java tiene pendientes.

Colecciones

Cuando se necesitan características más sofisticadas para almacenar objetos, que las que proporciona un simple array, Java pone a disposición del programador las clases colección: Vector, BitSet, Stack y Hashtable.

Entre otras características, las clases colección se redimensionan automáticamente, por lo que se puede colocar en ellas cualquier número de objetos, sin necesidad de tener que ir controlando continuamente en el programa la longitud de la colección.

La gran desventaja del uso de las colecciones en Java es que se pierde la información de tipo cuando se coloca un objeto en una colección. Esto ocurre porque cuando se escribió la colección, el programador de esa colección no tenía ni idea del tipo de datos específicos que se iban a colocar en ella, y teniendo en mente el hacer una herramienta lo más general posible, se hizo que manejase directamente objetos de tipo Object, que es el objeto raíz de todas las clases en Java. La solución es perfecta, excepto por dos razones:

1. Como la información de tipo se pierde al colocar un objeto en la colección, cualquier tipo de objeto se va a poder colar en ella, es decir, si la colección está destinada a contener animales mamíferos, nada impide que se pueda colar un coche en ella.

2. Por la misma razón de la pérdida de tipo, la única cosa que sabe la colección es que maneja un Object. Por ello, hay que colocar siempre un moldeo al tipo adecuado antes de utilizar cualquier objeto contenido en una colección.

La verdad es que no todo es tan negro, Java no permite que se haga uso inadecuado de los objetos que se colocan en una colección. Si se introduce un coche en una colección de animales mamíferos, al intentar extraer el coche se obtendrá una excepción. Y del mismo modo, si se intenta colocar un moldeo al coche que se está sacando de la colección para convertirlo en animal mamífero, también se obtendrá una excepción en tiempo de ejecución.

El ejemplo ilustra estas circunstancias.

```
import java.util.*;

class Coche {
    private int numCoche;
    Coche( int i ) {
        numCoche = i;
        }
    void print() {
        System.out.println( "Coche #"+numCoche );
        }
    }

class Barco {
    private int numBarco;
    Barco( int i ) {
        numBarco = i;
        }
    void print() {
        System.out.println( "Barco #"+numBarco );
        }
    }

public class java411 {
```

```
    public static void main( String args[] ) {
        Vector coches = new Vector();
        for( int i=0; i < 7; i++ )
            coches.addElement( new Coche( i ) );
          // No hay ningun problema en añadir un barco a los coches
        coches.addElement( new Barco( 7 ) );
        for( int i=0; i < coches.size(); i++ )
            (( Coche )coches.elementAt( i ) ).print();
        // El barco solamente es detectado en tiempo de ejecucion
        }
    }
```

Como se puede observar, el uso de un Vector es muy sencillo: se crea uno, se colocan elementos en él con el método addElement() y se recuperan con el método elementAt(). Vector tiene el método size() que permite conocer cuántos elementos contiene, para evitar el acceso a elementos fuera de los límites del Vector y obtener una excepción.

Las clases Coche y Barco son distintas, no tienen nada en común excepto que ambas son Object. Si no se indica explícitamente de la clase que se está heredando, automáticamente se hereda de Object. La clase Vector maneja elementos de tipo Object, así que no solamente es posible colocar en ella objetos Coche utilizando el método addElement(), sino que también se pueden colocar elementos de tipo Barco sin que haya ningún problema ni en tiempo de compilación ni a la hora de ejecutar el programa. Cuando se recupere un objeto que se supone es un Coche utilizando el método elementAt() de la clase Vector, hay que colocar un moldeo para convertir el objeto Object en el Coche que se espera, luego hay que colocar toda la expresión entre paréntesis para forzar la evaluación del moldeo antes de llamar al método print() de la clase Coche, sino habrá un error de sintaxis. Posteriormente, ya en tiempo de ejecución, cuando se intente moldear un objeto Barco a un Coche, se generará una excepción, tal como se puede comprobar en las siguientes líneas, que reproducen la salida de la ejecución del ejemplo:

```
%java java411
Coche #0
Coche #1
Coche #2
Coche #3
Coche #4
Coche #5
Coche #6
java.lang.ClassCastException: Barco
    at java411.main(java411.java:54)
```

Lo cierto es que esto es un fastidio, porque puede ser la fuente de errores que son muy difíciles de encontrar. Si en una parte, o en varias partes, del programa se insertan elementos en la colección, y se descubre en otra parte diferente del programa que se genera una excepción es porque hay algún elemento erróneo en la colección, así que hay que buscar el sitio donde se ha insertado el elemento de la discordia, lo cual puede llevar a intensas sesiones de depuración. Así que, para enredar al principio, es mejor empezar con clases estandarizadas en vez de aventurarse en otras más complicadas, a pesar de que estén menos optimizadas.

Enumeraciones

En cualquier clase de colección, debe haber una forma de meter cosas y otra de sacarlas; después de todo, la principal finalidad de una colección es almacenar cosas. En un Vector, el método addElement() es la manera en que se colocan objetos dentro de la

colección y llamando al método elementAt() es cómo se sacan. Vector es muy flexible, se puede seleccionar cualquier cosa en cualquier momento y seleccionar múltiples elementos utilizando diferentes índices.

Si se quiere empezar a pensar desde un nivel más alto, se presenta un inconveniente: la necesidad de saber el tipo exacto de la colección para utilizarla. Esto no parece que sea malo en principio, pero si se empieza implementando un Vector a la hora de desarrollar el programa, y posteriormente se decide cambiarlo a List, por eficiencia, entonces sí es problemático.

El concepto de enumerador, o iterador, que es su nombre más común en C++ y OOP, puede utilizarse para alcanzar el nivel de abstracción que se necesita en este caso. Es un objeto cuya misión consiste en moverse a través de una secuencia de objetos y seleccionar aquellos objetos adecuados sin que el programador cliente tenga que conocer la estructura de la secuencia . Además, un iterador es normalmente un objeto ligero, lightweight, es decir, que consumen muy pocos recursos, por lo que hay ocasiones en que presentan ciertas restricciones; por ejemplo, algunos iteradores solamente se puede mover en una dirección.

La Enumeration en Java es un ejemplo de un iterador con esas características, y las cosas que se pueden hacer son:

- Crear una colección para manejar una Enumeration utilizando el método elements(). Esta Enumeration estará lista para devolver el primer elemento en la secuencia cuando se llame por primera vez al método nextElement().
- Obtener el siguiente elemento en la secuencia a través del método nextElement().
- Ver si hay más elementos en la secuencia con el método hasMoreElements().

Y esto es todo. No obstante, a pesar de su simplicidad, alberga bastante poder. Para ver cómo funciona, el ejemplo , es la modificación de anterior, en que se utilizaba el método elementAt() para seleccionar cada uno de los elementos. Ahora se utiliza una enumeración para el mismo propósito, y el único código interesante de este nuevo ejemplo es el cambio de las líneas del ejemplo original

```
for( int i=0; i < coches.size(); i++ )
    (( Coche )coches.elementAt( i ) ).print();
```

por estas otras en que se utiliza la enumeración para recorrer la secuencia de objetos

```
while( e.hasMoreElements() )
    (( Coche )e.nextElement()).print();
```

Con la Enumeration no hay que preocuparse del número de elementos que contenga la colección, ya que del control sobre ellos se encargan los métodos hasMoreElements() y nextElement().

Tipos de Colecciones

Con el JDK 1.0 y 1.1 se proporcionaban librerías de colecciones muy básicas, aunque suficientes para la mayoría de los proyectos. En el JDK 1.2 ya se amplía esto y, además, las anteriores colecciones han sufrido un profundo rediseño. A continuación se

verán cada una de ellas por separado para dar una idea del potencial que se ha incorporado a Java.

Vector

El Vector es muy simple y fácil de utilizar. Aunque los métodos más habituales en su manipulación son **addElement**() para insertar elementos en el Vector, **elementAt**() para recuperarlos y **elements**() para obtener una Enumeration con el número de elementos del Vector, lo cierto es que hay más métodos, pero no es el momento de relacionarlos todos, así que, al igual que sucede con todas las librerías de Java, se remite al lector a que consulte la documentación electrónica que proporciona Javasoft, para conocer los demás métodos que componen esta clase.

Las colecciones estándar de Java contienen el método toString(), que permite obtener una representación en forma de String de sí mismas, incluyendo los objetos que contienen. Dentro de Vector, por ejemplo, toString() va saltando a través de los elementos del Vector y llama al método toString() para cada uno de esos elementos. En caso, por poner un ejemplo, de querer imprimir la dirección de la clase, parecería lógico referirse a ella simplemente como this (los programadores C++ estarán muy inclinados a esta posibilidad), así que tendríamos el código que muestra el ejemplo y que se reproduce en las siguientes líneas.

```
import java.util.*;

public class java413 {
    public String toString() {
        return( "Direccion del objeto: "+this+"\n" );
        }

    public static void main( String args[] ) {
        Vector v = new Vector();

        for( int i=0; i < 10; i++ )
            v.addElement( new java413() );
        System.out.println( v );
        }
    }
```

El ejemplo no puede ser más sencillo, simplemente crea un objeto de tipo java413 y lo imprime; sin embargo, a la hora de ejecutar el programa lo que se obtiene es una secuencia infinita de excepciones. Lo que está pasando es que cuando se le indica al compilador:

```
"Direccion del objeto: "+this
```

el compilador ve un String seguido del operador + y otra cosa que no es un String, así que intenta convertir this en un String. La conversión la realiza llamando al método toString() que genera una llamada recursiva, llegando a llenarse la pila.

Si realmente se quiere imprimir la dirección del objeto en este caso, la solución pasa por llamar al método toString() de la clase Object. Así, si en vez de this se coloca super.toString(), el ejemplo funcionará. En otros casos, este método también funcionará siempre que se esté heredando directamente de Object o, aunque no sea así, siempre que ninguna clase padre haya sobreescrito el método toString().

BitSet

Se llama así lo que en realidad es un Vector de bits. Lo que ocurre es que está optimizado para uso de bits. Bueno, optimizado en cuanto a tamaño, porque en lo que respecta al tiempo de acceso a los elementos, es bastante más lento que el acceso a un array de elementos del mismo tipo básico.

Además, el tamaño mínimo de un BitSet es de 64 bits. Es decir, que si se está almacenando cualquier otra cosa menor, por ejemplo de 8 bits, se estará desperdiciando espacio.

En un Vector normal, la colección se expande cuando se añaden más elementos. En el BitSet ocurre los mismo pero ordenadamente. El ejemplo , muestra el uso de esta colección.

Se utiliza el generador de números aleatorios para obtener un byte, un short y un int, que son convertidos a su patrón de bits e incorporados al BitSet.

Stack

Un Stack es una Pila, o una colección de tipo LIFO (last-in, first-out). Es decir, lo último que se coloque en la pila será lo primero que se saque. Como en todas las colecciones de Java, los elementos que se introducen y sacan de la pila son Object, así que hay que tener cuidado con el moldeo a la hora de sacar alguno de ellos.

Los diseñadores de Java, en vez de utilizar un Vector como bloque para crear un Stack, han hecho que Stack derive directamente de Vector, así que tiene todas las características de un Vector más alguna otra propia ya del Stack. El ejemplo siguiente, , es una demostración muy simple del uso de una Pila que consisten en leer cada una de las líneas de un array y colocarlas en un String.

Cada línea en el array diasSemana se inserta en el Stack con push() y posteriormente se retira con pop(). Para ilustrar una afirmación anterior, también se utilizan métodos propios de Vector sobre el Stack. Esto es posible ya que en virtud de la herencia un Stack es un Vector, así que todas las operaciones que se realicen sobre un Vector también se podrán realizar sobre un Stack, como por ejemplo, elementAt().

Hashtable

Un Vector permite selecciones desde una colección de objetos utilizando un número, luego parece lógico pensar que hay números asociados a los objetos. Bien, entonces ¿qué es lo que sucede cuando se realizan selecciones utilizando otros criterios? Un Stack podría servir de ejemplo: su criterio de selección es "lo último que se haya colocado en el Stack". Si rizamos la idea de "selección desde una secuencia", nos encontramos con un mapa, un diccionario o un array asociativo. Conceptualmente, todo parece ser un vector, pero en lugar de acceder a los objetos a través de un número, en realidad se utiliza otro objeto. Esto nos lleva a utilizar claves y al procesado de claves en el programa. Este concepto se expresa en Java a través de la clase abstracta Dictionary. El interfaz para esta clase es muy simple:

- **size**(), indica cuántos elementos contiene,

- **isEmpty**(), es true si no hay ningún elemento,
- **put**(Object clave,Object valor), añade un valor y lo asocia con una clave
- **get**(Object clave), obtiene el valor que corresponde a la clave que se indica
- **remove**(Object clave), elimina el par clave-valor de la lista
- **keys**(), genera una Enumeration de todas las claves de la lista
- **elements**(), genera una Enumeration de todos los valores de la lista

Todo es lo que corresponde a un Diccionario (Dictionary), que no es excesivamente difícil de implementar. El ejemplo es una aproximación muy simple que utiliza dos Vectores, uno para las claves y otro para los valores que corresponden a esas claves.

```
import java.util.*;

public class java416 extends Dictionary {
    private Vector claves = new Vector();
    private Vector valores = new Vector();
    public int size() {
        return( claves.size() );
        }
    public boolean isEmpty() {
        return( claves.isEmpty() );
        }

    public Object put( Object clave,Object valor ) {
        claves.addElement( clave );
        valores.addElement( valor );
        return( clave );
        }

    public Object get( Object clave ) {
        int indice = claves.indexOf( clave );
        // El metodo indexOf() devuelve -1 si no encuentra la clave que
se
        // esta buscando
        if( indice == -1 )
            return( null );
        return( valores.elementAt( indice ) );
        }

    public Object remove(Object clave) {
        int indice = claves.indexOf( clave );

        if( indice == -1 )
            return( null );
        claves.removeElementAt( indice );
        Object valorRetorno = valores.elementAt( indice );
        valores.removeElementAt( indice );
        return( valorRetorno );
        }

    public Enumeration keys() {
        return( claves.elements() );
        }

    public Enumeration elements() {
        return( valores.elements() );
        }
```

```
    // Ahora es cuando se prueba el ejemplo
    public static void main( String args[] ) {
        java416 ej = new java416();
        for( char c='a'; c <= 'z'; c++ )
            ej.put( String.valueOf( c ),String.valueOf( c ).toUpperCase()
);

        char[] vocales = { 'a','e','i','o','u' };
        for( int i=0; i < vocales.length; i++ )
            System.out.println( "Mayusculas: " +
            ej.get( String.valueOf( vocales[i] ) ) );
        }
    }
```

La primera cosa interesante que se puede observar en la definición de java416 es que extiende a Dictionary. Esto significa que java416 es un tipo de Diccionario, con lo cual se pueden realizar las mismas peticiones y llamar a los mismos métodos que a un Diccionario. A la hora de construirse un Diccionario propio todo lo que se necesita es rellenar todos los métodos que hay en Dictionary. Se deben sobreescribir todos ellos, excepto el constructor, porque todos son abstractos.

Los Vectores claves y valores están relacionados a través de un número índice común. Es decir, si se llama al método put() con la clave "león" y el valor "rugido" en la asociación de animales con el sonido que producen, y ya hay 100 elementos en la clase java416, entonces "león" será el elemento 101 de claves y "rugido" será el elemento 101 de valores. Y cuando se pasa al método get() como parámetro "león", genera el número índice con claves.indexOf(), y luego utiliza este índice para obtener el valor asociado en el vector valores.

Para mostrar el funcionamiento, en main() se utiliza algo tan simple como mapear las letras minúsculas y mayúsculas, que aunque se pueda hacer de otras formas más eficientes, sí sirve para mostrar el funcionamiento de la clase, que es lo que se pretende por ahora.

La librería estándar de Java solamente incorpora una implementación de un Dictionary, la Hashtable. Esta Hashtable tiene el mismo interfaz básico que la clase del ejemplo anterior java416, ya que ambas heredan de Dictionary, pero difiere en algo muy importante: la eficiencia. Si en un Diccionario se realiza un get() para obtener un valor, se puede observar que la búsqueda es bastante lenta a través del vector de claves. Aquí es donde la Hashtable acelera el proceso, ya que en vez de realizar la tediosa búsqueda línea a línea a través del vector de claves, utiliza un valor especial llamado código hash. El código hash es una forma de conseguir información sobre el objeto en cuestión y convertirlo en un int relativamente único para ese objeto. Todos los objetos tienen un código hash y hashCode() es un método de la clase Object. Una Hashtable coge el hashCode() del objeto y lo utiliza para cazar rápidamente la clave. El resultado es una impresionante reducción del tiempo de búsqueda. La forma en que funciona una tabla Hash se escapa del Tutorial, hay muchos libros que lo explican en detalle, por ahora es suficiente con saber que la tabla Hash es un Diccionario muy rápido y que un Diccionario es una herramienta muy útil.

Para ver el funcionamiento de la tabla Hash está el ejemplo , que intenta comprobar la aleatoriedad del método Math.random(). Idealmente, debería producir una distribución perfecta de números aleatorios, pero para poder comprobarlo sería necesario generar una buena cantidad de números aleatorios y comprobar los rangos en que caen. Una Hashtable

es perfecta para este propósito al asociar objetos con objetos, en este caso, los valores producidos por el método Math.random() con el número de veces en que aparecen esos valores.

En el método main() del ejemplo, cada vez que se genera un número aleatorio, se convierte en objeto Integer para que pueda ser manejado por la tabla Hash, ya que no se pueden utilizar tipos básicos con una colección, porque solamente manejan objetos. El método containsKey() comprueba si la clave se encuentra ya en la colección. En caso afirmativo, el método get() obtiene el valor asociado a la clave, que es un objeto de tipo Contador. El valor i dentro del contador se incrementa para indicar que el número aleatorio ha aparecido una vez más.

Si la clave no se encuentra en la colección, el método put() colocará el nuevo par clave-valor en la tabla Hash. Como Contador inicializa automáticamente su variable i a 1 en el momento de crearla, ya se indica que es la primera vez que aparece ese número aleatorio concreto.

Para presentar los valores de la tabla Hash, simplemente se imprimen. El método toString() de Hashtable navega a través de los pares clave-valor y llama a método toString() de cada uno de ellos. El método toString() de Integer está predefinido, por lo que no hay ningún problema en llamar a toString() para Contador. Un ejemplo de ejecución del programa sería la salida que se muestra a continuación:

```
%java java417
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495, 13=512, 12=483,
11=488, 10=487, 9=514, 8=523, 7=497, 6=487, 5=489, 3=509, 2=503,
1=475, 0=505}
```

Al lector le puede parecer superfluo el uso de la clase Contador, que parece que no hace nada que no haga ya la clase Integer. ¿Por qué no utilizar int o Integer? Pues bien, int no puede utilizarse porque como ya se ha indicado antes, las colecciones solamente manejan objetos, por ello están las clases que envuelven a esos tipos básicos y los convierten en objetos. Sin embargo, la única cosa que pueden hacer estas clases es inicializar los objetos a un valor determinado y leer ese valor. Es decir, no hay modo alguno de cambiar el valor de un objeto correspondiente a un tipo básico, una vez que se ha creado. Esto hace que la clase Integer sea inútil para resolver el problema que plantea el ejemplo, así que la creación de la clase Contador es imprescindible. Quizás ahora que el lector sabe que no puede colocar objetos creados a partir de las clases correspondientes a tipos básicos en colecciones, estas clases tengan un poco menos de valor, pero... la vida es así, por un lado da y por otro quita... y Java no va a ser algo diferente.

En el ejemplo se utiliza la clase Integer, que forma parte de la librería estándar de Java como clave para la tabla Hash, y funciona perfectamente porque tiene todo lo necesario para funcionar como clave. Pero un error muy común se presenta a la hora de crear clases propias para que funcionen como claves. Por ejemplo, supóngase que se quiere implementar un sistema de predicción del tiempo en base a objetos de tipo Oso y tipo Prediccion, para detectar cuando entra la primavera. Tal como se muestra en el ejemplo , la cosa parece muy sencilla, se crean las dos clases y se utiliza Oso como clave y Prediccion como valor.

Cada Oso tendrá un número de identificación, por lo que sería factible buscar una Prediccion en la tabla Hash de la forma: "Dime la Prediccion asociada con el Oso número

3". La clase Prediccion contiene un booleano que es inicializado utilizando Math.random(), y una llamada al método toString() convierte el resultado en algo legible. En el método main(), se rellena una Hashtable con los Osos y sus Predicciones asociadas. Cuando la tabla Hash está completa, se imprime. Y ya se hace la consulta anterior sobre la tabla para buscar la Prediccion que corresponde al Oso número 3.

Esto parece simple y suficiente, pero no funciona. El problema es que Oso deriva directamente de la clase raíz Object, que es lo que ocurre cuando no se especifica una clase base, que en última instancia se hereda de Object. Luego es el método hashCode() de Object el que se utiliza para generar el código hash para cada objeto que, por defecto, utiliza la dirección de ese objeto. Así, la primera instancia de Oso(3) no va a producir un código hash igual que producirá una segunda instancia de Oso(3), con lo cual no se puede utilizar para obtener buenos resultados de la tabla.

Se puede seguir pensando con filosofía ahorrativa y decir que todo lo que se necesita es sobreescribir el método hashCode() de la forma adecuada y ya está. Pero, esto tampoco va a funcionar hasta que se haga una cosa más: sobreescribir el método equals(), que también es parte de Object. Este es el método que utiliza la tabla Hash para determinar si la clave que se busca es igual a alguna de las claves que hay en la tabla. De nuevo, el método Object.equals() solamente compara direcciones de objetos, por lo que un Oso(3) probablemente no sea igual a otro Oso(3).

Por lo tanto, a la hora de escribir clases propias que vayan a funcionar como clave en una Hastable, hay que sobreescribir los métodos hashCode() y equals(). El ejemplo ya se incorporan estas circunstancias.

```
import java.util.*;

// Si se crea una clase que utilice una clave en una Tabla Hash, es
// imprescindible sobreescribir los metodos hashCode() y equals()
// Utilizamos un oso para saber si está hibernando en su temporada de
// invierno o si ya tine que despertarse porque le llega la primavera
class Oso2 {
    int numero;
    Oso2( int n ) {
        numero = n;
        }

    public int hashCode() {
        return( numero );
        }

    public boolean equals( Object obj ) {
        if( (obj != null) && (obj instanceof Oso2) )
            return( numero == ((Oso2)obj).numero );
        else
            return( false );
        }
    }

// En función de la oscuridad, o claridad del día, pues intenta saber si
// ya ha la primavera ha asomado a nuestras puertas
class Prediccion {
    boolean oscuridad = Math.random() > 0.5;

    public String toString() {
        if( oscuridad )
            return( "Seis semanas mas de Invierno!" );
```

```
        else
            return( "Entrando en la Primavera!" );
        }
    }

public class java419 {
    public static void main(String args[]) {
        Hashtable ht = new Hashtable();

        for( int i=0; i < 10; i++ )
            ht.put( new Oso2( i ),new Prediccion() );
        System.out.println( "ht = "+ht+"\n" );

        System.out.println( "Comprobando la prediccion para el oso #3:");
        Oso2 oso = new Oso2( 3 );
        if( ht.containsKey( oso ) )
            System.out.println( (Prediccion)ht.get( oso ) );
        }
    }
```

El método hashCode() devuelve el número que corresponde a un Oso como un identificador, siendo el programador el responsable de que no haya dos números iguales. El método hashCode() no es necesario que devuelva un identificador, sino que eso es necesario porque equals() debe ser capaz de determinar estrictamente cuando dos objetos son equivalentes.

El método equals() realiza dos comprobaciones adicionales, una para comprobar si el objeto es null, y, en caso de que no lo sea, comprobar que sea una instancia de Oso, para poder realizar las comparaciones, que se basan en los números asignados a cada objeto Oso. Cuando se ejecuta este nuevo programa, sí se produce la salida correcta. Hay muchas clases de la librería de Java que sobreescriben los métodos hashCode() y equals() basándose en el tipo de objetos que son capaces de crear.

Nuevas colecciones

Para el autor, las colecciones son una de las herramientas más poderosas que se pueden poner en manos de un programador. Por ello, las colecciones que incorporaba Java, adolecían de precariedad y de demasiada rapidez en su desarrollo. Por todo ello, para quien escribe esto ha sido una tremenda satisfacción comprobar las nuevas colecciones que incorpora el JDK 1.2, y ver que incluso las antiguas han sido rediseñadas. Probablemente, las colecciones, junto con la librería Swing, son las dos cosas más importantes que aporta la versión 1.2 del JDK, y ayudarán enormemente a llevar a Java a la primera línea de los lenguajes de programación.

Hay cambios de diseño que hacen su uso más simple. Por ejemplo, muchos nombres son más cortos, más claros y más fáciles de entender; e incluso algunos de ellos han sido cambiados totalmente para adaptarse a la terminología habitual.

El rediseño también incluye la funcionalidad, pudiendo encontrar ahora listas enlazadas y colas. El diseño de una librería de colecciones es complicado y difícil. En C++, la Standard Template Library (STL) cubría la base con muchas clases diferentes. Desde luego, esto es mejor que cuando no hay nada, pero es difícil de trasladar a Java porque el resultado llevaría a tal cantidad de clases que podría ser muy confuso. En el otro extremo, hay librerías de colecciones que constan de una sola clase, Collection, que actúa como un Vector y una Hashtable al mismo tiempo. Los diseñadores de las nuevas colecciones han

intentado mantener un difícil equilibrio: por un lado disponer de toda la funcionalidad que el programador espera de una buena librería de colecciones, y, por otro, que sea tan fácil de aprender y utilizar como la STL y otras librerías similares. El resultado puede parecer un poco extraño en ocasiones, pero, al contrario que en las librerías anteriores al JDK 1.2, no son decisiones accidentales, sino que están tomadas a conciencia en función de la complejidad. Es posible que se tarde un poco en sentirse cómodo con algunos de los aspectos de la librería, pero de seguro que el programador intentará adoptar rápidamente estos nuevos métodos. Hay que reconocer que Joshua Bloch de Sun, ha hecho un magnífico trabajo en el rediseño de esta librería.

La nueva librería de colecciones parte de la premisa de almacenar objetos, y diferencia dos conceptos en base a ello:

- Colección (Collection): un grupo de elementos individuales, siempre con alguna regla que se les puede aplicar. Una List almacenará objetos en una secuencia determinada y, un Set no permitirá elementos duplicados.

- Mapa (Map): un grupo de parejas de objetos clave-valor, como la Hastable ya vista. En principio podría parecer que esto es una Collection de parejas, pero cuando se intenta implementar, este diseño se vuelve confuso, por lo que resulta mucho más claro tomarlo como un concepto separado. Además, es conveniente consultar porciones de un Map creando una Collection que represente a esa porción; de este modo, un Map puede devolver un Set de sus claves, una List de sus valores, o una List de sus parejas clave-valor. Los Mapas, al igual que los arrays, se pueden expandir fácilmente en múltiples dimensiones sin la incorporación de nuevos conceptos: simplemente se monta un Map cuyos valores son Mapas, que a su vez pueden estar constituidos por Mapas,

Las Colecciones y los Mapas pueden ser implementados de muy diversas formas, en función de las necesidades concretas de programación, por lo que puede resultar útil el siguiente diagrama de herencia de las nuevas colecciones que utiliza la notación gráfica propugnada por la metodología OMT (Object Modeling Technique).

El diagrama está hecho a partir de la versión beta del JDK 1.2, así que puede haber cosas cambiadas con respecto a la versión final. Quizás también, un primer vistazo puede abrumar al lector, pero a lo largo de la sección se comprobará que es bastante simple, porque solamente hay tres colecciones: Map, List y Set; y solamente dos o tres implementaciones de cada una de ellas. Las cajas punteadas representan interfaces y las sólidas representan clases normales, excepto aquellas en que el texto interior comienza por Abstract, que representan clases abstractas. Las flechas indican que una clase puede generar objetos de la clase a la que apunta; por ejemplo, cualquier Collection puede producir un Iterator, mientras que una List puede producir un ListIterator (al igual que un Iterator normal, ya que List hereda de Collection).

Los interfaces que tienen que ver con el almacenamiento de datos son: Collection, Set, List y Map. Normalmente, un programador creará casi todo su código para entenderse con estos interfaces y solamente necesitará indicar específicamente el tipo de datos que se están usando en el momento de la creación. Por ejemplo, una Lista se puede crear de la siguiente forma:

```
List lista = new LinkedList();
```

Desde luego, también se puede decidir que lista sea una lista enlazada, en vez de una lista genérica, y precisar más el tipo de información de la lista. Lo bueno, y la intención, del uso de interfaces es que si ahora se decide cambiar la implementación de la lista, solamente es necesario cambiar el punto de creación, por ejemplo:

```
List lista = new ArrayList();
```

el resto del código permanece invariable.

En la jerarquía de clases, se pueden ver algunas clases abstractas que pueden confundir en un principio. Son simplemente herramientas que implementan parcialmente un interfaz. Si el programador quiere hacer su propio Set, por ejemplo, no tendría que

empezar con el interfaz Set e implementar todos los métodos, sino que podría derivar directamente de AbstractSet y ya el trabajo para crear la nueva clase es mínimo. Sin embargo, la nueva librería de colecciones contiene suficiente funcionalidad para satisfacer casi cualquier necesidad, así que en este Tutorial se ignorarán las clases abstractas.

Por lo tanto, a la hora de sacar provecho del diagrama es suficiente con lo que respecta a los interfaces y a las clases concretas. Lo normal será construir un objeto correspondiente a una clase concreta, moldearlo al correspondiente interfaz y ya usas ese interfaz en el resto del código. El ejemplo es muy simple y consiste en una colección de objetos String que se imprimen.

```
import java.util.*;

public class java420 {
    public static void main( String args[] ) {
        Collection c = new ArrayList();
        for( int i=0; i < 10; i++ )
            c.add( Integer.toString( i ) );

        Iterator it = c.iterator();
        while( it.hasNext() )
            System.out.println( it.next() );
        }
    }
```

Como las nuevas colecciones forman parte del paquete java.util, no es necesario importar ningún paquete adicional para utilizarlas.

A continuación se comentan los trozos interesantes del código del ejemplo. La primera línea del método main() crea un objeto ArrayList y lo moldea a una Collection. Como este ejemplo solamente utiliza métodos de Collection, cualquier objeto de una clase derivada de Collection debería funcionar, pero se ha cogido un ArrayList porque es el caballo de batalla de las colecciones y viene a tomar el relevo al Vector.

El método add(), como su nombre sugiere, coloca un nuevo elemento en la colección. Sin embargo, la documentación indica claramente que add() "asegura que la colección contiene el elemento indicado". Esto es para que un Set tenga significado, ya que solamente añadirá el elemento si no se encuentra en la colección. Con un ArrayList, o cualquier otra lista ordenada, add() significa siempre "colocarlo dentro".

Todas las colecciones pueden producir un Iterator invocando al método iterator(). Un Iterator viene a ser equivalente a una Enumeration, a la cual reemplaza, excepto en los siguientes puntos:

- Utiliza un nombre que está históricamente aceptado y es conocido en toda la literatura de programación orientada a objetos
- Utiliza nombres de métodos más cortos que la Enumeration: hasNext() en vez de hasMoreElements(), o next() en lugar de nextElement()
- Añade un nuevo método, remove(), que permite eliminar el último elemento producido por el Iterator. Solamente se puede llamar a remove() una vez por cada llamada a next()

En el ejemplo se utiliza un Iterator para desplazarse por la colección e ir imprimiendo cada uno de sus elementos.

Colecciones

A continuación se indican los métodos que están disponibles para las colecciones, es decir, lo que se puede hacer con un Set o una List, aunque las listas tengan funcionalidad añadida que ya se verá, y Map no hereda de Collection, así que se tratará aparte.

boolean add (Object)

Asegura que la colección contiene el argumento. Devuelve false si no se puede añadir el argumento a la colección

boolean addAll (Collection)

Añade todos los elementos que se pasan en el argumento. Devuelve true si es capaz de incorporar a la colección cualquiera de los elementos del argumento

void clear()

Elimina todos los elementos que componen la colección

boolean contains (Object)

Verdadero si la colección contiene el argumento que se pasa como parámetro

boolean isEmpty()

Verdadero si la colección está vacía, no contiene elemento alguno

Iterator iterator()

Devuelve un Iterator que se puede utilizar para desplazamientos a través de los elementos que componen la colección

boolean remove(Object)

Si el argumento está en la colección, se elimina una instancia de ese elemento y se devuelve true si se ha conseguido

boolean removeAll(Collection)

Elimina todos los elementos que están contenidos en el argumento. Devuelve true si consigue eliminar cualquiera de ellos

boolean retainAll(Collection)

Mantiene solamente los elementos que están contenidos en el argumento, es lo que sería una intersección en la teoría de conjuntos. Devuelve verdadero en caso de que se produzca algún cambio

int size()

Devuelve el número de elementos que componen la colección

Object[] toArray()

Devuelve un array conteniendo todos los elementos que forman parte de la colección. Este es un método opcional, lo cual significa que no está implementado para una Collection determinada. Si no puede devolver el array, lanzará una excepción de tipo UnsupportedOperationException

El siguiente ejemplo muestra un ejemplo de todos estos métodos. De nuevo, recordar que esto funcionaría con cualquier cosa que derive de Collection, y que se utiliza un ArrayList para mantener un común denominador solamente.

El primer método proporciona una forma se rellenar la colección con datos de prueba, en esta caso enteros convertidos a cadenas. El segundo método será utilizado con bastante frecuencia a partir de ahora.

Las dos versiones de nuevaColeccion() crean ArrayList conteniendo diferente conjunto de datos que devuelven como objetos Collection, está claro que no se utiliza ningún otro interfaz diferente de Collection.

El método print() también se usará a menudo a partir de ahora, y lo que hace es moverse a través de la Colección utilizando un Iterator, que cualquier Collection puede generar, y funciona con Listas, Conjuntos y Mapas.

El método main() se usa simplemente para llamar a los métodos de la Colección.

Listas

Hay varias implementaciones de List, siendo ArrayList la que debería ser la elección por defecto, en caso de no tener que utilizar las características que proporcionan las demás implementaciones.

List (interfaz)

La ordenación es la característica más importante de una Lista, asegurando que los elementos siempre se mantendrán en una secuencia concreta. La Lista incorpora una serie de métodos a la Colección que permiten la inserción y borrar de elementos en medio de la Lista. Además, en la Lista Enlazada se puede generar un ListIterator para moverse a través de las lista en ambas direcciones.

ArrayList

Es una Lista volcada en un Array. Se debe utilizar en lugar de Vector como almacenamiento de objetos de propósito general. Permite un acceso aleatorio muy rápido a los elementos, pero realiza con bastante lentitud las operaciones de insertado y borrado de elementos en medio de la Lista. Se puede utilizar un ListIterator para moverse hacia atrás y hacia delante en la Lista, pero no para insertar y eliminar elementos.

LinkedList

Proporciona un óptimo acceso secuencial, permitiendo inserciones y borrado de elementos de en medio de la Lista muy rápidas. Sin embargo es bastante lento el acceso aleatorio, en comparación con la ArrayList. Dispone además de los métodos addLast(),

getFirst(), getLast(), removeFirst() y removeLast(), que no están definidos en ningún interfaz o clase base y que permiten utilizar la Lista Enlazada como una Pila, una Cola o una Cola Doble.

En el ejemplo , cubren gran parte de las acciones que se realizan en las Listas, como moverse con un Iterator, cambiar elementos, ver los efectos de la manipulación de la Lista y realizar operaciones sólo permitidas a las Listas Enlazadas.

En testBasico() y moverIter() las llamadas se hacen simplemente para mostrar la sintaxis correcta, y aunque se recoge el valor devuelto, no se usa para nada. En otros casos, el valor devuelto no es capturado, porque no se utiliza normalmente. No obstante, el lector debe recurrir a la documentación de las clases para comprobar el uso de cualquier método antes de utilizarlo.

Sets

Set tiene exactamente el mismo interfaz que Collection, y no hay ninguna funcionalidad extra, como en el caso de las Listas. Un Set es exactamente una Colección, pero tiene utilizada en un entorno determinado, que es ideal para el uso de la herencia o el polimorfismo. Un Set sólo permite que exista una instancia de cada objeto.

A continuación se muestran las diferentes implementaciones de Set, debiendo utilizarse HashSet en general, a no ser que se necesiten las características proporcionadas por alguna de las otras implementaciones.

Set (interfaz)

Cada elemento que se añada a un Set debe ser único, ya que el otro caso no se añadirá porque el Set no permite almacenar elementos duplicados. Los elementos incorporados al Conjunto deben tener definido el método equals(), en aras de establecer comparaciones para eliminar duplicados. Set tiene el mismo interfaz que Collection, y no garantiza el orden en que se encuentren almacenados los objetos que contenga.

HashSet

Es la elección más habitual, excepto en Sets que sean muy pequeños. Debe tener definido el método hashCode().

ArraySet

Un Set encajonado en un Array. Esto es útil para Sets muy pequeños, especialmente aquellos que son creados y destruidos con frecuencia. Para estos pequeños Sets, la creación e iteración consume muchos menos recursos que en el caso del HashSet. Sin embargo, el rendimiento es muy malo en el caso de Sets con gran cantidad de elementos.

TreeSet

Es un Set ordenado, almacenado en un árbol balanceado. En este caso es muy fácil extraer una secuencia ordenada a partir de un Set de este tipo.

El ejemplo , no muestra todo lo que se puede hacer con un Set, sino que como Set es una Collection, y las posibilidades de las Colecciones ya se han visto, pues el ejemplo se limita a mostrar aquellas cosas que son particulares de los Sets.

```
import java.util.*;

public class java423 {
    public static void testVisual( Set a ) {
        java421.fill( a );
        java421.fill( a );
        java421.fill( a );
        java421.print( a ); // No permite Duplicados!

        // Se añade otro Set al anterior
        a.addAll( a );
        a.add( "uno" );
        a.add( "uno" );
        a.add( "uno" );
        java421.print( a );

        // Buscamos ese elemento
        System.out.println(  "a.contains(\"uno\"):  "+a.contains(  "uno"  )
);
    }

    public static void main( String args[] ) {
        testVisual( new HashSet() );
        testVisual( new ArraySet() );
        }
    }
```

Aunque se añaden valores duplicados al Set, a la hora de imprimirlos, se puede observar que solamente se acepta una instancia de cada valor. Cuando se ejecuta el ejemplo, se observa también que el orden que mantiene el HashSet es diferente del que presenta el ArraySet, ya que cada uno tiene una forma de almacenar los elementos para la recuperación posterior. El ArraySet mantiene los elementos ordenados, mientras que el HashSet utiliza sus propias funciones para que las búsquedas sean muy rápidas. Cuando el lector cree sus propios tipos de estructuras de datos, deberá prestar atención porque un Set necesita alguna forma de poder mantener el orden de los elementos que lo integran, como se muestra en el ejemplo siguiente, .

Las definiciones de los métodos equals() y hashCode() son semejantes a las de ejemplos anteriores. Se debe definir equals() en ambos casos, mientras que hashCode() solamente es necesario si la clase corresponde a un HashSet, que debería ser la primera elección a la hora de implementar un Set.

Mapas

Los Mapas almacenan información en base a parejas de valores, formados por una clave y el valor que corresponde a esa clave.

Map (interfaz)

Mantiene las asociaciones de pares clave-valor, de forma que se puede encontrar cualquier valor a partir de la clave correspondiente.

HashMap

Es una implementación basada en una tabla hash. Proporciona un rendimiento muy constante a la hora de insertar y localizar cualquier pareja de valores; aunque este rendimiento se puede ajustar a través de los constructores que permite fijar la capacidad y el factor de carga de la tabla hash.

ArrayMap

Es un Mapa circunscrito en un Array. Proporciona un control muy preciso sobre el orden de iteración. Está diseñado para su utilización con Mapas muy pequeños, especialmente con aquellos que se crean y destruyen muy frecuentemente. En este caso de Mapas muy pequeños, la creación e iteración consume muy pocos recursos del sistema, y muchos menos que el HashMap. El rendimiento cae estrepitosamente cuando se intentan manejar Mapas grandes.

TreeMap

Es una implementación basada en un árbol balanceado. Cuando se observan las claves o los valores, se comprueba que están colocados en un orden concreto, determinado por Comparable o Comparator, que ya se verán. Lo importante de un TreeMap es que se pueden recuperar los elementos en un determinado orden. TreeMap es el único mapa que define el método subMap(), que permite recuperar una parte del árbol solamente.

El ejemplo contiene dos grupos de datos de prueba y un método rellena() que permite llenar cualquier mapa con cualquier array de dos dimensiones de Objects.

Los métodos printClaves(), printValores() y print() no son solamente unas cuantas utilidades, sino que demuestran como se pueden generar Colecciones que son vistas de un Mapa. El método keySet() genera un Set que contiene las claves que componen el Mapa; en este caso, es tratado como una Colección. Tratamiento similar se da a values(), que genera una List conteniendo todos los valores que se encuentran almacenados en el Mapa. Observar que las claves deben ser únicas, mientras que los valores pueden contener elementos duplicados. Debido a que las Colecciones son dependientes del Mapa, al representar solamente una vista concreta de parte de los datos del Mapa, cualquier cambio en una Colección se reflejará inmediatamente en el Mapa asociado.

El método print() recoge el Iterator producido por entries() y lo utiliza para imprimir las parejas de elementos clave-valor. El resto del ejemplo proporciona ejemplos muy simples de cada una de las operaciones permitidas en un Mapa y prueba cada tipo de Mapa.

A la hora de crear Mapas propios, el lector debe tener en cuenta las mismas recomendaciones que anteriormente se proporcionaban en el caso de los Sets.

Nuevas colecciones dos

Elegir una Implementación

Si se observa detenidamente el diagrama de herencia de las nuevas colecciones, se puede comprobar que en realidad hay solamente tres Colecciones: Map, List y Set, y solamente dos o tres implementaciones de cada interfaz. En el caso de tener la necesidad de utilizar la funcionalidad ofrecida por un interfaz determinado, el problema viene a la hora de seleccionar que implementación particular se debe emplear.

Para comprender la respuesta, se debe tener muy presente que cada una de las diferentes implementaciones tiene sus características propias, sus virtudes y sus defectos. Por ejemplo, en el diagrama se puede observar que una de las principales características de Hashtable, Vector y Stack es que son clases ya empleadas, es decir, que ya están totalmente soportadas en el JDK 1.1, por lo que los programas que las utilicen funcionarán perfectamente con los navegadores y máquinas virtuales que están actualmente en el mercado. Pero, por otro lado, si se va a utiliza código que utilice características del JDK 1.2, es mejor no utilizar las clases anteriores.

La distinción entre las nuevas colecciones y las anteriores viene dada por la estructura de datos en que se basa cada una; es decir, la estructura de datos que físicamente implementa el interfaz deseado. Esto quiere decir que, por ejemplo, ArrayList, LinkedList y Vector (que viene a ser equivalente a un ArrayList), todos implementan el interfaz List, con lo cual un programa debe producir los mismos resultados, independientemente del que se esté utilizando. Sin embargo, ArrayList (y Vector) están basados en un array, mientras que LinkedList está implementado en la forma habitual de una lista doblemente enlazada, con los objetos conteniendo información de qué elemento está antes y después en la lista. Por ello, si se van a realizar muchas inserciones y borrado de elementos en medio de la lista, la LinkedList es una opción mucho más adecuada que cualquier otra. Y en todos los demás casos, probablemente un ArrayList sea mucho más rápido.

Otro ejemplo, un Set puede ser implementado a partir de un ArraySet o de un HashSet. Un ArraySet está basado en un ArrayList y está diseñado para soportar solamente pequeñas cantidades de elementos, especialmente en situaciones en las que se estén creando y destruyendo gran cantidad de objetos Set. Sin embargo, si el Set va a contener gran cantidad de elementos, el rendimiento del ArraySet se viene abajo muy rápidamente. A la hora de escribir un programa que necesite un Set, la elección por defecto debería ser un HashSet, y cambiar a un ArraySet solamente en aquellos casos en que se observe un rendimiento muy precario o donde las optimizaciones indiquen que es adecuado.

Operaciones No Soportadas

Es posible convertir un array en una Lista utilizando el método estático Arrays.toList(), tal como se muestra en el ejemplo .

```
import java.util.*;

public class java426 {
    private static String s[] = {
        "uno", "dos", "tres", "cuatro", "cinco",
        "seis", "siete", "ocho", "nueve", "diez",
        };
    static List a = Arrays.toList( s );
    static List a2 = Arrays.toList( new String[] { s[3],s[4],s[5] } );

    public static void main( String args[] ) {
        java421.print( a ); // Iteración
        System.out.println(
```

```
            "a.contains("+s[0]+") = "+a.contains(s[0]) );
        System.out.println(
            "a.containsAll(a2) = "+a.containsAll(a2) );
        System.out.println( "a.isEmpty() = "+a.isEmpty() );
        System.out.println(
            "a.indexOf("+s[5]+") = "+a.indexOf(s[5]) );

        // Movimientos hacia atrás
        ListIterator lit = a.listIterator( a.size() );
        while( lit.hasPrevious() )
            System.out.print( lit.previous() );
        System.out.println();

        // Fija algunos elementos a valores diferentes
        for( int i=0; i < a.size(); i++ )
            a.set( i,"47" );
        java421.print( a );

        // Lo siguente compila, pero no funciona
        lit.add( "X" ); // Operación no soportada
        a.clear(); // No soportada
        a.add( "once" ); // No soportada
        a.addAll( a2 ); // No soportada
        a.retainAll( a2 ); // No soportada
        a.remove( s[0] ); // No soportada
        a.removeAll( a2 ); // No soportada
        }
}
```

El lector podrá descubrir que solamente una parte de los interfaces de Collection y List están actualmente implementados. El resto de los métodos provocan la aparición de un desagradable mensaje de algo que se llama UnsupportedOperationException. La causa es que el interfaz Collection, al igual que los otros interfaces en la nueva librería de colecciones, contienen métodos opcionales, que pueden estar o no soportados en la clase concreta que implemente ese interfaz. La llamada a un método no soportado hace que aparezca la excepción anterior para indicar el error de programación. Las líneas siguientes reproducen la ejecución del ejemplo, en donde se observa el mensaje que genera el íntérprete Java a la hora de realizar la operación no soportada sobre la Colección.

```
C:\>java java426
uno dos tres cuatro cinco seis siete ocho nueve diez
a.contains(uno) = true
a.containsAll(a2) = true
a.isEmpty() = false
a.indexOf(seis) = 5
dieznueveochosieteseiscincocuatrotresdosuno
47 47 47 47 47 47 47 47 47 47
java.lang.UnsupportedOperationException
at java.util.AbstractList.add(AbstractList.java:139)
at java.util.AbstractList$ListItr.add(AbstractList.java:523)
at java426.main(java426.java:57)
```

Me imagino la cara de incredulidad del lector. Lo anterior echa por tierra todas las promesas que se hacían con los interfaces, ya que se supone que todos los métodos definidos en interfaces y clases base tienen algún significado y están dotados de código en sus implementaciones; pero es más, en este caso no solamente no hacen nada, sino que detienen la ejecución del programa. Bueno, la verdad es que no todo está tan mal; con Collection, List, Set o Map, el compilador todavía restringe la llamada a métodos, permitiendo solamente la llamada a métodos que se encuentren en el interfaz correspondiente, y además, muchos de los métodos que tienen una Collection como

argumento solamente leen desde esa Colección, y todos los métodos de lectura son no opcionales.

Esta posibilidad, o característica, evita una explosión de interfaces en el diseño. Otros diseños de librerías de colecciones siempre parecen tener al final una plétora de interfaces que describen cada una de las variaciones del tema principal, lo que hace que sean difíciles de aprender. Además que no es posible recoger todos los casos especiales en interfaces, ya que siempre hay alguien dispuesto a inventarse un nuevo interfaz. La posibilidad de la operación no soportada consigue una meta importante en las nuevas colecciones: las hace muy fáciles de aprender y su uso es muy simple. Pero para que esto funcione, hay que tener en cuenta dos cuestiones:

1. La UnsupportedOperationException debe ser un evento muy raro. Es decir, en la mayoría de las clases todos los métodos deben funcionar, y solamente en casos muy especiales una operación podría no estar soportada. Esto es cierto en las nuevas colecciones, ya que en el 99 por ciento de las veces, tanto ArrayList, LinkedList, HashSet o HashMap, así como otras implementaciones concretas, soportan todas las operaciones. El diseño proporciona una puerta falsa si se quiere crear una nueva Collection sin asignar significado a todos los métodos del interfaz Collection, y todavía está en la librería.

2. Cuando una operación esté no soportada, sería muy adecuado que apareciese en tiempo de compilación la UnsupportedOperationException, antes de que se genere el programa que se vaya a dar al cliente. Después de todo, una excepción indica un error de programación: hay una clase que se está utilizando de forma incorrecta.

En el ejemplo anterior, Arrays.toList() genera una Lista que está basada en un array de tamaño fijo; por lo tanto, tiene sentido el que solamente estén soportadas aquellas operaciones que no alteran el tamaño del array. Si, de otro modo, fuese necesario un interfaz nuevo para recoger este distinto comportamiento, llamado TamanoFijoList, por ejemplo; se estaría dejando la puerta abierta al incremento de la complejidad de la librería, que se volvería insoportable cuando alguien ajeno intentase utilizarla al cabo del tiempo, por el montón de interfaces que tendría que aprenderse.

La documentación de todo método que tenga como argumento Collection, List, Set o Map, debería especificar cuales de los métodos opcionales deben ser implementados. Por ejemplo, la ordenación necesita los métodos set() e Iterator.set(), pero no necesita para nada los métodos add() y remove().

Ordenación y Búsqueda

El JDK 1.2 incorpora utilidades para realizar ordenaciones y búsquedas tanto en arrays como en listas. Estas utilidades son métodos estáticos de dos nuevas clases: Arrays para la ordenación y búsqueda en arrays y Collections para la ordenación y búsqueda en Listas.

Arrays

La clase Arrays tiene los métodos sort() y binarySearch() sobrecargados para todos los tipos básicos de datos, incluidos String y Object. El ejemplo muestra la ordenación y

búsqueda en un array de elementos de tipo byte, para todos los demás tipos básicos se haría de forma semejante, y para un array de String.

La primera parte de la clase contiene utilidades para la generación de objetos aleatorios de tipo String, utilizando un array de caracteres desde el cual se pueden seleccionar las letras. El método cadAleat() devuelve una cadena de cualquier longitud, y cadsAleat() crea un array de cadenas aleatorias, dada la longitud de cada String y el número de elementos que va a tener el array. Los dos métodos print() simplifican la presentación de los datos. El método Random.nextBytes() rellena el array argumento con bytes seleccionados aleatoriamente; en los otros tipos de datos básicos no hay un método semejante.

Una vez construido el array se puede observar que se hace una sola llamada al sort() o a binarySearch(). Hay una cuestión importante respecto a binarySearch(), y es que, si no se llama al método sort() antes de llamar a binarySearch() para realizar la búsqueda, se pueden producir resultados impredecibles, incluso la entrada en un bucle infinito.

La ordenación y búsqueda en el caso de String parece semejante, pero a la hora de ejecutar el programa se observa una cosa interesante: la ordenación es alfabética y lexicográfica, es decir, las letras mayúsculas preceden a las letras minúsculas. Así, todas las letras mayúsculas están al principio de la lista, seguidas de las letras minúsculas, luego 'Z' está antes que 'a'.

Comparable y Comparator

¿Qué pasa si eso no es lo que se quiere? Por ejemplo, si se quieres generar el índice de un libro, esa ordenación no es admisible, ya que lo lógico es que la 'a' vaya después de la 'A'.

Y ¿qué pasa en un array de elementos de tipo Object? ¿Qué determina el orden en que se encuentran dos elementos de este tipo? Desgraciadamente, los diseñadores originales de Java no consideraron que este fuese un problema importante, porque sino lo hubiesen incluido en la clase raíz Object. Como resultado, el orden en que se encuentran los elementos Object ha de ser impuesto desde fuera, y la nueva librería de colecciones proporciona una forma estándar de hacerlo, que es casi tan buena como si la hubiesen incluido en Object.

Hay un método sort() para array de elementos Object (y String, desde luego, es un Object) que tiene un segundo argumento: un objeto que implementa el interfaz Comparator, que es parte de la nueva librería, y realiza comparaciones a través de su único método *compare*(). Este método coge los dos objetos que van a compararse como argumentos y devuelve un entero negativo si el primer argumento es menor que el segundo, cero si son iguales y un entero positivo si el primer argumento es más grande que el segundo. Sabiendo esto, en el ejemplo , se vuelve a implementar la parte correspondiente al array de elementos String del ejemplo anterior para que la ordenación sea alfabética.

```
import java.util.*;

public class java428 implements Comparator {
    public int compare( Object obj1,Object obj2 ) {
        // Suponemos que solamente vamos a utilizar cadenas
        String s1 = ( (String)obj1).toLowerCase();
        String s2 = ( (String)obj2).toLowerCase();
```

```
        return( s1.compareTo( s2 ) );
        }


    public static void main( String args[] ) {
        String s[] = java427.cadsAleat( 4,10 );
        java427.print( s );
        java428 ac = new java428();
        Arrays.sort( s,ac );
        java427.print( s );

        // Se debe utilizar un Comparador para realizar la búsqueda
        int loc = Arrays.binarySearch( s,s[3],ac );
        System.out.println( "Posicion de "+s[3]+" = "+loc );
        }
    }
```

Haciendo el moldeo a String, el método compare() implícitamente se asegura que solamente se están utilizando objetos de tipo String. Luego se fuerzan los dos Strings a minúsculas y el método String.compareTo() devuelve el resultado que se desea.

A la hora de utilizar un Comparator propio para llamar a sort(), se debe utilizar el mismo Comparator cuando se vaya a llamar a binarySearch(). La clase Arrays tiene otro método sort() que toma un solo argumento: un array de Object, sin ningún Comparator. Este método también puede comparar dos Object, utilizando el método natural de comparación que es comunicado a la clase a través del interfaz Comparable. Este interfaz tiene un único método, compareTo(), que compara el objeto con su argumento y devuelve negativo, cero o positivo dependiendo de que el objeto sea menor, igual o mayor que el argumento. El ejemplo , es un programita muy sencillo que ilustra esta circunstancia.

Listas

Una Lista puede ser ordenada y se puede buscar en ella del mismo modo que en un array. Los métodos estáticos para ordenar y buscar en una Lista están contenidos en la clase Collections, pero tienen un formato similar a los correspondientes de la clase Arrays: sort(List) para ordenar una Lista de objetos que implementen el interfaz Comparable, binarySearch(List,Object) para buscar un objeto en una lista, sort(List,Comparator) para ordenar una Lista utilizando un Comparator y binarySearch(List,Object,Comparator) para buscar un objeto en esa lista. Seguro que en nuevas versiones del JDK se irán incorporando nuevos métodos para realizar acciones específicas, como por ejemplo, ya está anunciado el método stableSort() para realizar mezclas ordenadas. El ejemplo se basa en los dos anteriores para demostrar el uso de las herramientas de ordenación sobre la clase Collections.

El uso de estos métodos es idéntico a los correspondientes de la clase Arrays, pero utilizando una Lista en lugar de un Array. El TreeMap también puede ordenar sus objetos en función de Comparable y Comparator.

Utilidades

Hay otras utilidades que pueden resultar interesantes en la clase Collections, como por ejemplo:

enumeration(Collection)

Devuelve una Enumeration al viejo estilo a partir del argumento que se le pasa

max(Collection)
min(Collection)

Devuelven el elemento máximo o el mínimo de la colección que se le pasa como argumento, utilizando el método natural de comparación entre los objetos de la Colección

max(Collection,Comparator)
min(Collection,Comparator)

Devuelven el elemento máximo o el mínimo de la colección que se le pasa como argumento, utilizando el Comparator

nCopies(int n,Object o)

Devuelve una Lista de tamaño n cuyos handles apuntan a o

subList(List,int min,int max)

Devuelve una nueva Lista a partir de la Lista que se le pasa como argumento que es una porción de esta última cuyos índices comienzan en el elemento min y terminan en el elemento max

Tanto min() como max() trabajan con objetos de tipo Collection, no con List, por lo tanto es indiferente si la Colección está ordenada o no. Como ya se indicó antes, es necesario llamar a sort() en una Lista o array antes de realizar una búsqueda con binarySearch().

Colecciones o Mapas de Sólo Lectura

A menudo es conveniente crear una versión de sólo lectura de una Colección o un Mapa. La clase Collections permite que se haga esto pasando el contenedor original a un método que devuelve una versión de sólo lectura. Hay cuatro variaciones sobre este método, una para cada tipo de Collection (en caso de no querer tratar la Colección de forma más específica), List, Set y Map. El ejemplo muestra la forma en que se pueden construir versiones de sólo lectura de cada uno de ellos.

```
import java.util.*;

public class java431 {
    public static void main( String args[] ) {
        Collection c = new ArrayList();
        java421.fill( c ); // Añade datos útiles
        c = Collections.unmodifiableCollection( c );
        java421.print( c ); // La lectura es correcta
        //c.add( "uno" ); // No se puede cambiar
        List a = new ArrayList();
        java421.fill( a );
        a = Collections.unmodifiableList( a );
        ListIterator lit = a.listIterator();
```

```
        System.out.println( lit.next() ); // La lectura es correcta
        //lit.add( "uno" ); // No se puede cambiar

        Set s = new HashSet();
        java421.fill( s );
        s = Collections.unmodifiableSet( s );
        java421.print( s ); // La lectura es correcta
        //s.add( "uno" ); // No se puede cambiar
        Map m = new HashMap();
        java425.fill( m,java425.datosPrueba1 );
        m = Collections.unmodifiableMap( m );
        java425.print( m ); // La lectura es correcta
        //m.put( "CV","Caballo de Vapor" ); // No se puede cambiar
        }
    }
```

En cada caso, se debe rellenar el contenedor con datos significativos antes de hacerlo de sólo lectura. Una vez que está cargado, el mejor método es reemplazar el handle existente con el que genera la llamada al no modificable. De esta forma, no se corre el riego de cambiar accidentalmente el contenido. Por otro lado, esta herramienta permite que se pueda mantener un contenedor modificable como privado dentro de la clase y devolver un handle de sólo lectura hacia él a través de un método. De este modo se puede realizar cualquier cambio desde la propia clase, pero el resto del mundo solamente pueden leer los datos.

La llamada al método no modificable para un tipo determinado no genere ninguna comprobación en tiempo de compilación, pero una vez que se haya realizado la transformación, la llamada a cualquier método que intente modificar el contenido del contenedor obtendrá por respuesta una excepción de tipo UnsupportedOperationException.

Si en el ejemplo se descomenta la línea que añade un elemento a la primera de las Colecciones

```
c.add( "uno" ); // No se puede cambiar
```

y se vuelve a compilar el código fuente resultante, a la hora de ejecutar el programa, se obtendrá un resultado semejante al que reproducen las líneas siguientes

```
C:\>java java431
0 1 2 3 4 5 6 7 8 9
java.lang.UnsupportedOperationException
    at
java.util.Collections$UnmodifiableCollection.add(Collections.java:583)
at java431.main(java431.java:33)
```

Colecciones o Mapas Sincronizados

La palabra reservada synchronized es una parte muy importante de la programación multihilo, o multithread, que es un tema muy interesante y se tratará abundantemente en otra sección; aquí, en lo que respecta a las Colecciones basta decir que la clase Colletions contiene una forma de sincronizar automáticamente un contenedor entero, utilizando una sintaxis parecida a la del caso anterior de hacer un contenedor no modificable. El ejemplo es una muestra de su uso.

```
import java.util.*;

public class java432 {
    public static void main( String args[] ) {
        Collection    c    =    Collections.synchronizedCollection(    new
ArrayList() );
```

CONCEPTOS BASICOS DE JAVA

Objetos

Antes de entrar en el estudio de las Clases en Java, hay que presentar a los objetos, que son las instanciaciones de una clase. En este caso, haciendo un símil con la vida real, estamos colocando el carro delante del caballo, pero se supone que el lector tiene el bagaje suficiente como para entender lo que se expondrá, sin necesidad de conocer en profundidad lo que es y cómo funciona una clase. Además, esto permitirá obviar muchas cosas cuando se entre en el estudio de las clases. Y, en última instancia, siempre puede el lector pasar a la sección siguiente y luego volver aquí.

Un objeto es la instanciación de una clase y tiene un estado y un funcionamiento. El estado está contenido en sus variables (variables miembro), y el funcionamiento viene determinado por sus métodos. Las variables miembro pueden ser variables de instancia o variables de clase. Se activa el funcionamiento del objeto invocando a uno de sus métodos, que realizará una acción o modificará su estado, o ambas cosas. Cuando un objeto ya no se usa en C++, se destruye y la memoria que ocupaba se libera y es devuelta al Sistema. Cuando un objeto ya no se usa en Java, simplemente se anula. Posteriormente, el reciclador de memoria (garbage collector) puede recuperar esa memoria para devolverla al Sistema.

Las afirmaciones anteriores son un compendio de lo que se puede esperar de un objeto. A continuación se verá más en detalle cómo es la creación, el uso y la destrucción de un objeto. Se utilizará C++ como punto de comparación para dar el salto a Java, en deferencia a los muchos lectores que conocen los entresijos de los objetos en C++, de ahí que también que se inicie el estudio de la vida de los objetos suponiendo que se dispone de acceso a clases ya existentes o que se tiene conocimiento de cómo crearlas, aunque posteriormente se vuelva sobre ello.

Creación de Objetos

Un objeto es (de nuevo) una instancia de una clase. Tanto en Java como en C++, la creación de un objeto se realiza en tres pasos (aunque se pueden combinar):

- Declaración, proporcionar un nombre al objeto
- Instanciación, asignar memoria al objeto
- Inicialización, opcionalmente se pueden proporcionar valores iniciales a las variables de instancia del objeto

Cuando se trata de Java, es importante reconocer la diferencia entre objetos y variables de tipos básicos, porque en C++ se tratan de forma similar, cosa que no ocurre en Java.

Tanto en Java como en C++ la declaración e instanciación de una variable de tipo básico, utiliza una sentencia que asigna un nombre a la variable y reserva memoria para almacenar su contenido:

```
int miVariable;
```

En los dos lenguajes se puede inicializar la variable con un valor determinado en el momento de declararla, es decir, podemos resumir los tres pasos anteriormente citados de declaración, instanciación e inicialización, en una sola sentencia:

```
int miVariable = 7;
```

Y, lo más importante, este sucede en tiempo de compilación.

Java no permite la instanciación de variables de tipos básicos en memoria dinámica, aunque hay una serie de objetos que coinciden con los tipos básicos y que pueden utilizarse para este propósito. Java tampoco permite la instanciación de objetos en memoria estática.

Cuando C++ instancia un array de objetos en memoria dinámica, lo que hace es instanciar el array de objetos y devolver un puntero al primer objeto. En Java, cuando se instancia un array (que siempre ha de ser en memoria dinámica), se instancia un array de referencias, o punteros (llámese como se quiera) a los objetos. En Java, es necesario utilizar múltiples veces el operador new para instanciar un array de objetos, mientras que en C++ sólo es necesario su uso una única vez. El siguiente código de ejemplo, , corresponde a la aplicación Java que realiza el mismo trabajo que el programa C++ visto antes, pero con las restricciones que se acaban de enumerar. Se ha incorporado la presentación en pantalla no sólo del dato al que apunta un objeto, sino también de la dirección en la que se encuentra almacenado ese dato (la dirección del objeto). La salida del programa sería:

```
C:\> java java501
miVariableInt contiene 6
miPunteroObj contiene java501@208741
miPunteroObj apunta a 12
miArrayPunterosObj contiene java501@20875f java501@208760 java501@208761
miArrayPunterosObj apunta a 13 14 15
```

El formato en que se presenta un objeto en Java, cuando el compilador no conoce el tipo de que se trata, consiste en el identificador de la clase y el valor hexadecimal de la dirección.

En Java no siempre es necesaria la declaración de un objeto (darle un nombre). En el siguiente ejemplo, , se instancia un nuevo objeto que se usa en una expresión, sin previamente haberlo declarado.

```
import java.util.*;

class java502 {
    public static void main( String args[] ) {
        System.out.println( new Date() );
        }
    }
```

La inicialización de un objeto, se puede realizar utilizando las constantes de la clase, de forma que un objeto de un mismo tipo puede ser declarado e inicializado de formas diferentes.

Utilización de Objetos

Una vez que se tiene declarado un objeto con sus variables y sus métodos, podemos acceder a ellos para que el uso para el que se ha creado ese objeto entre en funcionamiento.

Para acceder a variables o métodos en Java, se especifica el nombre del objeto y el nombre de la variable, o método, separados por un punto (.).

```
fecha.displayFecha( dateContainer( "29/07/97" );
```

Destrucción de Objetos

El programador Java no necesitará preocuparse más de devolver la memoria, ese bien tan preciado, al sistema operativo. Eso se realizará automáticamente de la mano del reciclador de basura, que también es bien conocido como garbage collector.

Sin embargo, hay una mala noticia. Y es que en Java no hay soporte para algo parecido a un destructor (en C++), que pueda garantizar su ejecución cuando el objeto deje de existir. Por lo tanto, excepto la devolución de memoria al sistema operativo; la liberación de los demás recursos que el objeto utilizase vuelve a ser responsabilidad del programador. Pero, algo se ha ganado, ya no debería aparecer más el fatídico mensaje de "No hay memoria suficiente", o su versión inglesa de "Out of Memory".

Liberación de Memoria

El único propósito para el que se ha creado el reciclador de memoria es para devolver al sistema operativo la memoria ocupada por objetos que no es necesaria. Un objeto es blanco del garbage collector para su reciclado cuando ya no hay referencias a ese objeto. Sin embargo, el que un objeto sea elegido para su reciclado, no significa que eso se haga inmediatamente.

El **garbage collector** de Java comprueba todos los objetos, y aquellos que no tienen ninguna (esto es, que no son referenciados por ningún otro objeto) son marcados y liberada la memoria que estaban consumiendo.

El garbage collector es un thread, o hilo de ejecución, de baja prioridad que puede ejecutarse síncrona o asíncronamente, dependiendo de la situación en que se encuentre el sistema Java. Se ejecutará síncronamente cuando el sistema detecta poca memoria o en respuesta a un programa Java. El programador puede activar el garbage collector en cualquier momento llamando al método System.gc(). Se ejecutará asíncronamente cuando el sistema se encuentre sin hacer nada (idle) en aquellos sistemas operativos en los que para lanzar un thread haya que interrumpir la ejecución de otro, como es el caso de Windows 95/NT.

No obstante la llamada al sistema para que se active el garbage collector no garantiza que la memoria que consumía sea liberada.

A continuación se muestra un ejemplo sencillo de funcionamiento del garbage collector:

```
String s;                 // no se ha asignado memoria todavía
s = new String( "abc" ); // memoria asignada
s = "def";                // se ha asignado nueva memoria
                          // (nuevo objeto)
```

Más adelante veremos en detalle la clase String, pero una breve descripción de lo que hace esto es; crear un objeto String y rellenarlo con los caracteres "abc" y crear otro (nuevo) String y colocarle los caracteres "def".

En esencia se crean dos objetos:

- Objeto String "abc"
- Objeto String "def"

Al final de la tercera sentencia, el primer objeto creado de nombre s que contiene "abc" se ha salido de alcance. No hay forma de acceder a él. Ahora se tiene un nuevo objeto llamado s y contiene "def". Es marcado y eliminado en la siguiente iteración del thread reciclador de memoria.

El Método finalize()

Antes de que el reciclador de memoria reclame la memoria ocupada por un objeto, se puede invocar el método finalize(). Este método es miembro de la clase Object, por lo tanto, todas las clase lo contienen. La declaración, por defecto, de este método es:

```
protected void finalize() {
    }
```

Para utilizar este método, hay que sobrecargarlo, proporcionando el código que contenga las acciones que se desee ejecutar antes de liberar la memoria consumida por el objeto. Aunque se puede asegurar que el método finalize() se invocará antes de que el garbage collector reclame la memoria ocupada por el objeto, no se puede asegurar cuando se liberará esa memoria; e incluso, si esa memoria será liberada.

El ejemplo java504 trata de ilustrar todo esto.

```
/**
* Este ejemplo intenta demostrar que la actuacion del reciclador de
* de memoria, o garbage collector, tiene un funcionamiento asincrono e
* impredecible.Se instancian un numero de objetos que son inmediatamente
* seleccionados para su reciclado.
* Se instancia mas de 1000 objetos antes de que llegue la recoleccion
* y terminacion del programa.
*/
class Obj {
   static boolean recicladorActivo = false;
   static boolean flagSalida = false;
   static int objCreados = 0;
   static int objFinalizados = 0;
   static int objFinalizadosConSalidaATrue = 0;
   int numObjetos;
   int arrayEnteros[]; // Array utiliza para consumir memoria
   // Constructor parametrizado
   Obj( int contador ) {
       arrayEnteros = new int[500];
       numObjetos = contador;
       objCreados++;
       // La siguiente sentencia indica cuando se crea el objeto #1000
       if( numObjetos == 1000 )
           System.out.println( "Creado el objeto #1000" );
       }
   // El siguiente metodo sobreescrito se ejecuta de forma asincrona
   // en relacion con el programa principal. Sera llamado por el
   // reciclador de memoria una vez por cada uno de los objetos
   // antes de que la memoria ocupada por ese objeto sea reclamada


   protected void finalize() {
       // Contador del numero de objetos finalizados
       objFinalizados++;
```

```
        // En la primera llamada al metodo finalize(), fijamos ela bandera
        // que indica que el reciclador de memoria ha entrado en accion
        if( !recicladorActivo ) {
            recicladorActivo = true;
            System.out.println(
                "Iniciado el reciclado en el Objeto numero "
                + numObjetos );
            }
        // Se fija la bandera de salida cuando el reciclador finaliza el
        // objeto #1000. Esto no tendra impacto alguno sobre el programa
        // principal hasta que el hilo principal tome el control desde el
        // hilo del reciclador. El orden en que se finalizan los objetos,
        // no es necesariamente el mismo orden en el que se han creado,
        // sino que el orden y el momento en que se recicla un objeto,
        // viene determinado por el reciclador de memoria
        if( numObjetos == 1000 ) {
            System.out.println(
                "Finalizado el objeto #1000. Fija la salida a true" );
            flagSalida = true;
            }
        // Contador de los objetos finalizados despues de haber fijado
        // la bandera de salida y antes de que el control sea devuelto
        // al hilo principal del programa
        if( flagSalida )
            objFinalizadosConSalidaATrue++;
        }
    }
public class java504 {
    volatile static int objContador = 0;
    public static void main( String args[] ) {
        // El programa termina en algun momento despues de que el metodo
        // finalize() llamado por el reciclador, fije la bandera de
        // salida a true, pero no antes de que el hilo principal de
        // ejecucion del programa retome el control desde el hilo de
        // ejecucion del reciclador. Los objetos que son creados en
        // este bucle no estan asignados a una variable de referencia y
        // ademas son marcados inmediatamente para su reciclado
        while( !Obj.flagSalida ) {
            // Se van instanciando nuevos objetos mientras la bandera
            // de salida este a falso
            new Obj( objContador++ );
            }
        System.out.println( "Objetos totales creados =   "
            + Obj.objCreados );
        System.out.println( "Objetos totales finalizados = "
            + Obj.objFinalizados );
        System.out.println( + Obj.objFinalizadosConSalidaATrue +
            " objetos fueron finalizados despues de fijar la Salida" );
        }
    }
```

Este programa instancia una gran cantidad de objetos y monitoriza la actividad del garbage collector. En concreto, el programa instancia los objetos y los hace seleccionables inmediatamente por el garbage collector para su reciclado (no asignándolos a variable alguna, con lo cual no hay ninguna referencia a ellos, que es la condición para que sean marcados para su reciclaje). Cuando el objeto 1000 es finalizado, se fija el flagSalida, que está monitorizado en el thread principal, para terminar el programa cuando sea true. Sin embargo, si se observa la ejecución, el garbage collector finaliza 86 objetos más, después de haberse fijado el flagSalida a true antes de que se devuelva el control al thread principal y pueda concluir el programa. La salida por pantalla de la ejecución sería:

```
%java java504
```

```
Iniciado el reciclado en el Objeto numero 0
Creado el objeto #1000
Finalizado el objeto #1000. Fija la salida a true
Objetos totales creados = 1087
Objetos totales finalizados = 1086
86 objetos fueron finalizados despues de fijar la Salida
```

Si se experimenta con distintos valores para el número de objetos creados, se podrá observar un funcionamiento diferente del garbage collector y además, de forma impredecible.

Reiteramos que el método finalize() no es un destructor al uso como en C++. Nunca se puede asegurar cuando se ejecutará, o incluso, si será ejecutado. Por tanto, si se necesita que un objeto libere los recursos que estaba consumiendo, antes de que el programa finalice, no se puede confiar esa tarea al método finalize(), sino que hay que llamar a métodos que realicen esas tareas explícitamente.

Además del método System.gc(), que permite invocar al garbage collector en cualquier instante, también puede se posible forzar la ejecución de los métodos finalize() de los objetos marcados para reciclar, llamando al método:

```
System.runFinalization();
```

Pero, la circunstancia explicada anteriormente sigue vigente, es decir, nadie garantiza que esto libere todos los recursos y, por lo tanto, que se produzca la finalización del objeto.

Si se modifica el programa del ejemplo anterior, para incluir la finalización, tal como se muestra en el ejemplo , se podrá observar esta circunstancia. La clase principal del ejemplo quedaría como muestra el trozo de código que se reproduce a continuación.

```
public class java505 {
    volatile static int objContador = 0;

    public static void main( String args[] ) {
        while( objContador < 1200 ) {
            // Solicita el reciclado cada 120 objetos
            if( (objContador % 120 ) == 0 )
                System.gc();
            new Obj( objContador++ );
            }
        System.out.println( "Objetos totales creados = " + Obj.objCreados
);
            System.out.println( "Objetos totales finalizados = " +
Obj.objFinalizados );
        // Intenta forzar la finalizacion de los objetos
        System.out.println( "Intenta forzar la finalizacion." );
        System.runFinalization();
        System.out.println( "Objetos totales finalizados = " +
Obj.objFinalizados );
        }
    }
```

La salida del programa por pantalla en esta ocasión sería:

```
%java java505
Iniciado el reciclado en el Objeto numero 0
Creado el objeto #1000
Finalizado el objeto #1000. Fija la salida a true
Objetos totales creados = 1200
Objetos totales finalizados = 1010
Intenta forzar la finalizacion.
```

```
Objetos totales finalizados = 1089
```

Como se puede observar, solamente se han finalizado 1080 objetos de los 1200 instanciados. 1010 de ellos fueron finalizados cuando se instanciaron, por lo tanto, parece una acción del garbage collector. Posteriormente, se finalizaron 79, posiblemente debido a la invocación del método de finalización, System.runFinalization().

Clases

Las clases son lo más simple de Java. Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas Java. Todas las acciones de los programas Java se colocan dentro del bloque de una clase o un objeto. Un objeto es una instancia de una clase. Todos los métodos se definen dentro del bloque de la clase, Java no soporta funciones o variables globales. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos. En C la unidad fundamental son los ficheros con código fuente, en Java son las clases. De hecho son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave **import** (equivalente al #include) puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

La definición de una clase consta de dos partes, la declaración y el cuerpo, según la siguiente sintaxis:

```
DeclaracionClase {
    CuerpoClase
    }
```

sin el punto y coma (;) final

La declaración de la clase indica al compilador el nombre de la clase, la clase de la que deriva (su superclase), los privilegios de acceso a la clase (pública, abstracta, final) y si la clase implementa o no, uno o varios interfaces. El nombre de la clase debe ser un identificador válido en Java.

Cada clase Java deriva, directa o indirectamente, de la clase Object. La clase padre inmediatamente superior a la clase que se está declarando se conoce como superclase. Si no se especifica la superclase de la que deriva una clase, se entiende que deriva directamente de la clase Object (definida en el paquete java.lang).

En la declaración de una clase se utiliza la palabra clave **extends** para especificar la superclase, de la forma:

```
class MiClase extends SuperClase {
    // cuerpo de la clase
    }
```

Una clase hereda las variables y métodos de su superclase y de la superclase de esa clase, etc.; es decir, de todo el árbol de jerarquía de clases desde la clase que estamos declarando hasta la raíz del árbol: Object. En otras palabras, un objeto que es instanciado desde una clase determinada, contiene todas las variables y métodos de instancia definidos

para esta clase y sus antecesores; aunque los métodos pueden ser modificados (sobreescritos) en algún lugar.

Una clase puede implementar uno o más interfaces, declarándose esto utilizando la palabra clave **implements**, seguida de la lista de interfaces que implementa, se paradas por coma (,), de la forma:

```
class MiClase extends SuperClase implements MiInterfaz,TuInterfaz {
    // cuerpo de la clase
    }
```

Cuando una clase indique que implementa un interfaz, se puede asegurar que proporciona una definición para todos y cada uno de los métodos declarados en ese interfaz; en caso contrario, el compilador generará errores al no poder resolver los métodos del interfaz en la clase que lo implementa.

Hay cierta similitud entre un interfaz y una clase abstracta, aunque las definiciones de métodos no están permitidas en un interfaz y sí se pueden definir en una clase abstracta. El propósito de los interfaces es proporcionar nombres, es decir, solamente declara lo que necesita implementar el interfaz, pero no cómo se ha de realizar esa implementación; es una forma de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Cuando se implementa un interfaz, los nombres de los métodos de la clase deben coincidir con los nombres de los métodos que están declarados en es interfaz, todos y cada uno de ellos.

El cuerpo de la clase contiene las declaraciones, y posiblemente la inicialización, de todos los datos miembros, tanto variables de clase como variables de instancia, y la definición completa de todos los métodos.

Las variables pueden declararse dentro del cuerpo de la clase o dentro del cuerpo de un método de la clase. Sin embargo, éstas últimas no son variables miembro de la clase, sino variables locales del método en la que están declaradas.

Un objeto instanciado de una clase tiene un estado definido por el valor actual de las variables de instancia y un entorno definido por los métodos de instancia. Es típico de la programación orientada a objetos el restringir el acceso a las variables y proporcionar métodos que permitan el acceso a esas variables, aunque esto no es estrictamente necesario.

Alguna o todas las variables pueden declararse para que se comporten como si fuesen constantes, utilizando la palabra clave **final**.

Los objetos instanciados desde una clase contienen todos los métodos de instancia de esa clase y también todos los métodos heredados de la superclase y sus antecesores. Por ejemplo, todos los objetos en Java heredan, directa o indirectamente, de la clase Object; luego todo objeto contiene los miembros de la clase Object, es decir, la clase Object define el esta y entorno básicos para todo objeto Java. Esto difiere significativamente de C++, en donde no hay una clase central de la que se derive todo, por lo que no hay una definición de un estado básico en C++.

Las características más importantes que la clase Object cede a sus descendientes son:

- Posibilidad de cooperación consigo o con otro objeto
- Posibilidad de conversión automática a String
- Posibilidad de espera sobre una variable de condición
- Posibilidad de notificación a otros objetos del estado de la variable de condición

La sintaxis general de definición de clase se podría extender tal como se muestra en el siguiente diagrama, que debe tomarse solamente como referencia y no al pie de la letra:

```
NombreDeLaClase {
    // declaración de las variables de instancia
    // declaración de las variables de la clase
    metodoDeInstancia() {
        // variables locales
        // código
        }
    metodoDeLaClase() {
        // variables locales
        // código
        }
    }
```

Tipos de Clases

Hasta ahora sólo se ha utilizado la palabra clave public para calificar el nombre de las clases que hemos visto, pero hay tres modificadores más. Los tipos de clases que podemos definir son:

public

Las clases public son accesibles desde otras clases, bien sea directamente o por herencia, desde clases declaradas fuera del paquete que contiene a esas clases públicas, ya que, por defecto, las clases solamente son accesibles por otras clases declaradas dentro del mismo paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas. La sintaxis es:

```
public class miClase extends SuperClase implements
miInterface,TuInterface {
    // cuerpo de la clase
    }
```

Aquí la palabra clave public se utiliza en un contexto diferente del que se emplea cuando se define internamente la clase, junto con private y protected.

abstract

Una clase abstract tiene al menos un método abstracto. Una clase abstracta no se instancia, sino que se utiliza como clase base para la herencia. Es el equivalente al prototipo de una función en C++.

final

Una clase final se declara como la clase que termina una cadena de herencia, es lo contrario a una clase abstracta. Nadie puede heredar de una clase final. Por ejemplo, la clase Math es una clase final. Aunque es técnicamente posible declarar clases con varias combinaciones de public, abstract y final, la declaración de una clase abstracta y a la vez final no tiene sentido, y el compilador no permitirá que se declare una clase con esos dos modificadores juntos.

synchronizable

Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintos threads; el sistema se encarga de colocar los flags necesarios para evitarlo. Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobreescriban.

Si no se utiliza alguno de los modificadores expuestos, por defecto, Java asume que una clase es:

- No final
- No abstracta
- Subclase de la clase Object
- No implementa interfaz alguno

Variables Miembro

Una clase en Java puede contener variables y métodos. Las variables pueden ser tipos primitivos como int, char, etc. Los métodos son funciones.

Por ejemplo, en el siguiente trozo de código podemos observarlo:

```
public class MiClase {
    int i;

    public MiClase() {
        i = 10;
        }
    public void Suma_a_i( int j ) {
        int suma;
        suma = i + j;
        }
    }
```

La clase MiClase contiene una variable (i) y dos métodos, MiClase() que es el constructor de la clase y Suma_a_i(int j).

La declaración de una **variable miembro** aparece dentro del cuerpo de la clase, pero fuera del cuerpo de cualquier método de esa clase. Si se declara dentro de un métodos, será una **variable local del método** y no una variable miembro de la clase. En el ejemplo anterior, i es una variable miembro de la clase y suma es una variable local del método Suma_ a_i().

El tipo de una variable determina los valores que se le pueden asignar y las operaciones que se pueden realizar con ella.

El nombre de una variable ha de ser un identificador válido en Java. Por convenio, los programadores Java empiezan los nombres de variables con una letra minúscula, pero no es imprescindible. Los nombres de las variables han de ser únicos dentro de la clase y se permite que haya variables y métodos con el mismo nombre.

Java permite la inicialización de variables miembro de tipos primitivos en el momento de la declaración .

La sintaxis completa de la declaración de una variable miembro de una clase en Java sería:

```
[especificador_de_acceso] [static] [final] [transient] [volatile]
    tipo nombreVariable [= valor_inicial];
```

Ambito de una Variable

Los bloques de sentencias compuestas en Java se delimitan con dos llaves. Las variables de Java sólo son válidas desde el punto donde están declaradas hasta el final de la sentencia compuesta que la engloba. Se pueden anidar estas sentencias compuestas, y cada una puede contener su propio conjunto de declaraciones de variables locales. Sin embargo, no se puede declarar una variable con el mismo nombre que una de ámbito exterior.

El siguiente ejemplo intenta declarar dos variables separadas con el mismo nombre. En Java, esto es ilegal, aunque en C se lo tragara.

```
class Ambito {
    int i = 1;      // ámbito exterior
    {               // crea un nuevo ámbito
        int i = 2; // error de compilación
    }
}
```

Variables de Instancia

La declaración de una variable miembro dentro de la definición de una clase sin anteponerle la palabra clave static, hace que sea una **variable de instancia** en todos los objetos de la clase. El significado de variable de instancia sería, más o menos, que cualquier objeto instanciado de esa clase contiene su propia copia de toda variable de instancia. Si se examinara la zona de memoria reservada a cada objeto de la clase, se encontraría la reserva realizada para todas las variables de instancia de la clase. En otras palabras, como un objeto es una instancia de una clase, y como cada objeto tiene su propia copia de un dato miembro particular de la clase, entonces se puede denominar a ese dato miembro como variable de instancia.

En Java, se accede a las variables de instancia asociadas a un objeto determinado utilizando el nombre del objeto, el operador punto (.) y el nombre de la variable:

```
miObjeto.miVariableDeInstancia;
```

Variables Estáticas

La declaración de un dato miembro de una clase usando **static**, crea una **variable de clase** o variable estática de la clase. El significado de variable estática es que todas las instancias de la clase (todos los objetos instanciados de la clase) contienen la mismas variables de clase o estáticas. En otras palabras, en un momento determinado se puede querer crear una clase en la que el valor de una variable de instancia sea el mismo (y de hecho sea la misma variable) para todos los objetos instanciados a partir de esa clase. Es decir, que exista una única copia de la variable de instancia, entonces es cuando debe usarse la palabra clave static.

```
class Documento extends Pagina {
        static int version = 10;
        }
```

El valor de la variable version será el mismo para cualquier objeto instanciado de la clase Documento. Siempre que un objeto instanciado de Documento cambie la variable version, ésta cambiará para todos los objetos.

Si se examinara en este caso la zona de memoria reservada por el sistema para cada objeto, se encontraría con que todos los objetos comparten la misma zona de memoria para cada una de las variables estáticas, por ello se llaman también variables de clase, porque son comunes a la clase, a todos los objetos instanciados de la clase.

Java se puede acceder a las variables de clase utilizando el nombre de la clase y el nombre de la variable, no es necesario instanciar ningún objeto de la clase para acceder a las variables de clase.

En Java, a las variables de clase se accede utilizando el nombre de la clase, el nombre de la variable y el operador punto (.). La siguiente línea de código, ya archivista, se utiliza para acceder a la variable out de la clase System. En el proceso, se accede al método println() de la variable de clase que presenta una cadena en el dispositivo estándar de salida.

```
System.out.println( "Hola, Mundo" );
```

Es importante recordar que todos los objetos de la clase comparten las misma variables de clase, porque si alguno de ellos modifica alguna de esas variables de clase, quedarán modificadas para todos los objetos de la clase. Esto puede utilizarse como una forma de comunicación entre objetos.

Constantes

En Java, se utiliza la palabra clave **final** para indicar que una variable debe comportarse como si fuese constante, significando con esto que no se permite su modificación una vez que haya sido declarada e inicializada.

Como es una constante, se le ha de proporcionar un valor en el momento en que se declare, por ejemplo:

```
class Elipse {
    final float PI = 3.14159;
    . . .
    }
```

Si se intenta modificar el valor de una variable final desde el código de la aplicación, se generará un error de compilación.

Si se usa la palabra clave final con una variable o clase estática, se pueden crear **constantes de clase**, haciendo de esto modo un uso altamente eficiente de la memoria, porque no se necesitarían múltiples copias de las constantes.

La palabra clave final también se puede aplicar a métodos, significando en este caso que los métodos no pueden ser sobreescritos.

Métodos

Los métodos son funciones que pueden ser llamadas dentro de la clase o por otras clases. La implementación de un método consta de dos partes, una *declaración* y un *cuerpo*. La declaración en Java de un método se puede expresar esquemáticamente como:

```
tipoRetorno nombreMetodo( [lista_de_argumentos] )
{ cuerpoMetodo }
```

La sintaxis de la declaración completa de un método es la que se muestra a continuación .

```
especificadorAcceso static abstract final native synchronized
tipoRetorno nombreMetodo( lista_de_argumentos )throws listaEscepciones
```

especificadorAcceso, determina si otros objetos pueden acceder al método y cómo pueden hacerlo.

`static`, indica que los métodos pueden ser accedidos sin necesidad de instanciar un objeto del tipo que determina la clase.

`abstract`, indica que el método no está definido en la clase, sino que se encuentra declarado ahí para ser definido en una subclase (sobreescrito).

`final`, evita que un método pueda ser sobreescrito.

`native`, son métodos escritos es otro lenguaje.

`synchronized`, se usa en el soporte de multithreading.

lista_de_argumentos, es la lista opcional de parámentros que se pueden pasar al método.

`throws` *listaExcepciones*, indica las excepciones que puede generar y manipular el método.

Valor de Retorno de un Método

En Java es imprescindible que a la hora de la declaración de un método, se indique el tipo de dato que ha de devolver. Si no devuelve ningún valor, se indicará el tipo **`void`** como retorno. Todos los tipos primitivos en Java se devuelven por valor y todos los objetos se devuelven por referencia. Para devolver un valor se utiliza la palabra clave **`return`**. La palabra clave `return` va seguida de una expresión que será evaluada para saber el valor de retorno. Esta expresión puede ser compleja o puede ser simplemente el nombre de un objeto, una variable de tipo primitivo o una constante.

El ejemplo java506.java ilustra el retorno por valor y por referencia.

```
// Este ejemplo ilustra el retorno de valores tanto por referencia
```

```
// como por valor

// Un objeto de esta clase sera devuelto por referencia
class miClase {
   int varInstancia = 10;
   }
class java506 {
   // Metodo que devuelve por Valor
   int retornoPorValor() {
       // Devuelve un tipo primitivo por valor
       return( 5 );
       }
   // Metodo que devuelve por Referencia
   miClase retornoPorReferencia() {
       // Devuelve un objeto por referencia
       return( new miClase() );
       }
   public static void main( String args[] ) {
       // Instancia un objeto
     java506 obj = new java506();
     System.out.println( "El Valor devuelto es "+obj.retornoPorValor());
     System.out.println(
     "El Valor de la variable de instancia en el objeto devuelto es "+
     obj.retornoPorReferencia().varInstancia ); // Atencion a los dos
puntos
       }
   }
```

Si un programa Java devuelve una referencia a un objeto y esa referencia no es asignada a ninguna variable, o utilizada en una expresión, el objeto se marca inmediatamente para que el reciclador de memoria en su siguiente ejecución devuelve la memoria ocupada por el objeto al sistema, asumiendo que la dirección no se encuentra ya almacenada en ninguna otra variable.

Nombre del Método

El nombre del método puede ser cualquier identificador legal en Java. Java soporta el concepto de ***sobrecarga de métodos***, es decir, permite que dos métodos compartan el mismo nombre pero con diferente lista de argumentos, de forma que el compilador pueda diferenciar claramente cuando se invoca a uno o a otro, en función de los parámetros que se utilicen en la llamada al método.

El siguiente fragmento de código muestra una clase Java con cuatro métodos sobrecargados, el último no es legal porque tiene el mismo nombre y lista de argumentos que otro previamente declarado:

```
class MiClase {

       . . .

   void miMetodo( int x,int y ) { . . . }

   void miMetodo( int x ) { . . . }

   void miMetodo( int x,float y ) { . . . }

  // void miMetodo( int a,float b ) { . . . } // no válido

   }
```

Todo lenguaje de programación orientado a objetos debe soportar las características de encapsulación, herencia y polimorfismo. La sobrecarga de métodos es considerada por algunos autores como polimorfismo en tiempo de compilación. En Java, los métodos sobrecargados siempre deben devolver el mismo tipo.

Métodos de Instancia

Cuando se incluye un método en una definición de una clase Java sin utilizar la palabra clave `static`, estamos generando un ***método de instancia***. Aunque cada objeto de la clase no contiene su propia copia de un método de instancia (no existen múltiples copias del método en memoria), el resultado final es como si fuese así, como si cada objeto dispusiese de su propia copia del método. Cuando se invoca un método de instancia a través de un objeto determinado, si este método referencia a variables de instancia de la clase, en realidad se están referenciando variables de instancia específicas del objeto específico que se está invocando. La llamada a los métodos de instancia en Java se realiza utilizando el nombre del objeto, el operador punto y el nombre del método.

```
miObjeto.miMetodoDeInstancia();
```

Los métodos de instancia tienen acceso tanto a las variables de instancia como a las variables de clase.

Métodos Estáticos

Cuando una función es incluida en una definición de clase o un método de una clase Java, y se utiliza la palabra `static`, se obtiene un **método estático** o método de clase. Lo más significativo de los métodos de clase es que pueden ser invocados sin necesidad de que haya que instanciar ningún objeto de la clase. En Java se puede invocar un método de clase utilizando el nombre de la clase, el operador punto y el nombre del método.

```
MiClase.miMetodoDeClase();
```

En Java, los métodos de clase operan solamente como variables de clase; no tienen acceso a variables de instancia de la clase, a no ser que se cree un nuevo objeto y se acceda a las variables de instancia a través de ese objeto.

Si se observa el siguiente trozo de código de ejemplo:

```
class Documento extends Pagina {
 static int version = 10;       // Variable de clase
 int numero_de_capitulos;       // Variable de instancia
 static void annade_un_capitulo() {
     numero_de_capitulos++; // esto no funciona
       }
 static void modifica_version( int i ) {
       version++;              // esto si funciona
        }
    }
```

la modificación de la variable `numero_de_capitulos` no funciona porque se está violando una de las reglas de acceso al intentar acceder desde un método estático a una variable no estática.

Todas las clases que se derivan, cuando se declaran estáticas, comparten la misma página de variables; es decir, todos los objetos que se generen comparten la misma zona de memoria. Los métodos estáticos se usan para acceder solamente a variables estáticas.

Paso de parámetros

En Java, todos los métodos deben estar declarados y definidos dentro de la clase, y hay que indicar el tipo y nombre de los argumentos o parámetros que acepta. Los argumentos son como variables locales declaradas en el cuerpo del método que están inicializadas al valor que se pasa como parámetro en la invocación del método. En Java, todos los argumentos de tipos primitivos deben pasarse por valor, mientras que los objetos deben pasarse por referencia. Cuando se pasa un objeto por referencia, se está pasando la dirección de memoria en la que se encuentra almacenado el objeto. Si se modifica una variable que haya sido pasada por valor, no se modificará la variable original que se haya utilizado para invocar al método, mientras que si se modifica una variable pasada por referencia, la variable original del método de llamada se verá afectada de los cambios que se produzcan en el método al que se le ha pasado como argumento.

El ejemplo java515.java se ilustra el paso de parámetros de tipo primitivo y también el paso de objetos, por valor y por referencia, respectivamente.

```
// Esta clase se usa para instanciar un objeto referencia

class MiClase {
      int varInstancia = 100;
         }
           // Clase principal
class java515 {
           // Función para ilustrar el paso de parámetros
      void pasoVariables( int varPrim,MiClase varRef ) {
      System.out.println( "--> Entrada en la funcion pasoVariables" );
      System.out.println( "Valor de la variable primitiva: "+varPrim );
      System.out.println( "Valor contenido en el objeto: "+
varRef.varInstancia );
      System.out.println( "-> Modificamos los valores" );
      varRef.varInstancia = 101;
      varPrim = 201;
      System.out.println( "--> Todavia en la funcion pasoVariables" );
      System.out.println( "Valor de la variable primitiva: "+varPrim );
      System.out.println( "Valor contenido en el objeto: "+
varRef.varInstancia );
        }
      public static void main( String args[] ) {
       // Instanciamos un objeto para acceder a sus métodos
```

```
        java515 aObj = new java515();
        // Instanciamos un objeto normal
        MiClase obj = new MiClase();

             // Instanciamos una variable de tipo primitivo
        int varPrim = 200;
        System.out.println( "> Estamos en main()" );
        System.out.println( "Valor de la variable primitiva: "+varPrim );
        System.out.println( "Valor contenido en el objeto: "+
obj.varInstancia );
        // Llamamos al método del objeto
        aObj.pasoVariables( varPrim,obj );
        System.out.println( "> Volvemos a main()" );
        System.out.println( "Valor de la variable primitiva, todavia : "+
varPrim );
        System.out.println( "Valor contenido ahora en el objeto: "+
obj.varInstancia );
        }
}
```

En Java los métodos tienen acceso directo a las variables miembro de la clase. El nombre de un argumento puede tener el mismo nombre que una variable miembro de la clase. En este caso, la variable local que resulta del argumento del método, *oculta* a la variable miembro de la clase. Cuando se instancia un método se pasa siempre una referencia al propio objeto que ha llamado al método, es la referencia **`this`**.

Constructores

Java soporta la sobrecarga de métodos, es decir, que dos o más métodos puedan tener el mismo nombre, pero distinta lista de argumentos en su invocación. Si se sobrecarga un método, el compilador determinará ya en tiempo de compilación, en base a lista de argumentos con que se llame al método, cual es la versión del método que debe utilizar.

Java soporta la noción de ***constructor***. El constructor es un tipo específico de método que siempre tiene el mismo nombre que la clase y se utiliza para construir objetos de esa clase. No tiene tipo de dato específico de retorno, ni siquiera `void`. Esto se debe a que el tipo específico que debe devolver un constructor de clase es el propio tipo de la clase.

En este caso, pues, no se puede especificar un tipo de retorno, ni se puede colocar ninguna sentencia que devuelva un valor. Los constructores pueden sobrecargarse, y aunque puedan contener código, su función primordial es inicializar el nuevo objeto que se instancia de la clase. En Java, ha de hacerse una llamada al constructor para instanciar un nuevo objeto.

Cuando se declara una clase en Java, se pueden declarar uno o más constructores opcionales que realizan la inicialización cuando se instancia (se crea una ocurrencia) un objeto de dicha clase.

Utilizando el código de la sección anterior, cuando se crea una nueva instancia de **MiClase**, se crean (instancias) todos los métodos y variables, y se llama al constructor de la clase:

```
MiClase mc;
mc = new MiClase();
```

La palabra clave **new** se usa para crear una instancia de la clase. Antes de ser instanciada con `new` no consume memoria, simplemente es una declaración de tipo. Después de ser instanciado un nuevo objeto `mc`, el valor de `i` en el objeto `mc` será igual a 10. Se puede referenciar la variable (de instancia) `i` con el nombre del objeto:

```
mc.i++; // incrementa la instancia de i de mc
```

Al tener `mc` todas las variables y métodos de **MiClase**, se puede usar la primera sintaxis para llamar al método *Suma_a_i()* utilizando el nuevo nombre de clase **mc**:

```
mc.Suma_a_i( 10 );
```

y ahora la variable `mc.i` vale 21.

Luego, en Java, cuando se instancia un objeto, siempre se hace una llamada directa al constructor como argumento del operador `new`. Este operador se encarga de que el sistema proporcione memoria para contener al objeto que se va a crear.

En Java solamente se pueden instanciar en la pila de memoria, es decir, solamente se pueden instanciar utilizando el operador `new` para poder solicitar memoria al sistema en tiempo de ejecución y utilizar el constructor para instanciar el objeto en esa zona de memoria. Si al intentar instanciar un objeto, *la Máquina Virtual Java* (JVM) no puede localizar la memoria que necesita ese objeto, bien inmediatamente o haciendo ejecutarse al reciclador de memoria, el sistema lanzará un *OutOfMemoryError*.

En Java si no se proporciona explícitamente un constructor, el sistema proporciona uno por defecto que inicializará automáticamente todas las variables miembro a cero o su equivalente.

Se puede pensar en el constructor de defecto en Java como un método que tiene el mismo nombre que la clase y una lista de argumentos vacía.

```
MiClase objeto;
```

Si se proporciona uno o más constructores, el constructor de defecto no se proporciona automáticamente y si fuese necesaria su utilización, tendría que proporcionarlo explícitamente el programa.

Las dos sentencias siguientes muestran cómo se utiliza el constructor en Java para declarar, instanciar y, opcionalmente, inicializar un objeto:

```
MiClase miObjeto = new MiClase();
MiClase miObjeto = new MiClase( 1,2,3 );
```

Las dos sentencias devuelven una referencia al nuevo objeto que es almacenada en la variable `miObjeto`. También se puede invocar al constructor sin asignar la referencia a una variable. Esto es útil cuando un método requiere un objeto de un tipo determinado como argumento, ya que se puede incluir una llamada al constructor de este objeto en la llamada al método:

```
miMetodo( new MiConstructor( 1,2,3 ) );
```

Aquí se instancia e inicializa un objeto y se pasa a la función. Para que el programa compile adecuadamente, debe existir una versión de la función que espere recibir un objeto de ese tipo como parámetro.

Cuando un método o una función comienza su ejecución, todos los parámetros se crean como variables locales automáticas. En este caso, el objeto es instanciado en conjunción con la llamada a la función que será utilizada para inicializar esas variables locales cuando comience la ejecución y luego serán guardadas. Como son automáticas, cuando el método concluye su ejecución, será marcado para su destrucción (en Java).

En el siguiente ejemplo, java507.java, se ilustran algunos de los conceptos sobre constructores que se han planteado en esta sección.

```
class MiClase {
   int varInstancia;
     // Este es el constructor parametrizado
   MiClase( int dato ) {
       // rellenamos la variable de instancia con los datos
       // que se pasan al constructor
       varInstancia = dato;
       }
   void verVarInstancia() {
       System.out.println( "El Objeto contiene " + varInstancia );
       }
   }
class java507 {
   public static void main( String args[] ) {
       System.out.println( "Lanzando la aplicacion" );
       // Instanciamos un objeto de este tipo llamando al
       // constructor de defecto
       java507 obj = new java507();
       // Llamamos a la funcion pasandole un constructor
       // parametrizado como parametro
       obj.miFuncion( new MiClase( 100 ) );
       }
   // Esta funcion recibe un objeto y llama a uno de sus metodos
   // para presentar en pantalla el dato que contiene el objeto
   void miFuncion( MiClase objeto){
       objeto.verVarInstancia();
       }
   }
```

Herencia

En casos en que se vea involucrada la *herencia*, los constructores toman un significado especial porque lo normal es que la subclase necesite que se ejecute el constructor de la superclase antes que su propio constructor, para que se inicialicen correctamente aquellas variables que deriven de la superclase. En Java, la sintaxis para conseguir esto es sencilla y consiste en incluir en el cuerpo del constructor de la subclase como primera línea de código la siguiente sentencia:

```
super( parametros_opcionales );
```

Esto hará que se ejecute el constructor de la superclase, utilizando los parámetros que se pasen para la inicialización. En el código del ejemplo siguiente, java508.java se ilustra el uso de esta palabra clase para llamar al constructor de la superclase desde una subclase.

```
class SuperClase {
   int varInstancia;
   // Es necesario proporcionar el constructor por defecto,que
```

```
    // es aquel que no tiene parametros de llamada
    SuperClase(){}
    // Este es el constructor parametrizado de la superclase
    SuperClase( int pDato ) {
        System.out.println(
            "Dentro del constructor de la SuperClase" );
        varInstancia = pDato;
        }
    void verVarInstancia() {
        System.out.println( "El Objeto contiene " + varInstancia );
        }
    }
  class SubClase extends SuperClase {
    // Este es el constructor parametrizado de la subclase
    SubClase( int bDato ) {
        // La siguiente sentencia println no compila, la llamada
        // a super() debe estar al principio de un metodo en caso de
        // que aparezca
        // System.out.println( "En el constructor de la SubClase" );
        // Llamamos al constructor de la superclase
        super( bDato );
        System.out.println(
            "Dentro del constructor de la SubClase" );
        }
    }
  class java508 {
    public static void main( String args[] ) {
        System.out.println( "Lanzando la aplicacion" );
        // Instanciamos un objeto de este tipo llamando al
        // constructor de defecto
        java508 obj = new java508();
        // Llamamos a la funcion pasandole un constructor de la
        // subclase parametrizado como parametro
        obj.miFuncion( new SubClase( 100 ) );
        }
        // Esta funcion recibe un objeto y llama a uno de sus metodos
        // para presentar en pantalla el dato que contiene el objeto,
        // en este caso el metodo es heredado de la SuperClase
        void miFuncion( SubClase objeto ) {
            objeto.verVarInstancia();
        }
    }
```

Si `super` no aparece como primera sentencia del cuerpo de un constructor, el compilador Java inserta una llamada implícita, *super()*, al constructor de la superclase inmediata. Es decir, el constructor por defecto de la superclase es invocado automáticamente cuando se ejecuta el constructor para una nueva subclase, si no se especifica un constructor parametrizado para llamar al constructor de la superclase.

Control de Acceso

El control de acceso también tiene un significado especial cuando se trata de constructores. Aunque en otra sección se trata a fondo el tela del control de acceso en Java, con referencia a los constructores se puede decir que el control de acceso que se indique determina la forma en que otros objetos van a poder instanciar objetos de la clase. En la siguiente descripción, se indica cómo se trata el control de acceso cuando se tienen entre manos a los constructores:

private

Ninguna otra clase puede instanciar objetos de la clase. La clase puede contener métodos públicos, y estos métodos pueden construir un objeto y devolverlo, pero nadie más puede hacerlo.

protected

Solamente las subclases de la clase pueden crear instancias de ella.

public

Cualquier otra clase puede crear instancias de la clase.

package

Nadie desde fuera del paquete puede construir una instancia de la clase. Esto es útil si se quiere tener acceso a las clases del paquete para crear instancias de la clase, pero que nadie más pueda hacerlo, con lo cual se restringe quien puede crear instancias de la clase.

En Java una instancia de una clase, un objeto, contiene todas las variables y métodos de instancia de la clase y de todas sus superclases. Sin embargo, soporta la posibilidad de sobreescribir un método declarado en una superclase, indicando el mismo nombre y misma lista de argumentos.

Como aclaración a terminología que se empleo en este documento, quiero indicar que cuando digo *sobrecargar métodos*, quiero decir que Java requiere que los dos métodos tengan el mismo nombre, devuelvan el mismo tipo, pero tienen una diferente lista de argumentos. Y cuando digo *sobreescribir métodos*, quiero decir que Java requiere que los dos métodos tengan el mismo nombre, mismo tipo de retorno y misma lista de argumentos de llamada.

En Java, si una clase define un método con el mismo nombre, mismo tipo de retorno y misma lista de argumentos que un método de una superclase, el nuevo método sobreescribirá al método de la superclase, utilizándose en todos los objetos que se creen en donde se vea involucrado el tipo de la subclase que sobreescribe el método.

Finalizadores

Java no utiliza destructores ya que tiene una forma de recoger automáticamente todos los objetos que se salen del alcance. No obstante proporciona un método que, cuando se especifique en el código de la clase, el reciclador de memoria (*garbage collector*) llamará:

```
 // Cierra el canal cuando este objeto es reciclado
protected void finalize() {
    close();
    }
```

Cada objeto tiene el método *finalize()*, que es heredado de la clase **Object**. Si se necesitase realizar alguna limpieza asociada con la memoria, se puede sobreescribir el método *finalize()* y colocar en él el código que sea necesario.

La regla de oro a seguir es que no se debe poner ningún código que deba ser ejecutado en el método *finalize()*. Por ejemplo, si se necesita concluir la comunicación con un servidor cuando ya no se va a usar un objeto, no debe ponerse el código de desconexión en el método *finalize()*, porque puede que *nunca* sea llamado. Luego, en Java, es responsabilidad del programador escribir métodos para realizar limpieza que no involucre a la memoria ocupada por el objeto y ejecutarlos en el instante preciso. El método *finalize()* y el reciclador de memoria son útiles para liberar la memoria de la pila y debería restringirse su uso solamente a eso, y no depender de ellos para realizar ningún otro tipo de limpieza.

No obstante, Java dispone de dos métodos para asegurar que los finalizadores se ejecuten. Los dos métodos habilitan la finalización a la salida de la aplicación, haciendo que los finalizadores de todos los objetos que tengan finalizador y que todavía no hayan sido invocados automáticamente, se ejecuten antes de que la Máquina Virtual Java concluya la ejecución de la aplicación. Estos dos métodos son:

runFinalizersOnExit(`boolean` *)*, método estático de **java.lang.Runtime**, y

runFinalizersOnExit(`boolean` *)*, método estático de **java.lang.System**

Una clase también hereda de sus superclase el método *finalize()*, y en caso necesario, debe llamarse una vez que el método *finalize()* de la clase haya realizado las tareas que se le hayan encomendado, de la forma:

```
super.finalize();
```

En la construcción de un objeto, se desplaza uno por el árbol de jerarquía, de herencia, desde la raíz del árbol hacia las ramas, y en la finalización, es al revés, los desplazamientos por la herencia debe ser desde las ramas hacia las superclases hasta llegar a la clase raíz.

La Clase String

Java posee gran capacidad para el manejo de cadenas dentro de sus clases String y StringBuffer. Un objeto String representa una cadena alfanumérica de un valor constante que no puede ser cambiada después de haber sido creada. Un objeto StringBuffer representa una cadena cuyo tamaño puede variar, o puede ser modificada por programa.

Los Strings son objetos constantes y por lo tanto muy baratos para el sistema. La mayoría de las funciones relacionadas con cadenas esperan valores String como argumentos y devuelven valores String.

Hay que tener en cuenta que las funciones estáticas no consumen memoria del objeto, con lo cual es más conveniente usar Character que char . No obstante, char se usa, por ejemplo, para leer ficheros que están escritos desde otro lenguaje.

Existen varios constructores para crear nuevas cadenas:

```
String();
String( String value );
String( char value[] );
String( char value[],int offset,int count );
String( byte bytes[],int offset,int length );
String( byte bytes[],int offset,int length,String enc );
String( byte bytes[] );
String( byte bytes[],String enc );
```

```
String( StringBuffer buffer );
```

Tal como uno puede imaginarse, las cadenas pueden ser muy complejas, existiendo muchos métodos muy útiles para trabajar con ellas y, afortunadamente, la mayoría están codificados en la clase String, por lo que sería conveniente que el programador tuviese una copia de la declaración de la clase String encima de su mesa de trabajo, para determinar el significado de los parámetros en los constructores y métodos de la clase String. Es más, esta necesidad puede extenderse a todas las demás clases, pero claro, teniendo en cuenta el espacio disponible sobre la mesa de trabajo.

También merece comentario el parámetro enc que se utiliza en algunos constructores de la clase y que permite encriptar la cadena que se le pasa como parámetro en función de los caracteres que se indiquen en el parámetro. En caso de usar alguno de estos constructores, hay que tener en cuenta que la cadena que se devuelve puede que no tenga la misma longitud que la que se le pasa para codificar, por lo que deben utilizarse con cuidado estos constructores.

Hay que resaltar el hecho de que mientras que el contenido de un objeto String no puede modificarse, una referencia a un objeto String puede hacerse que apunte a otro objeto String, de tal forma que parece que el primer objeto ha sido modificado. Esta característica es la que ilustra el ejemplo .

Es importante resaltar que la siguiente sentencia no modifica el objeto original referenciado por la variable str1:

```
thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
```

sino que, esta sentencia crea un nuevo objeto que es la concatenación de los objetos existentes y hace que la variable de referencia str1 apunte al nuevo objeto, en lugar de apuntar al objeto original. Ese objeto original queda marcado para que el reciclador de memoria en su siguiente pasada, devuelva la memoria que ocupaba al sistema.

Una reflexión especial merecen los arrays de Strings. La siguiente sentencia declara e instancia un array de referencias a cinco objetos de tipo String:

```
String miArray = new String[5];
```

Este array no contiene los datos de las cadenas. Solamente reserva una zona de memoria para almacenar las cinco referencias a cadenas. No se guarda memoria para almacenar los caracteres que van a componer esas cinco cadenas, por lo que hay que reservar expresamente esa memoria a la hora de almacenar los datos concretos de las cadenas, tal como se hace en la siguiente línea de código:

```
miArray[0] = new String( "Esta es la primera cadena" );
miArray[1] = new String( "Esta es la segunda cadena" );
```

Esto resultará conocido a los programadores C/C++, ya que sería el método habitual de reservar memoria para un array de punteros de tipo char y luego hacer que esos punteros apunten a cadenas localizadas en algún lugar de la memoria.

Funciones Básicas

La primera devuelve la longitud de la cadena y la segunda devuelve el carácter que se encuentra en la posición que se indica en indice :

```
int length();
char charAt( int indice );
```

Funciones de Comparación de Strings

```
boolean equals( Object obj );
boolean equalsIgnoreCase( String str2 );
```

Lo mismo que equals() pero no tiene en cuenta mayúsculas o minúsculas.

```
int compareTo( String str2 );
```

Devuelve un entero menor que cero si la cadena es léxicamente menor que str2 . Devuelve cero si las dos cadenas son léxicamente iguales y un entero mayor que cero si la cadena es léxicamente mayor que str2.

Funciones de Comparación de Subcadenas

```
boolean regionMatches( int offset,String other,int ooffset,
    int len );
boolean regionMatches( boolean ignoreCase,int offset,String other,
    int ooffset,int len );
```

Comprueba si una región de esta cadena es igual a una región de otra cadena.

```
boolean startsWith( String prefix );
boolean startsWith( String prefix,int toffset );
boolean endsWith( String suffix );
```

Devuelve si esta cadena comienza o termina con un cierto prefijo o sufijo comenzando en un determinado desplazamiento.

```
int indexOf( int ch );
int indexOf( int ch,int fromIndex );
int lastIndexOf( int ch );
int lastIndexOf( int ch,int fromIndex );
int indexOf( String str );
int indexOf( String str,int fromIndex );
int lastIndexOf( String str );
int lastIndexOf( String str,int fromIndex );
```

Devuelve el primer/último índice de un carácter/cadena empezando la búsqueda a partir de un determinado desplazamiento.

```
String substring( int beginIndex );
String substring( int beginIndex,int endIndex );
String concat( String str );
String replace( char oldChar,char newChar );
String toLowerCase();
String toUpperCase();
String trim();
```

Ajusta los espacios en blanco al comienzo y al final de la cadena.

```
void getChars( int srcBegin,int srcEnd,char dst[],int dstBegin );
void getBytes( int srcBegin,int srcEnd,byte dst[],int dstBegin );
String toString();
char toCharArray();
int hashCode();
```

Con frecuencia se necesita convertir un objeto cualquiera a un objeto String, porque sea necesario pasarlo a un método que solamente acepte valores String, o por cualquier otra razón. Todas las clases heredan el método toString() de la clase Object y muchas de ellas lo sobreesciben para proporcionar una implementación que tenga sentido para esas clases. Además, puede haber ocasiones en que sea necesario sobreescribir el método toString() para clases propias y proporcionarles un método de conversión adecuado.

En el ejemplo , se sobreescribe el método toString() para la nueva clase que se crea en el código.

```
class java805 {
    // Se definen las variables de instancia para la clase
    String uno;
    String dos;
    String tres;

    // Constructor de la clase
    java805( String a,String b,String c ) {
        uno = a;
        dos = b;
        tres = c;
        }

    public static void main( String args[] ) {
        // Se instancia un objeto de la clase
        java805 obj = new java805( "Tutorial","de","Java" );

        // Se presenta el objeto utilizando el metodo sobreescrito
        System.out.println( obj.toString() );
        }

    // Sobreescritura del metodo toString() de la clase Object
    public String toString() {
        // Convierte un objeto a cadena y lo devuelve
        return( uno+" "+dos+" "+tres );
        }
    }
```

Funciones de Conversión

La clase String posee numerosas funciones para transformar valores de otros tipos de datos a su representación como cadena. Todas estas funciones tienen el nombre de valueOf, estando el método sobrecargado para todos los tipos de datos básicos.

A continuación se muestra un ejemplo de su utilización:

```
String Uno = new String( "Hola Mundo" );
float f = 3.141592;

String PI = Uno.valueOf( f );
String PI = String.valueOf( f );    // Mucho más correcto
String valueOf( boolean b );
String valueOf( char c );
String valueOf( int i );
String valueOf( long l );
String valueOf( float f );
String valueOf( double d );
String valueOf( Object obj );
String valueOf( char data[] );
String valueOf( char data[],int offset,int count );
```

Usa arrays de caracteres para la cadena.

```
String copyValueOf( char data[] );
String copyValueOf( char data[],int offset,int count );
```

Crea un nuevo array equivalente para la cadena.

Ahora bien, la conversión contraria de valor numérico a cadena no tiene métodos en Java, a semejanza del atoi() o atof() de C++. En el ejemplo , se muestra una forma de realizar esta conversión, aunque puede haber otras y seguramente mejores.

```
class java806 {
    public static void main( String args[] ) {
        int numero = new Integer( "1234" ).intValue();
```

```
        System.out.println( "El valor entero de la variable " +
            "numero es " + numero );
        }
    }
```

Se crea un objeto Integer temporal que contiene el valor especificado para el String. El método intValue() de la clase Integer es el que se utiliza para extraer el valor del entero de ese objeto y asignarlo a una variable de tipo int . Esto puede utilizarse como patrón para implementar métodos de conversión atoi() propios.

this

Al acceder a variables de instancia de una clase, la palabra clave this hace referencia a los miembros de la propia clase en el objeto actual; es decir, this se refiere al objeto actual sobre el que está actuando un método determinado y se utiliza siempre que se quiera hace referencia al objetoactual de la clase. Volviendo al ejemplo de MiClase, se puede añadir otro constructor de la forma siguiente:

```
public class MiClase {
    int i;
    public MiClase() {
        i = 10;
        }
    // Este constructor establece el valor de i
    public MiClase( int valor ) {
        this.i = valor; // i = valor
        }
      // Este constructor también establece el valor de i
    public MiClase( int i ) {
        this.i = i;
        }
    public void Suma_a_i( int j ) {
        i = i + j;
        }
    }
```

Aquí this.i se refiere al entero i en la clase MiClase, que corresponde al objeto actual. La utilización de this en el tercer constructor de la clase, permite referirse directamente al objeto en sí, en lugar de permitir que el ámbito actual defina la resolución de variables, al utilizar i como parámetro formal y después this para acceder a la variable de instancia del objeto actual.

La utilización de this en dicho contexto puede ser confusa en ocasiones, y algunos programadores procuran no utilizar variables locales y nombres de parámetros formales que ocultan variables de instancia. Una filosofía diferente dice que en los métodos de inicialización y constructores, es bueno seguir el criterio de utilizar los mismos nombres por claridad, y utilizar this para no ocultar las variables de instancia. Lo cierto es que es más una cuestión de gusto personal que otra cosa el hacerlo de una forma u otra.

La siguiente aplicación de ejemplo, , utiliza la referencia this al objeto para acceder a una variable de instancia oculta para el método que es llamado.

```
class java509 {
    // Variable de instancia
    int miVariable;

    // Constructor de la clase
    public java509() {
```

```
        miVariable = 5;
        }

    // Metodo con argumentos
    void miMetodo(int miVariable) {
        System.out.println( "La variable Local miVariable contiene "
            + miVariable );
        System.out.println(
            "La variable de Instancia miVariable contiene "
            + this.miVariable );
        }

    public static void main( String args[] ) {
        // Instanciamos un objeto del tipo de la clase
        java509 obj = new java509();
        // que utilizamos para llamar a su unico metodo
        obj.miMetodo( 10 );
        }
    }
```

super

Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se puede hacer referencia al método padre con la palabra clave super:

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        super.Suma_a_i( j );
    }
}
```

En el siguiente código, el constructor establecerá el valor de i a 10, después lo cambiará a 15 y finalmente el método Suma_a_i() de la clase padre MiClase lo dejará en 25:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

Si un método sobreescribe un método de su superclase, se puede utilizar la palabra clave super para eludir la versión sobreescrita de la clase e invocar a la versión original del método en la supreclase. Del mismo modo, se puede utilizar super para acceder a variables miembro de la superclase.

En el ejemplo , la aplicación utiliza super para referirse a una variable local en un método y a una variable de la superclase que tiene el mismo nombre. El programa también utiliza super para invocar al constructor de la superclase desde en constructor de la subclase.

Herencia

La herencia es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase Ave, se puede crear la subclase Pato, que es una especialización de Ave.

```
class Pato extends Ave {
    int numero_de_patas;
    }
```

La palabra clave **extends** se usa para generar una subclase (especialización) de un objeto. Un Pato es una subclase de Ave. Cualquier cosa que contenga la definición de Ave será copiada a la clase Pato, además, en Pato se pueden definir sus propios métodos y variables de instancia. Se dice que Pato deriva o hereda de Ave.

Además, se pueden sustituir los métodos proporcionados por la clase base.

Utilizando nuestro anterior ejemplo de MiClase, aquí hay un ejemplo de una clase derivada sustituyendo a la función Suma_a_i():

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        }
    }
```

Ahora cuando se crea una instancia de MiNuevaClase, el valor de i también se inicializa a 10, pero la llamada al método Suma_a_i() produce un resultado diferente:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

Java se diseñó con la idea de que fuera un lenguaje sencillo y, por tanto, se le denegó la capacidad de la herencia múltiple, tal como es conocida por los programadores C++.

Subclases

Como ya se ha indicado en múltiples ocasiones en esta sección, cuando se puede crear nuevas clases por herencia de clases ya existentes, las nuevas clases se llaman subclases, mientras que las clases de donde hereda se llaman superclases.

Cualquier objeto de la subclase contiene todas las variables y todos los métodos de la superclase y sus antecesores.

Todas las clases en Java derivan de alguna clase anterior. La clase raíz del árbol de la jerarquía de clases de Java es la clase Object, definida en el paquete java.lang. Cada vez que se desciende en el árbol de jerarquía, las clases van siendo más especializadas.

Cuando se desee que nadie pueda derivar de una clase, se indica que es final; y lo mismo con los métodos, si no se desea que se puedan sobreescribir, se les antepone la palabra clave final.

Lo contrario de final es abstract. Una clase marcada como abstracta, únicamente está diseñada para crear subclases a partir de ella, no siendo posible instanciar ningún objeto a partir de una clase abstracta.

La clase OBJECT

La clase **Object**, como ya se ha indicado anteriormente, es la clase raíz de todo el árbol de la jerarquía de clases Java, y proporciona un cierto número de métodos de utilidad general que pueden utilizar todos los objetos. La lista completa se puede ver en la documentación del API de Java, aquí solamente se tratarán algunos de ellos; por ejemplo, **Object** proporciona:

- Un método por el que un objeto se puede comparar con otro objeto
- Un método para convertir un objeto a una cadena
- Un método para esperar a que ocurra una determinada condición
- Un método para notificar a otros objetos que una condición ha cambiado
- Un método para devolver la clase de un objeto

El método *equals()*

```
public boolean equals( Object obj );
```

Todas las clases que se definan en Java heredarán el método *equals()*, que se puede utilizar para comparar dos objetos. Esta comparación no es la misma que proporciona el operador `==`, que solamente compara si dos referencias a objetos apuntan al mismo objeto.

El método ***equals()*** se utiliza para saber si dos objetos separados son del mismo tipo y contienen los mismos datos. El método devuelve `true` si los objetos son iguales y `false` en caso contrario.

El sistema ya sabe de antemano cómo aplicar el método a todas las clases estándar y a todos los objetos de los que el compilador tiene conocimiento. Por ejemplo, se puede usar directamente para saber si dos objetos `String` son iguales.

Las subclases pueden sobreescribir el método *equals()* para realizar la adecuada comparación entre dos objetos de un tipo que haya sido definido por el programador.

En la aplicación de ejemplo siguiente, java511.java se sobreescribe el método para comparar dos objetos de la nueva clase que crea la aplicación.

Hay que observar que en la lista de argumentos del método *equals()* hay que pasarle un argumento de tipo `Object`. Si se define un método con un argumento de tipo diferente, se estará sobrecargando el método, no sobreescribiéndolo.

En el ejemplo, una vez que se ejecuta, es necesario hacer un moldeo del argumento al tipo de la clase antes de intentar realizar la comparación. Se utiliza el operador **`instanceof`** para confirmar que el objeto es del tipo correcto. Uno de los objetos proporcionados para comprobar su equivalencia es de tipo erróneo (`String`) y el método sobreescrito *equals()* indicará que no es un objeto equivalente.

```
class java511 {
    int miDato;

    // Constructor parametrizado
    java511( int dato ) {
        miDato = dato;
        }

    public static void main(String args[] ) {
        // Se instancian los objetos que se van a testear
        java511 obj1 = new java511( 2 );
        java511 obj2 = new java511( 2 );
        java511 obj3 = new java511( 3 );
        String obj4 = "Un objeto String";

        // Se realizan las comprobaciones y se presenta por pantalla
        // el resultado de cada una de ellas
```

```
        System.out.println( "obj1 equals obj1: " +
            obj1.equals( obj1 ) );
        System.out.println( "obj1 equals obj2: " +
            obj1.equals( obj2 ) );
        System.out.println( "obj1 equals obj3: " +
            obj1.equals( obj3 ) );
        System.out.println( "obj1 equals obj4: " +
            obj1.equals( obj4 ) );
        }

    // Se sobreescribe el metodo equals()
    public boolean equals( Object arg ) {
        // Se comprueba que el argumento es del tipo adecuado y
        // que no es nulo. Si lo anterior se cumple se realiza
        // la comprobacion de equivalencia de los datos.
        // Observese que se ha empleado el operador instanceof
        if( (arg != null) && (arg instanceof java511) ) {
            // Hacemos un moldeado del Object general a tipo java511
            java511 temp = (java511)arg;

            // Se realiza la comparacion y se devuelve el resultado
            return( this.miDato == temp.miDato );
            }
        else {
            // No es del tipo esperado
            return( false );
            }
        }
    }
```

El método *getClass()*

```
public final native Class getClass();
```

En Java existe la clase **Class**, que se define de la forma (la declaración no está completa, consultar el API de Java para ello):

```
public final class java.lang.Class extends java.lang.Object {
   // Métodos
   public static Class forName(String className);
   public ClassLoader getClassLoader();
   public Class[] getClasses();
   public Class getComponentType();
   public Constructor getConstructor(Class[] parameterTypes);
   public Constructor[] getConstructors();
   public Class[] getDeclaredClasses();
   public Constructor[] getDeclaredConstructors();
   public Field getDeclaredField(String name);
   public Field[] getDeclaredFields();
   public Method getDeclaredMethod(String name, Class[] parTypes);
   public Method[] getDeclaredMethods();
   public Class getDeclaringClass();
   public Field getField(String name);
   public Field[] getFields();
   public Class[] getInterfaces();
   public Method getMethod(String name, Class[] parameterTypes);
   public Method[] getMethods();
   public int getModifiers();
   public String getName();
   public URL getResource(String name);
   public InputStream getResourceAsStream(String name);
   public Object[] getSigners();
   public Class getSuperclass();
```

```
public boolean isArray();
public boolean isAssignableFrom(Class cls);
public boolean isInstance(Object obj);
public boolean isInterface();
public boolean isPrimitive();
public Object newInstance();
public String toString(); // Overrides Object.toString()
}
```

Instancias de la clase **Class** representan las clases e interfaces que está ejecutando la aplicación Java. No hay un constructor para la clase **Class**, sus objetos son construidos automáticamente por la *Máquina Virtual Java* (JVM) cuando las clases son cargadas, o por llamadas al método *defineClass()* del cargados de clases.

Es una clase importante porque se le pueden realizar peticiones de información sobre objetos, como cuál es su nombre o cómo se llama su superclase.

El método *getClass()* de la clase **Object** se puede utilizar para determinar la clase de un objeto. Es decir, devuelve un objeto de tipo `Class`, que contiene información importante sobre el objeto que crea la clase. Una vez determinada la clase del objeto, se pueden utilizar los métodos de la clase **Class** para obtener información acerca del objeto.

Además, habiendo determinado la clase del objeto, el método ***newInstance()*** de la clase **Class** puede invocarse para instanciar otro objeto del mismo tipo. El resultado es que el operador `new` será utilizado con un constructor de una clase conocida.

Hay que hacer notar que la última afirmación del párrafo anterior es una situación que el compilador no conoce en tiempo de compilación, es decir, no sabe el tipo del objeto que va a ser instanciado. Por lo tanto, si se necesita referenciar al nuevo objeto, es necesario declarar la variable de referencia del tipo genérico `Object`, aunque el objeto actual tomará todos los atributos de la subclase actual por la que será instanciado.

Hay que hacer notar que el método *getClass()* es un método final y no puede ser sobreescrito. Devuelve un objeto de tipo `Class` que permite el uso de los métodos definidos en la clase **Class** sobre ese objeto.

El siguiente programa, java512.java , ilustra alguna de estas características. Primero, instancia un objeto, mira la clase de ese objeto y utiliza alguno de los métodos de la clase **Class** para obtener y presentar información acerca del objeto. Luego, pregunta al usuario si quiere instanciar un nuevo objeto, instanciando un objeto de tipo String en un caso o, en el otro caso, se aplica el método *getClass()* a un objeto existente y utilizando el método *newInstance()* se instancia un nuevo objeto del mismo tipo.

```
import java.io.*;

class java512 {

   public static void main( String args[] ) {
       java512 obj1 = new java512();
       // Se usa el metodo getClass() de la clase Object y dos
       // metodos de la clase Class para obtener y presentar
       // informacion acerca del objeto
       System.out.println( "Nombre de la clase de obj1: "
           + obj1.getClass().getName() );
       System.out.println( "Nombre de la superclase de obj1: "
           + obj1.getClass().getSuperclass() );
```

```
        // Ahora se instancia otro objeto basandose en la entrada
        // del usuario, de forma que el compilador no tiene forma
        // de conocer el tipo del objeto en tiempo de compilacion
        // Se declara una referencia a un objeto generico
        Object obj2 = null;

        // Se pide la entrada al usuario
        System.out.println( "Introduce un 0 o un 1" );
        try {
            // Captura la entrada del usuario
            int dato = System.in.read();
            // Si se ha introducido un 0
            if( (char)dato == '0' )
                // Se crea un objeto String
                obj2 = "Esto es un objeto String";
            // Sino, se crea un nuevo objeto del mismo tipo
            // que el anterior
            else
                obj2 = obj1.getClass().newInstance();
        } catch( Exception e ) {
            System.out.println( "Excepcion " + e );
            }

        // Ahora se indican la clase y superclase del nuevo objeto
        System.out.println( "Nombre de la clase de obj2: "
            + obj2.getClass().getName() );
        System.out.println( "Nombre de la superclase de obj2: "
            + obj2.getClass().getSuperclass());
        }
    }
```

El método *toString()*

```
public String toString();
```

La clase **Object** dispone de este método que puede usarse para convertir todos los objetos conocidos por el compilador a algún tipo de representación de cadena, que dependerá del objeto.

Por ejemplo, el método *toString()* extrae el entero contenido en un objeto `Integer`. De forma similar, si se aplica el método *toString()* al objeto `Thread`, se puede obtener información importante acerca de los threads y presentarla como cadena.

Este método también se puede sobreescribir, o redefinir, para convertir los objetos definidos por el programador a cadenas. El programa siguiente, java513.java , redefine el método *toString()* de una clase recién definida para que pueda utilizarse en la conversión de objetos de esta clase a cadenas.

```
class java513 {
   // Se definen las variables de instancia para la clase
   String uno;
   String dos;
   String tres;
   // Constructor de la clase
   java513( String a,String b,String c ) {
       uno = a;
       dos = b;
       tres = c;
       }

   public static void main( String args[] ) {
       // Se instancia un objeto de la clase
       java513 obj = new java513( "Tutorial","de","Java" );
```

```
        // Se presenta el objeto utilizando el metodo sobreescrito
        System.out.println( obj.toString() );
        }

    // Sobreescritura del metodo toString() de la clase Object
    public String toString() {
        // Convierte un objeto a cadena y lo devuelve
        return( uno+" "+dos+" "+tres );
        }
    }
```

Otros Métodos

Hay otros métodos útiles en la clase **Object** que son, o serán, discutidos en otras secciones de este documento. Por ejemplo, el método

```
protected void finalize();
```

que se tratará en el apartado de *finalizadores*. O también, los métodos que se utilizan en la programación de threads para hacer que varios threads se sincronicen, como son

```
    public final void wait();
    public final native void wait( long timeout );
    public final native void notify();
    public final native void notifyAll();
```

que se tratarán cuando se hable del *multithreading* en Java.

Interfaces

Los métodos abstractos son útiles cuando se quiere que cada implementación de la clase parezca y funcione igual, pero necesita que se cree una nueva clase para utilizar los métodos abstractos. Los interfaces proporcionan un mecanismo para abstraer los métodos a un nivel superior, lo que permite simular la herencia múltiple de otros lenguajes. Un interfaz sublima el concepto de clase abstracta hasta su grado más alto.

Un interfaz podrá verse simplemente como una forma, es como un molde, solamente permite declarar nombres de métodos, listas de argumentos, tipos de retorno y adicionalmente miembros datos (los cuales podrán ser únicamente tipos básicos y serán tomados como constantes en tiempo de compilación, es decir, `static` y `final`).

Un interfaz contiene una colección de métodos que se implementan en otro lugar. Los métodos de una clase son `public`, `static` y `final`. La principal diferencia entre *interface* y *abstract* es que un interfaz proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia. Por ejemplo

```
public interface VideoClip {
    // comienza la reproduccion del video
    void play();
    // reproduce el clip en un bucle
    void bucle();
    // detiene la reproduccion
    void stop();
    }
```

Las clases que quieran utilizar el interfaz VideoClip utilizarán la palabra **implements** y proporcionarán el código necesario para implementar los métodos que se han definido para el interfaz:

```
class MiClase implements VideoClip {
    void play() {
          <código>
        }
    void bucle() {
          <código>
        }
    void stop() {
          <código>
        }
```

Al utilizar implements para el interface es como si se hiciese una acción de copiar-y-pegar del código del interface, con lo cual no se hereda nada, solamente se pueden usar los métodos.

La ventaja principal del uso de interfaces es que una clase interface puede ser implementada por cualquier número de clases, permitiendo a cada clase compartir el interfaz de programación sin tener que ser consciente de la implementación que hagan las otras clases que implementen el interface.

```
class MiOtraClase implements VideoClip {
    void play() {
          <código nuevo>
        }
    void bucle() {
          <código nuevo>
        }
    void stop() {
          <código nuevo>
        }
```

Es decir, el aspecto más importante del uso de interfaces es que múltiples objetos de clases diferentes pueden ser tratados como si fuesen de un mismo tipo común, donde este tipo viene indicado por el nombre del interfaz.

Aunque se puede considerar el nombre del interfaz como un tipo de prototipo de referencia a objetos, no se pueden instanciar objetos en sí del tipo interfaz. La definición de un interfaz no tiene constructor, por lo que no es posible invocar el operador new sobre un tipo interfaz.

Un interfaz puede heredar de varios interfaces sin ningún problema. Sin embargo, una clase solamente puede heredar de una clase base, pero puede implementar varios interfaces. También, el JDK ofrece la posibilidad de definir un interfaz vacío, como es el caso de Serialize, que permite serializar un objeto. Un interfaz vacío se puede utilizar como un flag, un marcador para marcar a una clase con una propiedad determinada.

La aplicación , ilustra algunos de los conceptos referentes a los interfaces. Se definen dos interfaces, en uno de ellos se definen dos constantes y en el otro se declara un método put() y un método get(). Las constantes y los métodos se podrían haber colocado en la misma definición del interfaz, pero se han separado para mostrar que una clase simple puede implementar dos o más interfaces utilizando el separador coma (,) en la lista de interfaces.

También se definen dos clases, implementando cada una de ellas los dos interfaces. Esto significa que cada clase define el método put() y el método get(), declarados en un interfaz y hace uso de las constantes definidas en el otro interfaz. Estas clase se encuentran en ficheros separados por exigencias del compilador, los ficheros son y , y el contenido de ambos ficheros es el que se muestra a continuación:

```
public interface Constantes {
    public final double pi = 6.14;
    public final int constanteInt = 125;
    }

public interface MiInterfaz {
    void put( int dato );
    int get();
    }
```

Es importante observar que en la definición de los dos métodos del interfaz, cada clase los define de la forma más adecuada para esa clase, sin tener en cuenta cómo estará definidos en las otras clases.

Una de las clases también define el método show(), que no está declarado en el interfaz. Este método se utiliza para demostrar que un método que no está declarado en el interfaz no puede ser accedido utilizando una variable referencia de tipo interfaz.

El método main() en la clase principal ejecuta una serie de instanciaciones, invocaciones de métodos y asignaciones destinadas a mostrar las características de los interfaces descritos anteriormente. Si se ejecuta la aplicación, las sentencias que se van imprimiendo en pantalla son autoexplicactivas de lo que está sucediendo en el corazón de la aplicación.

Los interfaces son útiles para recoger las similitudes entre clase no relacionadas, forzando una relación entre ellas. También para declarar métodos que forzosamente una o más clases han de implementar. Y también, para tener acceso a un objeto, para permitir el uso de un objeto sin revelar su clase, son los llamados objetos anónimos, que son muy útiles cuando se vende un paquete de clases a otros desarrolladores.

Java no permite herencia múltiple. Hay autores que dicen que el interfaz de Java es un sustituto de la herencia múltiple y hay otros que están en desacuerdo con eso. Lo cierto es que sí parece una alternativa y los interfaces resuelven algunos de los problemas de la herencia múltiple, por ejemplo:

- No se pueden heredar variables desde un interfaz
- No se pueden heredar implementaciones de métodos desde un interfaz
- La jerarquía de un interfaz es independiente de la jerarquía de clases que implementen el mismo interfaz

y esto no es cierto en la herencia múltiple, tal como se ve desde C++.

Definición

La definición de un interfaz es semejante a la de una clase. La definición de interfaz tiene dos componentes, declaración y cuerpo. En forma esquemática sería:

```
DeclaracionInterfaz {
    // cuerpoInterfaz
    }
```

Declaración

La mínima declaración consiste en la palabra clave interface y el nombre del interfaz. Por convenio, los nombres de interfaces comienzan con una letra mayúscula, como los nombres de las clases, pero no es obligatorio.

La declaración de interfaz puede tener dos componentes adicionales, el especificador de acceso public y la lista de superinterfaces.

Un interfaz puede extender otros interfaces. Sin embargo, mientras que una clase solamente puede extender otra clase, un interfaz puede extender cualquier número de interfaces. En el ejemplo se muestra la definición completa de un interfaz, declaración y cuerpo.

```
public interface MiInterfaz extends InterfazA,InterfazB {
    public final double PI = 3.14159;
    public final int entero = 125;
    void put( int dato );
    int get();
    }
```

El especificador de acceso public indica que el interfaz puede ser utilizado por cualquier clase de cualquier paquete. Si se omite, el interfaz solamente será accesible a aquellas clases que estén definidas en el mismo paquete.

La cláusula extends es similar a la de la declaración de clase, aunque un interfaz puede extender múltiples interfaces. Un interfaz no puede extender clases.

La lista de superinterfaces es un lista separada por comas de todas las interfaces que la nueva interfaz va a extender. Un interfaz hereda todas las constantes y métodos de sus superinterfaces, excepto si el interfaz oculta una constante con otra del mismo nombre, o si redeclara un método con una nueva declaración de ese método.

El cuerpo del interfaz contiene las declaraciones de los métodos, que terminan en un punto y coma y no contienen código alguno en su cuerpo. Todos los métodos declarados en un interfaz son implícitamente, public y abstract, y no se permite el uso de transient, volatile, private, protected o syncronized en la declaración de miembros en un interfaz. En el cuerpo del interfaz se pueden definir constantes, que serán implícitamente public, static y final.

Implementación

Un interfaz se utiliza definiendo una clase que implemente el interfaz a través de su nombre. Cuando una clase implementa un interfaz, debe proporcionar la definición completa de todos los métodos declarados en el interfaz y, también, la de todos los métodos declarados en todos los superinterfaces de ese interfaz.

Una clase puede implementar más de un interfaz, incluyendo varios nombre de interfaces separados por comas. En este caso, la clase debe proporcionar la definición completa de todos los métodos declarados en todos los interfaces de la lista y de todos los superinterfaces de esos interfaces.

En ejemplo, se puede observar una clase que implementa dos interfaces, Constantes y MiInterfaz.

```
class ClaseA implements Constantes,MiInterfaz {
    double dDato;
    // Define las versiones de put() y get() que utiliza la ClaseA
    public void put( int dato ) {
        // Se usa "pi" del interfaz Constantes
        dDato = (double)dato * pi;
        System.out.println(
            "En put() de ClaseA, usando pi del interfaz " +
            "Constantes: " + dDato );
        }
    public int get() {
        return( (int)dDato );
        }
    // Se define un metodo show() para la ClaseA que no esta
    // declarado en el interfaz MiInterfaz
    void show() {
        System.out.println(
            "En show() de ClaseA, dDato = " + dDato );
        }
    }
```

Como se puede observar, esta clase proporciona la definición completa de los métodos put() y get() del interfaz MiInterfaz, a la vez que utiliza las constantes definidas en el interfaz Constantes. Además, la clase proporciona la definición del método show() que no está declarado en ninguno de los interfaces, sino que es propio de la clase.

La definición de un interfaz es una definición de un nuevo tipo de datos. Se puede utilizar el nombre del interfaz en cualquier lugar en que se pueda utilizar un nombre de tipo de dato. Sin embargo, no se pueden instanciar objetos del tipo interfaz, porque un interfaz no tiene constructor. En esta sección, hay numerosos ejemplos de uso del nombre de un interfaz como tipo.

Herencia "Multiple"

Un interfaz no es solamente una forma más pura de denominar a una clase abstracta, su propósito es mucho más amplio que eso. Como un interfaz no tiene ninguna implementación, es decir, no hay ningun tipo de almacenamiento asociado al interfaz, no hay nada que impida la combinación de varios interfaces. Esto tiene mucho valor porque hay veces en que es necesario que un objeto X sea a y b y c. En Java, se puede hacer , pero solamente una de las clases puede tener implementación, con lo cual los problemas que se presentan en C++ no suceden con Java cuando se combinan multiples interfaces.

En una clase derivada, el programador no está forzado a tener una clase base sea esta abstracta o concreta, es decir, una sin métodos abstractos. Si se hereda desde una clase no interfaz, solamente se puede heredar de una. El resto de los elementos base deben ser interfaces. Todos los nombres de interfaces se colocan después de la palabra clave implements y separados por comas. Se pueden tener tantos interfaces como se quiera y cada uno de ellos será un tipo independiente. El ejemplo siguiente, , muestra una clase concreta combinada con varios interfaces para producir una nueva clase

```
import java.util.*;

interface Luchar {
    void luchar();
    }
```

```
interface Nadar {
    void nadar();
    }
interface Volar {
    void volar();
    }
class Accion {
    public void luchar() {}
    }

class Heroe extends Accion implements Luchar,Nadar,Volar {
    public void nadar() {}
    public void volar() {}
    }

public class java516 {
    static void a( Luchar x ) {
        x.luchar();
        }
    static void b( Nadar x ) {
        x.nadar();
        }
    static void c( Volar x ) {
        x.volar();
        }
    static void d( Accion x ) {
        x.luchar();
        }

    public static void main( String args[] ) {
        Heroe i = new Heroe();
        a( i ); // Trata al Heroe como Luchar
        b( i ); // Trata al Heroe como Nadar
        c( i ); // Trata al Heroe como Volar
        d( i ); // Trata al Heroe como Accion
        }
    }
```

Se puede observar que Heroe combina la clase concreta Accion con los interfaces Luchar, Nadar y Volar. Cuando se combina la clase concreta con interfaces de este modo, la clase debe ir la primera y luego los interfaces; en caso contrario, el compilador dará un error.

La autentificación, signature, para luchar() es la misma en el interfaz Luchar y en la clase Accion, y luchar() no está proporcionado con la definicion de Heroe. Si se quiere crear un objeto del nuevo tipo, la clase debe tener todas las definiciones que necesita, y aunque Heroe no proporciona una definicion explicita para luchar(), la definición es proporcionada automáticamente por Accion y así es posible crear objetos de tipo Heroe.

En la clase java516, hay cuatro métodos que toman como argumentos los distintos interfaces y la clase concreta. Cuando un objeto Heroe es creado, puede ser pasado a cualquierea de estos métodos, lo que significa que se está realizando un upcasting al interfaz de que se trate. Debido a la forma en que han sido diseñados los interfaces en Java, esto funciona perfectamente sin ninguna dificultad ni esfuerzo extra por parte del programador.

La razón principal de la existencia de los interfaces en el ejemplo anterior es el proder realizar un upcasting a más de un tipo base. Sin embargo, hay una segunda razón que es la misma por la que se usan clases abstractas: evitar que el programador cliente

tenga que crear un objeto de esta clase y establecer que solamente es un interfaz. Y esto plantea la cuestión de qué se debe utilizar pues, un interfaz o una clase abastacta. Un interfaz proporciona los beneficios de una clase abstracta y, además, los beneficios propios del interfaz, así que es posible crear una clase base sin la definición de ningún método o variable miembro, por lo que se deberían utilizar mejor los interfaces que las clases abstractas. De hecho, si se desea tener una clase base, la primera elección debería ser un interfaz, y utilizar clases abstractas solamente si es necesario tener alguna variable miembro o algun método implementado.

Paquetes

Para explicar el tema de los paquetes imaginarse una ciudad en la cual hay varios bloques de apartamentos propiedad de una única empresa inmobiliaria. Esta empresa dispone además de comercios, zonas de recreo y almacenes. Se puede pensar en la empresa como una lista de referencias a cada una de sus propiedades; es decir, la inmobiliaria sabe exactamente donde está un apartamento determinado y puede hacer uso de él en el momento en que lo necesite.

Si ahora se mira lo anterior en términos de Java, la empresa inmobiliaria es el *paquete*. Los paquetes agrupan a librerías de clases, como las librerías que contienen información sobre distintas propiedades comerciales. Un paquete será, pues, la mayor unidad lógica de objetos en Java.

Los paquetes se utilizan en Java de forma similar a como se utilizan las librerías en C++, para agrupar funciones y clases, sólo que en Java agrupan diferentes clases y/o interfaces. En ellos las clases son únicas, comparadas con las de otros paquetes, y además proporcionan un método de control de acceso. Los paquetes también proporcionan una forma de *ocultar* clases, evitando que otros programas o paquetes accedan a clases que son de uso exclusivo de una aplicación determinada.

Declaración de Paquetes

Los paquetes se declaran utilizando la palabra **`package`** seguida del nombre del paquete. Esto debe estar al comienzo del fichero fuente, en concreto, debe ser la primera sentencia ejecutable del código Java, excluyendo, claro está, los comentarios y espacios en blanco. Por ejemplo:

```
package mamiferos;
class Ballena {
  . . .
  }
```

En este ejemplo, el nombre del paquete es `mamiferos`. La clase **Ballena** se considera como parte del paquete. La inclusión de nuevas clases en el paquete es muy sencilla, ya que basta con colocar la misma sentencia al comienzo de los ficheros que contengan la declaración de las clases. Como cada clase se debe colocar en un fichero separado, cada uno de los ficheros que contengan clases pertenecientes a un mismo paquete, deben incluir la misma sentencia `package`, y solamente puede haber una sentencia `package` por fichero.

Se recuerda que el compilador Java solamente requiere que se coloquen en ficheros separados las clases que se declaren públicas. Las clases no públicas se pueden colocar en el mismo fichero fuente, al igual que las clases anidadas. Aunque es una buena norma de

programación que todas las clases se encuentren en un único fichero, la sentencia `package` colocada el comienzo de un fichero fuente afectará a todas las clases que se declaren en ese fichero.

Java también soporta el concepto de jeraquía de paquetes. Esto es parecido a la jerarquía de directorios de la mayoría de los sistemas operativos. Se consigue especificando múltiples nombres en la sentencia `package`, separados por puntos. Por ejemplo, en las sentencias siguientes, la clase **Ballena** pertenece al paquete mamiferos que cae dentro de la jerarquía del paquete animales.

```
package animales.mamiferos;
class Ballena {
  . . .
  }
```

Esto permite agrupar clases relacionadas en un solo paquete, y agrupar paquetes relacionados en un paquete más grande. Para referenciar a un miembro de otro paquete, se debe colocar el nombre del paquete antes del nombre de la clase. La siguiente sentencia es un ejemplo de llamada al método *obtenerNombre()* de la clase **Ballena** que pertenece al subpaquete **mamiferos** del paquete **animales**:

```
animales.mamiferos.Ballena.obtenerNombre();
```

La analogía con la jerarquía de directorios se ve reforzada por el intérprete Java, ya que éste requiere que los ficheros **.class** se encuentren físicamente localizados en subdirectorios que coincidan con el nombre del subpaquete. En el ejemplo anterior, si se encontrase en una máquina Unix, la clase **Ballena** debería estar situada en el camino siguiente:

```
animales/mamiferos/Ballena.class
```

Por supuesto, las convenciones en el nombre de los directorios serán diferentes para los distintos sistemas operativos. El compilador Java colocará los ficheros **.class** en el mismo directorio que se encuentren los ficheros fuentes, por lo que puede ser necesario mover los ficheros **.class** resultantes de la compilación al directorio adecuado, en el caso de que no se encuentren los fuentes en el lugar correcto del árbol jerárquico. Aunque los ficheros **.class** también se pueden colocar directamente en el directorio que se desee especificando la opción `-d` (directorio) a la hora de invocar al compilador. La siguiente línea de comando colocará el fichero resultante de la compilación en el subdirectorio `animales/mamiferos/Ballenas`, independientemente de cual sea el directorio desde el cual se esté invocando al compilador.

```
> javac -d animales/mamiferos/Ballena Ballena.java
```

Todas las clases quedan englobadas dentro de un mismo paquete, si no se especifica explíctamente lo contrario, es decir, aunque no se indique nada, las clases pertenecen a un paquete; ya que, como es normal en Java, lo que no se declara explícitamente, toma valores por defecto. En este caso, hay un paquete sin nombre que agrupa a todos los demás paquetes. Si un paquete no tiene nombre, no es posible para los demás paquetes referenciar a ese paquete, por eso es conveniente colocar todas las clases no triviales en paquetes, para que puedan ser referenciadas posteriormente desde cualquier otro programa.

Acceso a Otros Paquetes

Se decía que se pueden referenciar paquetes precediendo con su nombre la clase que se quiere usar. También se puede emplear la palabra clave **`import`**, si se van a colocar múltiples referencias a un mismo paquete, o si el nombre del paquete es muy largo o complicado.

La sentencia **import** se utiliza para incluir una lista de paquetes en los que buscar una clase determinada, y su sintaxis es:

```
import nombre_paquete;
```

Esta sentencia, o grupo de ellas, deben aparecer antes de cualquier declaración de clase en el código fuente. Por ejemplo:

```
import animales.mamiferos.Ballena;
```

En este ejemplo, todos los miembros (variables, métodos) de la clase **Ballena** están accesibles especificando simplemente su nombre, sin tener que precederlo del nombre completo del paquete.

Esta forma de abreviar tienes sus ventajas y sus desventajas. La ventaja principal es que el código no se vuelve demasiado difícil de leer y además es más rápido de teclear. La desventaja fundamental es que resulta más complicado el saber exactamente a qué paquete pertenece un determinado miembro; y esto es especialmente complicado cuando hay muchos paquetes importados.

En la sentencia **import** también se admite la presencia del carácter *****, asterisco. Cuando se emplea, se indica que toda la jerarquía de clases localizada a partir del punto en que se encuentre, debe ser importada, en lugar de indicar solamente una determinada clase. Por ejemplo, la siguiente sentencia indicaría que todas la clases del subpaquete `animales.mamiferos`, deben ser importadas:

```
import animales.mamiferos.*;
```

Esta es una forma simple y sencilla de tener acceso a todas las clases de un determinado paquete. Aunque el uso del asterisco debe hacerse con cautela, porque al ya de por sí lento compilador, si se pone un asterisco, se cargarán todos los paquetes, lo que hará todavía más lenta la compilación. No obstante, el asterisco no tiene impacto alguno a la hora de la ejecución, solamente en tiempo de compilación.

La sentencia **import** se utiliza en casi todos los ejemplos del Tutorial, fundamentalmente para acceder a las distintas partes del API de Java. Por defecto, el conjunto de clases bajo `java.lang.*` se importan siempre; las otras librerías deben ser importadas explícitamente. Por ejemplo, las siguientes líneas de código premiten el acceso a las clases correspondientes a las librerías de manipulacion de imágenes y gráficos:

```
import java.awt.Image;
import java.awt.Graphics;
```

Nomenclatura de Paquetes

Los paquetes pueden nombrarse de cualquier forma que siga el esquema de nomenclatura de Java. Por convenio, no obstante, los nombres de paquetes comienzan por una letra minúscula para hacer más sencillo el reconocimiento de paquetes y clases, cuando se tiene una referencia explícita a una clase. Esto es porque los nombres de las clases, también por convenio, empiezan con una letra mayúscula. Por ejemplo, cuando se usa el convenio citado, es obvio que tanto **animales** como **mamiferos** son paquetes y que

Ballena es una clase. Cuanquier cosa que siga al nombre de la clase es un miembro de esa clase:

```
animales.mamiferos.Ballena.obtenerNombre();
```

Java sigue este convenio en todo el API. Por ejemplo, el método *System.out.println()* que tanto se ha utilizado sigue esta nomenclatura. El nombre del paquete no se declara explícitamente porque `java.lang.*` siempre es importado implícitamente. **System** es el nombre de la clase perteneciente al paquete `java.lang` y está capitalizado. El nombre completo del método es:

```
java.lang.System.out.println();
```

Cada nombre de paquete ha de ser único, para que el uso de paquetes sea realmente efectivo. Los conflictos de nombres pueden causar problemas a la hora de la ejecución en caso de duplicidad, ya que los ficheros de clases podrían saltar de uno a otro directorio. En caso de proyectos pequeños no es difícil mantener una unicidad de nombres, pero en caso de grandes proyectos; o se sigue una norma desde el comienzo del proyecto, o éste se convertirá en un auténtico caos.

No hay ninguna organización en Internet que controle esta nomenclatura, y muchas de las aplicaciones Java corren sobre Web. Hay que tener presente que muchos servidores Web incluyen applets de múltiples orígenes, con lo cual parece poco menos que imposible el evitar que alguien duplique nombres.

Como norma y resumen de todo lo dicho, a la hora de crear un paquete hay que tener presente una serie de ideas:

- La palabra clave `package` debe ser la primera sentencia que aparezca en el fichero, exceptuando, claro está, los espacios en blanco y comentarios
- Es aconsejable que todas las clases que vayan a ser incluidas en el paquete se encuentren en el mismo directorio. Como se ha visto, esta recomendación se la puede uno saltar a la torera, pero se corre el riesgo de que aparezcan determinados problemas difíciles de resolver a la hora de compilar, en el supuesto caso de que no se hile muy fino
- Ante todo, recordar que en un fichero únicamente puede existir, como máximo, una clase con el especificador de acceso `public`, debiendo coincidir el nombre del fichero con el nombre de la clase

Paquetes de Java

El lenguaje Java proporciona una serie de paquetes que incluyen ventanas, utilidades, un sistema de entrada/salida general, herramientas y comunicaciones. En la versión actual del JDK, algunos de los paquetes Java que se incluyen son los que se muestran a continuación, que no es una lista exhaustiva, sino para que el lector pueda tener una idea aproximada de lo que contienen los paquetes más importantes que proporciona el JDK de Sun. Posteriormente, en el desarrollo de otros apartados del Tutorial, se introducirán otros paquetes que también forman parte del JDK y que, incorporan características a Java que hacen de él un lenguaje mucho más potente y versátil, como son los paquetes *Java2D* o *Swing*, que han entrado a formar parte oficial del JDK en la versión **JDK 1.2**.

java.applet

Este paquete contiene clases diseñadas para usar con applets. Hay la clase **Applet** y tres interfaces: **AppletContext**, **AppletStub** y **AudioClip**.

java.awt

El paquete *Abstract Windowing Toolkit* (awt) contiene clases para generar widgets y componentes GUI (Interfaz Gráfico de Usuario), de manipulación de imágenes, impresión, fuentes de caracteres, cursores, etc.. Incluye las clases **Button**, **Checkbox**, **Choice**, **Component**, **Graphics**, **Menu**, **Panel**, **TextArea**, **TextField**...

java.io

El paquete de entrada/salida contiene las clases de acceso a ficheros, de filtrado de información, serialización de objetos, etc.: **FileInputStream**, **FileOutputStream**, **FileReader**, **FileWriter**. También contiene los interfaces que facilitan la utilización de las clases: **DataInput**, **DataOutput**, **Externalizable**, **FileFilter**, **FilenameFilter**, **ObjectInput**, **ObjectOutput**, **Serializable**...

java.lang

Este paquete incluye las clases del lenguaje Java propiamente dicho: **Object**, **Thread**, **Exception**, **System**, **Integer**, **Float**, **Math**, **String**, **Package**, **Process**, **Runtime**, etc.

java.net

Este paquete da soporte a las conexiones del protocolo TCP/IP y, además, incluye las clases **Socket**, **URL** y **URLConnection**.

java.sql

Este paquete incluye todos los interfaces que dan acceso a Bases de Datos a través de **JDBC**, *Java DataBase Connectivity*, como son: **Array**, **Blob**, **Connection**, **Driver**, **Ref**, **ResultSet**, **SQLData**, **SQLInput**, **SQLOutput**, **Statement**, **Struct**; y algunas clases específicas: **Date**, **DriveManager**, **Time**, **Types**...

java.util

Este paquete es una miscelánea de clases útiles para muchas cosas en programación: estructuras de datos, fechas, horas, internacionalización,etc. Se incluyen, entre otras, **Date** (fecha), **Dictionary** (diccionario), **List** (lista), **Map** (mapa), **Random** (números aleatorios) y **Stack** (pila FIFO). Dentro de este paquete, hay tres paquetes muy interesantes: **java.util.jar**, que proporciona clases para leer y crear ficheros **JAR**; **java.util.mime**, que proporciona clases para manipular tipos **MIME**, *Multipurpose Internet Mail Extension* (RFC 2045, RFC 2046) y **java.util.zip**, que proporciona clases para comprimir, descomprimir, calcular checksums de datos, etc. con los formatos estándar ZIP y GZIP.

Referencias

Java especifica sus tipos primitivos, elimina cualquier tipo de punteros y tiene tipos referencia mucho más claros que C y C++.

Todo este maremágnum de terminología provoca cierta consternación, así que los párrafos que siguen intentan aclarar lo que realmente significan los términos que se utilizan.

Se conocen ya ampliamente todos los tipos básicos de datos: datos base, integrados, primitivos e internos; que son muy semejantes en C, C++ y Java; aunque Java simplifica un poco su uso a los desarrolladores haciendo que el chequeo de tipos sea bastante más rígido. Además, Java añade los tipos boolean y hace imprescindible el uso de este tipo booleano en sentencias condicionales.

Punteros

C y C++ permiten la declaración y uso de punteros, que pueden ser utilizados en cualquier lugar. Esta tremenda flexibilidad resulta muy útil, pero también es la causa de que se pueda colgar todo el sistema.

La intención principal en el uso de los punteros es comunicarse más directamente con el hardware, haciendo que el código se acelere. Desafortunadamente, este modelo de tan bajo nivel hace que se pierda robustez y seguridad en la programación y hace muy difíciles tareas como la liberación automática de memoria, la defragmentación de memoria, o realizar programación distribuida de forma clara y eficiente.

Referencias en Java

Las referencias en Java no son punteros ni referencias como en C++. Este hecho crea un poco de confusión entre los programadores que llegan por primera vez a Java. Las referencias en Java son identificadores de instancias de las clases Java. Una referencia dirige la atención a un objeto de un tipo específico. No hay por qué saber cómo lo hace ni se necesita saber qué hace ni, por supuesto, su implementación.

Piénsese en una referencia como si se tratase de la llave electrónica de la habitación de un hotel. Vamos a utilizar precisamente este ejemplo del Hotel para demostrar el uso y la utilización que se puede hacer de las referencias en Java. Primero se crea la clase Habitacion, que está implementada en el fichero , mediante instancias de la cual se levantará el Hotel:

```
public class Habitacion {
    private int numHabitacion;
    private int numCamas;

    public Habitacion() {
        habitacion( 0 );
        }

    public Habitacion( int numeroHab ) {
        habitacion( numeroHab );
        }

    public Habitacion( int numeroHab,int camas ) {
        habitacion( numeroHab );
```

```
        camas( camas );
        }

    public synchornized int habitacion() {
        return( numHabitacion );
        }

    public synchronized void habitacion( int numeroHab ) {
        numHabitacion = numeroHab;
        }

    public synchronized int camas() {
        return( camas );
        }

    public syncronized void camas( int camas ) {
        numCamas = camas;
        }
      }
```

El código anterior sería el corazón de la aplicación. Ahora se construye el Hotel creando Habitaciones y asignándole a cada una de ellas su llave electrónica; tal como muestra el código siguiente, :

```
public class Hotel1 {
    public static void main( String args[] ) {
        Habitacion llaveHab1;               // paso 1
        Habitacion llaveHab2;

        llaveHab1 = new Habitacion( 222 ); // pasos 2, 3, 4 y 5
        llaveHab2 = new Habitacion( 1144,3 );
        ^^^^^^^^^   ^^^^^^^^^^^^^^  ^^^^^^
//
//          A           B y D         C
        }
    }
```

Para explicar el proceso, se dividen las acciones en los cinco pasos necesarios para poder entrar en la habitación del hotel. Aunque no se incluye, se puede también considerar el caso de la necesidad de un cerrajero, para que ante la pérdida de la llave se pueda abrir la puerta; y que, en el caso particular de este hotel virtual Java, sería el garbage collector, que recicla la habitación una vez que se hayan perdido todas las llaves.

El primer paso es la creación de la llave, es decir, definir la variable referencia, por defecto nula.

El resto de los pasos se agrupan en una sola sentencia Java. La parte B en el código anterior indica al gerente del Hotel que ya dispone de una nueva habitación. La parte C llama al decorador de interiores para que "vista" la habitación según un patrón determinado, para que no desentonen unas habitaciones con otras y no se pierdan las señas de identidad del hotel. El código electrónico que permitirá acceder a la habitación se genera en la parte D, una vez conocido el interior de la habitación y ya se programa en la llave en la parte A.

Si se abandona el ejemplo real a un lado y se detiene uno en lo que ocurre en la ejecución del código, se observa que el operador new busca espacio para una instancia de un objeto de una clase determinada e inicializa la memoria a los valores adecuados. Luego invoca al método constructor de la clase, proporcionándole los argumentos adecuados. El operador new devuelve una referencia a sí mismo, que es inmediatamente asignada a la variable referencia.

Se pueden tener múltiples llaves para una misma habitación:

```
. . .
Habitacion llaveHab3,llaveHab4;
llaveHab3 = llaveHab1;
 llaveHab4 = llavehab2;
```

De este modo se consiguen copias de las llaves. Las habitaciones en sí mismas no se han tocado en este proceso. Así que, ya hay dos llaves para la habitación 222 y otras dos para la habitación 1144.

Una llave puede ser programada para que funcione solamente con una habitación en cualquier momento, pero se puede cambiar su código electrónico para que funcione con alguna otra habitación; por ejemplo, para cambiar una habitación anteriormente utilizada por un empedernido fumador por otra limpia de olores y con vistas al mar. Se cambiaría la llave duplicada de la habitación del fumador (222) por la habitación con olor a sal marina, 1144:

```
. . .
 llaveHab3 = llaveHab2;
```

Ahora hay una llave para la habitación 222 y tres para la habitación 1144; manteniendo una llave para cada habitación en la conserjería, para poder utilizarla como llave maestra, en el caso de que alguien pierda su llave propia.

Alguien con la llave de una habitación puede hacer cambios en ella, y los compañeros que tengan llave de esa misma habitación, no tendrán conocimiento de esos cambios hasta que vuelvan a entrar en la habitación. Por ejemplo, si que quita una de las camas de la habitación, entrando en ella con la llave maestra:

```
. . .
llaveHab2.camas( 2 );
```

Ahora cuando los inquilinos entren en la habitación podrán comprobar el cambio realizado:

```
. . .
llaveHab4.printData();
```

Referencias y Arrays

Como en C y C++, Java dispone de arrays de tipos primitivos o de clases. Los arrays en C y C++ son básicamente un acompañante para los punteros. En Java, sin embargo, son ciudadanos de primera clase.

Para expandir el hotel creando todo un ala de habitaciones, , se creará un juego de llaves maestras y luego se construirán las habitaciones:

```
public class Hotel2 {
    // Número de habitaciones por ala
    public static final int habPorAla = 12;

    public static void main( String args[] ) {
        Habitacion llavesMaestras[];                    // paso 1
        llavesMaestras = new Habitacion[ habPorAla ]; // pasos 2-5
        ^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//
//            A                   B, C, D y E
        int numPiso = 1;
        for( int i=0; i < habPorAla; i++ )          // pasos 6-9
            llavesMaestras[ i ] = new Habitacion( numPiso * 100 + i,
                ( 0 == (i%2)) ? 2 : 1 );
```

```
        for( int i=0; i < habPorAla; i++ )          // pasos 10-11
            llavesMaestras[i].printData();
        }
    }
```

Cada paso en el ejemplo es semejante al que ya se ha visto antes. El paso 1 especifica que el juego de llaves maestras es un grupo de llaves de habitaciones.

Los pasos 2 a 5 son, en este caso, la parte principal. En lugar de crear una habitación, el gerente ordena construir un grupo contiguo de habitaciones. El número de llaves se especifica entre corchetes y todas se crean en blanco.

Los pasos 6 a 9 son idénticos a los pasos 2 a 5 del ejemplo anterior, excepto en que en este caso todas las llaves pasan a formar parte del juego maestro. Los números de piso se dan en miles para que cuando se creen las habitaciones, todas tengan el mismo formato. También todas las habitaciones de número par tienen una sola cama, mientras que las habitaciones impares tendrán dos camas.

Los pasos 10 y 11 permiten obtener información de cada una de las habitaciones del hotel.

Referencias y Listas

Hay gente que piensa que como Java no dispone de punteros, resulta demasiado complejo construir listas enlazadas, árboles binarios y grafos. Java dispone de estructuras de datos de esos tipos y además proporciona otras muchas como: mapas, conjuntos, tablas Hash, diccionarios, etc., que en el JDK 1.2 están muy bien diseñadas y siguen la nomenclatura al uso de la OOP. No obstante, en los párrafos siguientes se demuestra que quien así piensa está bastante equivocado, porque incluso de forma pedestre se pueden construir estas estructuras de datos sin demasiado esfuerzo.

Hay que retomar el ejemplo de los arrays, y en vez de éstos utilizar una lista doblemente enlazada. El paquete de la lista simple se compone de dos clases. Cada elemento de la lista es un `NodoListaEnlazada`, :

```
public class NodoListaEnlazada {
    private NodoListaEnlazada siguiente;
    private NodoListaEnlazada anterior;
    private Object datos;
    // . . .
    }
```

Cada NodoListaEnlazada contiene una referencia a su nodo precedente en la lista y una referencia al nodo que le sigue. También contiene una referencia genérica a cualquier clase que se use para proporcionar acceso a los datos que el usuario proporcione.

La lista enlazada, , contiene un nodo principio-fin y un contador para el número de nodos en la lista:

```
public class ListaEnlazada {
    public NodoListaEnlazada PrincipioFin;
    private int numNodos;
    // . . .
    }
```

El nodo especial PrincipioFin es sencillo, para simplificar el código. El contador se usa para optimizar los casos más habituales.

Hay que revisar pues el código del Hotel, ahora , que será prácticamente el mismo que en el caso de los arrays:

```
public class Hotel3 {
     // Número de habitaciones por ala
    public static final int habPorAla = 12;

    public static void main( String args[] ) {
        ListaEnlazada llaveMaestra;           // paso 1
        llaveMaestra = new ListaEnlazada();   // pasos 2-5

        int numPiso = 1;
        for( int i=0; i < habPorAla; i++ ) // pasos 6-9
            llaveMaestra.insertAt( i,
                new Habitacion( numPiso * 100 + i,
                ( 0 == (i%2)) ? 2 : 1 );
        for( int i=0; i < habPorAla; i++ ) // pasos 10-12
            ( (Habitacion)llaveMaestra.getAt(i) ).printData();
        }
    }
```

El paso 1 es la llave maestra de la lista. Está representada por una lista generica; es decir, una lista de llaves que cumple la convención que se ha establecido. Se podría acelerar el tiempo de compilación metiendo la lista genérica ListaEnlazada dentro de una ListaEnlazadaHabitacion.

Los pasos 2 a 5 son equivalentes a los del primer ejemplo. Se construye e inicializa una nueva ListaEnlazada, que se usará como juego de llaves maestras.

Los pasos 6 a 9 son funcionalmente idénticos a los del ejemplo anterior con arrays, pero con diferente sintaxis. En Java, los arrays y el operador [] son internos del lenguaje. Como Java no soporta la sobrecarga de operadores por parte del usuario, solamente se puede utilizar en su forma normal.

La ListaEnlazada proporciona el método insertAt() que coge el índice en la lista, donde el nuevo nodo ha de ser insertado, como primer argumento. El segundo argumento es el objeto que será almacenado en la lista. Obsérvese que no es necesario colocar moldeo alguno para hacer algo a una clase descendiente que depende de uno de sus padres.

Los pasos 10 a 12 provocan la misma salida que los pasos 10 y 11 del ejemplo con arrays. El paso 10 coge la llave del juego que se indica en el método getAt(). En este momento, el sistema no sabe qué datos contiene la llave, porque el contenido de la habitación es genérico. Pero el programa sí sabe lo que hay en la lista, así que informa al sistema haciendo un moldeado a la llave de la habitación (este casting generará un chequeo en tiempo de ejecución por parte del compilador, para asegurarse de que se trata de una Habitacion). El paso 12 usa la llave para imprimir la información.

Arrays

Java,dispone de un tipo array.En Java, al ser un tipo de datos verdadero, se dispone de comprobaciones exhaustivas del correcto manejo del array; por ejemplo, de la comprobación de sobrepasar los límites definidos para el array, en evitación de desbordamiento o corrupción de memoria.

En Java hay que declarar un array antes de poder utilizarlo. Y en la declaración hay que incluir el nombre del array y el tipo de datos que se van a almacenar en él. La sintaxis general para declarar e instanciar un array es:

```
tipoDeElementos[] nombreDelArray = new tipoDeElementos[tamañoDelArray];
```

Se pueden declarar en Java arrays de cualquier tipo:

```
char s[];

int iArray[];
```

Incluso se pueden construir arrays de arrays:

```
int tabla[][] = new int[4][5];
```

Al igual que los demás objetos en Java, la declaración del array no localiza, o reserva, memoria para contener los datos. En su lugar, simplemente localiza memoria para almacenar una referencia al array. La memoria necesaria para almacenar los datos que componen el array se buscará en memoria dinámica a la hora de instanciar y crear realmente el array.

Para crear un array en Java hay dos métodos básicos. Crear un array vacío:

```
int lista[] = new int[50];
```

o se puede crear ya el array con sus valores iniciales:

```
String nombres[] = {
    "Juan","Pepe","Pedro","Maria"
    };
```

Esto que es equivalente a:

```
String nombres[];

nombres = new String[4];

nombres[0] = new String( "Juan" );

nombres[1] = new String( "Pepe" );

nombres[2] = new String( "Pedro" );

nombres[3] = new String( "Maria" );
```

No se pueden crear arrays estáticos en tiempo de compilación:

```
int lista[50]; // generará un error en tiempo
```

```
                // de compilación
```

Tampoco se puede rellenar un array sin declarar el tamaño con el operador *new*:

```
int lista[];

for( int i=0; i < 9; i++ )

    lista[i] = i;
```

En las sentencias anteriores simultáneamente se declara el nombre del array y se reserva memoria para contener sus datos. Sin embargo, no es necesario combinar estos procesos. Se puede ejecutar la sentencia de declaración del array y posteriormente, otra sentencia para asignar valores a los datos.

Una vez que se ha instanciado un array, se puede acceder a los elementos de ese array utilizando un índice, de forma similar a la que se accede en otros lenguajes de programación. Sin embargo, Java no permite que se acceda a los elementos de un array utilizando punteros.

```
    miArray[5] = 6;

    miVariable = miArray[5];
```

Los índices de un array siempre empiezan en 0.

Todos los arrays en Java tienen una función miembro: ***length()***, que se puede utilizar para conocer la longitud del array.

```
int a[][] = new int[10][3];

a.length;     /* 10 */

a[0].length; /* 3 */
```

Además, en Java no se puede rellenar un array sin declarar el tamaño con el operador **new**. El siguiente código no sería válido:

```
int lista[];

for( int i=0; i < 9; i++ )

    lista[i] = i;
```

Es decir, todos los arrays en Java son estáticos. Para convertir un array en el equivalente a un array dinámico en C/C++, se usa la clase Vector, que permite operaciones de inserción, borrado, etc. en el array.

El ejemplo java403.java, intenta ilustrar un aspecto interesante del uso de arrays en Java. Se trata de que Java puede crear arrays multidimensionales, que se verían como arrays de arrays. Aunque esta es una característica común a muchos lenguajes de programación, en Java, sin embargo, los arrays secundarios no necesitan ser todos del mismo tamaño. En el ejemplo, se declara e instancia un array de enteros, de dos dimensiones, con un tamaño inicial (tamaño de la primera dimensión) de 3. Los tamaños de las dimensiones secundarias (tamaño de cada uno de los subarrays) es 2, 3 y 4, respectivamente.

En este ejemplo, también se pueden observar alguna otra cosa, como es el hecho de que cuando se declara un array de dos dimensiones, no es necesario indicar el tamaño de la dimensión secundaria a la hora de la declaración del array, sino que puede declararse posteriormente el tamaño de cada uno de los subarrays. También se puede observar el resultado de acceder a elementos que se encuentran fuera de los límites del array; Java protege la aplicación contra este tipo de errores de programación.

En C/C++, cuando se intenta acceder y almacenar datos en un elemento del array que está fuera de límites, siempre se consigue, aunque los datos se almacenen en una zona de memoria que no forme parte del array. En Java, si se intenta hacer esto, el Sistema generará una excepción, tal como se muestra en el ejemplo. En él, la excepción simplemente hace que el programa termine, pero se podría recoger esa excepción e implementar un controlador de excepciones (*exception handler*) para corregir automáticamente el error, sin necesidad de abortar la ejecución del programa.

La salida por pantalla de la ejecución del ejemplo sería:

```
%java java403

00

012

0246

Acceso a un elemento fuera de limites

java.lang.ArrayIndexOutOfBoundsException: at java403.main(java403.java:55)
```

El ejemplo java404.java ilustra otro aspecto del uso de arrays en Java. Se trata, en este caso, de que Java permite la asignación de un array a otro. Sin embargo, hay que extremar las precauciones cuando se hace esto, porque lo que en realidad está pasando es que se está haciendo una copia de la referencia a los mismos datos en memoria. Y, tener dos referencias a los mismos datos no parece ser una buena idea, porque los resultados pueden despistar.

La salida por pantalla que se genera tras la ejecución del ejemplo es:

```
%java java404

Contenido del primerArray

0 1 2

Contenido del segundoArray

0 1 2

--> Cambiamos un valor en el primerArray

Contenido del primerArray

0 10 2

Contenido del segundoArray

0 10 2
```

Excepciones en Java

Una excepción es un evento que ocurre durante la ejecución de un programa y detiene el flujo normal de la secuencia de instrucciones de ese programa; en otras palabras, una excepción es una condición anormal que surge en una secuencia de código durante su ejecución.

Las excepciones en Java están destinadas, al igual que en el resto de los lenguajes que las soportan, para la detección y corrección de errores. Si hay un error, la aplicación no debería morirse y generar un core (o un crash en caso del DOS). Se debería lanzar (throw) una excepción que a su vez debería capturar (catch) y resolver la situación de error, o poder ser tratada finalmente (finally) por un gestor por defecto u omisión. Java sigue el mismo modelo de excepciones que se utiliza en C++. Utilizadas en forma adecuada, las excepciones aumentan en gran medida la robustez de las aplicaciones.

La gestión de excepciones en Java proporciona un mecanismo excepcionalmente poderoso para controlar programas que tengan muchas características dinámicas durante su ejecución. Las excepciones son formas muy limpias de manejar errores y problemas inesperados en la lógica del programa, y no deberían considerarse como un mecanismo general de ramificaciones o un tipo de sentencias de salto. Los lugares más indicados para utilizar excepciones son aquellos en los que se usan valores como 0 o 1, en C/C++, para indicar algún fallo funcional. Por ejemplo:

```
#include <sys/errno.h>

int fd;
fd = open( "leeme.txt" );
if( fd == -1 && errno == EEXIT )
    fd = open( "defecto.txt" );
    }
```

En este programa C, si falla la primera sentencia open() por cualquiera de las 19 razones distintas de EEXIT por las que puede fallar, entonces el programa se continuaría ejecutando y moriría por alguna razón misteriosa más adelante, dejando atrás un problema de depuración complicado y frustrante.

La versión Java del código anterior, tal como se muestra a continuación:

```
FilterReader fr;
try {
    Fr = new FilterReader( "leeme.txt" );
} catch( FileNotFoundException e ) {
    fr = new FilterReader( "defecto.txt" );
    }
```

proporciona una oportunidad para capturar una excepción más genérica y tratar la situación con elegancia o, en el peor de los casos, imprimiría el estado de la pila de memoria.

Por ello, que la utilización adecuada de las excepciones proporcionará un refinamiento profesional al código que cualquier usuario futuro de las aplicaciones que salgan de la mano del programador que las utilice agradecerá con toda seguridad.

Manejo de excepciones

A continuación se muestra cómo se utilizan las excepciones, reconvirtiendo en primer lugar el applet de saludo a partir de la versión iterativa de :

```
import java.awt.*;
import java.applet.Applet;

public class HolaIte extends Applet {
    private int i = 0;
    private String Saludos[] = {
        "Hola Mundo!",
        "HOLA Mundo!",
        "HOLA MUNDO!!"
        };

    public void paint( Graphics g ) {
        g.drawString( Saludos[i],25,25 );
        i++;
        }
    }
```

Normalmente, un programa termina con un mensaje de error cuando se lanza una excepción. Sin embargo, Java tiene mecanismos para excepciones que permiten ver qué excepción se ha producido e intentar recuperarse de ella.

Vamos a reescribir el método paint() de esa versión iterativa del saludo:

```
public void paint( Graphics g ) {
    try {
        g.drawString( Saludos[i],25,25 );
        } catch( ArrayIndexOutOfBoundsException e ) {
            g.drawString( "Saludos desbordado",25,25 );
        } catch( Exception e ) {
                // Cualquier otra excepción
            System.out.println( e.toString() );
        } finally {
            System.out.println( "Esto se imprime siempre!" );
        }
    i++;
    }
```

La palabra clave finally define un bloque de código que se quiere que sea ejecutado siempre, de acuerdo a si se capturó la excepción o no. En el ejemplo anterior, la salida en la consola, con i=4 sería:

```
C:\>java HolaIte
Saludos desbordado
¡Esto se imprime siempre!
```

Generar excepciones en Java

Cuando se produce una condición excepcional en el transcurso de la ejecución de un programa, se debería generar, o lanzar, una excepción. Esta excepción es un objeto derivado directa, o indirectamente, de la clase Throwable. Tanto el intérprete Java como muchos métodos de las múltiples clases de Java pueden lanzar excepciones y errores.

La clase Throwable tiene dos subclases: Error y Exception. Un Error indica que se ha producido un fallo no recuperable, del que no se puede recuperar la ejecución normal del programa, por lo tanto, en este caso no hay nada que hacer. Los errores, normalmente,

hacen que el intérprete Java presente un mensaje en el dispositivo estándar de salida y concluya la ejecución del programa. El único caso en que esto no es así, es cuando se produce la muerte de un thread, en cuyo caso se genera el error ThreadDead, que lo que hace es concluir la ejecución de ese hilo, pero ni presenta mensajes en pantalla ni afecto a otros hilos que se estén ejecutando.

Una Exception indicará una condición anormal que puede ser subsanada para evitar la terminación de la ejecución del programa. Hay nueve subclases de la clase Exception ya predefinidas, y cada una de ellas, a su vez, tiene numerosas subclases.

Para que un método en Java, pueda lanzar excepciones, hay que indicarlo expresamente.

```
void MetodoAsesino() throws NullPointerException,CaidaException
```

Se pueden definir excepciones propias, no hay por qué limitarse a las nueve predefinidas y a sus subclases; bastará con extender la clase Exception y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción.

También pueden producirse excepciones no de forma explícita como en el caso anterior, sino de forma implícita cuando se realiza alguna acción ilegal o no válida.

Las excepciones, pues, pueden originarse de dos modos: el programa hace algo ilegal (caso normal), o el programa explícitamente genera una excepción ejecutando la sentencia throw (caso menos normal). La sentencia throw tiene la siguiente forma:

```
    throw ObtejoExcepction;
```

El objeto ObjetoException es un objeto de una clase que extiende la clase Exception.

El siguiente código de ejemplo, , origina una excepción de división por cero:

```
class java901 {
    public static void main( String[] a ) {
        int i=0, j=0, k;

        k = i/j;    // Origina un error de division-by-zero
        }
    }
```

Si compilamos y ejecutamos esta aplicación Java, obtendremos la siguiente salida por pantalla:

```
% javac java901.java
% java java901
java.lang.ArithmeticException:/by zero at java901.main(java901.java:25)
```

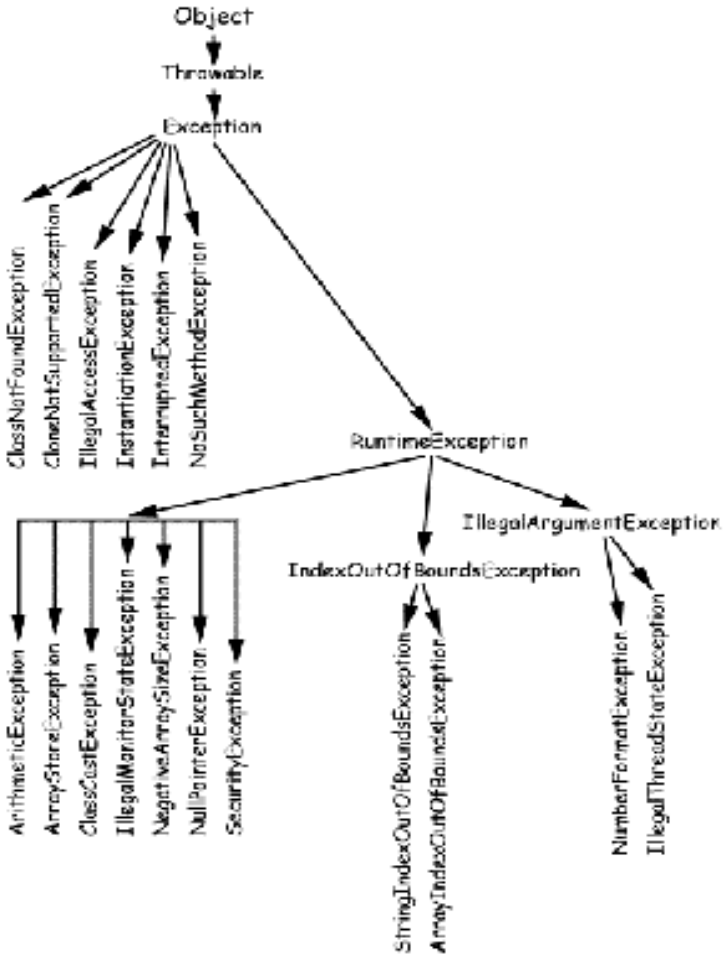
Las excepciones predefinidas, como ArithmeticException, se conocen como excepciones runtime. Actualmente, como todas las excepciones son eventos runtime, sería mejor llamarlas excepciones irrecuperables. Esto contrasta con las excepciones que se generan explícitamente, a petición del programador, que suelen ser mucho menos severas y en la mayoría de los casos no resulta complicado recuperarse de ellas. Por ejemplo, si un fichero no puede abrirse, se puede preguntar al usuario que indique otro fichero; o si una estructura de datos se encuentra completa, siempre se podrá sobreescribir algún elemento que ya no se necesite.

Todas las excepciones deben llevar un mensaje asociado a ellas al que se puede acceder utilizando el método **getMessage**(), que presentará un mensaje describiendo el error o la excepción que se ha producido.

Si se desea, se pueden invocar otros métodos de la clase Throwable que presentan un traceado de la pila en donde se ha producido la excepción, o también se pueden invocar para convertir el objeto Exception en una cadena, que siempre es más intelegible y agradable a la vista.

Excepciones Predefinidas

Las excepciones predefinidas por la implementación actual del lenguaje Java y su jerarquía interna de clases son las que se representan en el esquema de la figura que aparece a continuación



Los nombres de las excepciones indican la condición de error que representan. Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:

ArithmeticException

Las excepciones aritméticas son típicamente el resultado de división por 0:

```
int i = 12 / 0;
```

NullPointerException

Se produce cuando se intenta acceder a una variable o método antes de ser definido:

```
class Hola extends Applet {
    Image img;

    paint( Graphics g ) {
        g.drawImage( img,25,25,this );
        }
    }
```

IncompatibleClassChangeException

El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

ClassCastException

El intento de convertir un objeto a otra clase que no es válida.

```
y = (Prueba)x;      // donde x no es de tipo Prueba
```

NegativeArraySizeException

Puede ocurrir si hay un error aritmético al cambiar el tamaño de un array.

OutOfMemoryException

¡No debería producirse nunca! El intento de crear un objeto con el operador new ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el garbage collector se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

NoClassDefFoundException

Se referenció una clase que el sistema es incapaz de encontrar.

ArrayIndexOutOfBoundsException

Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

UnsatisfiedLinkException

Se hizo el intento de acceder a un método nativo que no existe. Aquí no existe un método a.kk()

```
class A {
    native void kk();
    }
```

y se llama a a.kk(), cuando debería llamar a A.kk().

InternalException

Este error se reserva para eventos que no deberían ocurrir. Por definición, el usuario nunca debería ver este error y esta excepción no debería lanzarse.

El compilador Java obliga al programador a proporcionar el código de manejo o control de algunas de las excepciones predefinidas por el lenguaje. Por ejemplo, el siguiente programa , no compilará porque no se captura la excepción InterruptedException que puede lanzar el método sleep().

```
import java.lang.Thread;

class java902 {
    public static void main( String args[] ) {
        java902 obj = new java902();
        obj.miMetodo();
        }

    void miMetodo() {
        // Aqui se produce el error de compilacion, porque no se esta
        // declarando la excepcion que genera este metodo
        Thread.currentThread().sleep( 1000 ); // currentThread() genera
                                              // una excepcion
        }
    }
```

Este es un programa muy simple, que al intentar compilar, producirá el siguiente error de compilación que se visualizará en la pantalla tal como se reproduce a continuación:

```
% javac java902.java
java902.java:41:   Exception   java.lang.InterruptedException   must   be
caught,or it must be declared in the throws clause of this method.
Thread.currentThread().sleep( 1000 ); // currentThread() genera
                                      ^
```

Como no se ha previsto la captura de la excepción, el programa no compila. El error identifica la llamada al método sleep() como origen del problema. Así que, la siguiente versión del programa, , soluciona el problema generado por esta llamada.

```
import java.lang.Thread;

class java903 {
    public static void main( String args[] ) {
        // Se instancia un objeto
        java903 obj = new java903();
        // Se crea la secuencia try/catch que llamara al metodo que
        // lanza la excepcion
        try {
            // Llamada al metodo que genera la excepcion
            obj.miMetodo();
            }catch(InterruptedException e){} // Procesa la excepcion
        }

    // Este es el metodo que va a lanzar la excepcion
    void miMetodo() throws InterruptedException {
        Thread.currentThread().sleep( 1000 ); // currentThread() genera
                                              // una excepcion
        }
    }
```

Lo único que se ha hecho es indicar al compilador que el método miMetodo() puede lanzar excepciones de tipo InterruptedException. Con ello conseguimos propagar las excepción que genera el método sleep() al nivel siguiente de la jerarquía de clases. Es

decir, en realidad no se resuelve el problema sino que se está pasando a otro método para que lo resuelva él.

En el método main() se proporciona la estructura que resuelve el problema de compilación, aunque no haga nada, por el momento. Esta estructura consta de un bloque try y un bloque catch, que se puede interpretar como que intentará ejecutar el código del bloque try y si hubiese una nueva excepción del tipo que indica el bloque catch, se ejecutaría el código de este bloque, si ejecutar nada del try.

La transferencia de control al bloque catch no es una llamada a un método, es una transferencia incondicional, es decir, no hay un retorno de un bloque catch.

Crear Excepciones Propias

También el programador puede lanzar sus propias excepciones, extendiendo la clase System.exception. Por ejemplo, considérese un programa cliente/servidor. El código cliente se intenta conectar al servidor, y durante 5 segundos se espera a que conteste el servidor. Si el servidor no responde, el servidor lanzaría la excepción de time-out:

```
class ServerTimeOutException extends Exception {}

public void conectame( String nombreServidor ) throws Exception {
    int exito;
    int puerto = 80;

    exito = open( nombreServidor,puerto );
    if( exito == -1 )
        throw ServerTimeOutException;
    }
```

Si se quieren capturar las propias excepciones, se deberá utilizar la sentencia try:

```
public void encuentraServidor() {
   ...
   try {
        conectame( servidorDefecto );
        catch( ServerTimeOutException e ) {
            g.drawString(
                "Time-out del Servidor, intentando alternativa",5,5 );
            conectame( servidorAlterno );
            }
    ...
    }
```

Cualquier método que lance una excepción también debe capturarla, o declararla como parte del interfaz del método. Cabe preguntarse entonces, el porqué de lanzar una excepción si hay que capturarla en el mismo método. La respuesta es que las excepciones no simplifican el trabajo del control de errores. Tienen la ventaja de que se puede tener muy localizado el control de errores y no hay que controlar millones de valores de retorno, pero no van más allá.

Y todavía se puede plantear una pregunta más, al respecto de cuándo crear excepciones propias y no utilizar las múltiples que ya proporciona Java. Como guía, se pueden plantear las siguientes cuestiones, y si la respuesta es afirmativa, lo más adecuado será implementar una clase Exception nueva y, en caso contrario, utilizar una del sistema.

- ¿Se necesita un tipo de excepción no representado en las que proporciona el entorno de desarrollo Java?

- ¿Ayudaría a los usuarios si pudiesen diferenciar las excepciones propias de las que lanzan las clases de otros desarrolladores?
- ¿Si se lanzan las excepciones propias, los usuarios tendrán acceso a esas excepciones?
- ¿El package propio debe ser independiente y auto-contenido?

Captura de excepciones

Las excepciones lanzadas por un método que pueda hacerlo deben recoger en bloque try/catch o try/finally.

```
int valor;
try {
    for( x=0,valor = 100; x < 100; x ++ )
        valor /= x;
    }
catch( ArithmeticException e ) {
    System.out.println( "Matemáticas locas!" );
    }
catch( Exception e ) {
    System.out.println( "Se ha producido un error" );
    }
```

try

Es el bloque de código donde se prevé que se genere una excepción. Es como si dijésemos "intenta estas sentencias y mira a ver si se produce una excepción". El bloque try tiene que ir seguido, al menos, por una cláusula catch o una cláusula finally.

La sintaxis general del bloque try consiste en la palabra clave try y una o más sentencias entre llaves.

```
try {
    // Sentencias Java
    }
```

Puede haber más de una sentencia que genere excepciones, en cuyo caso habría que proporcionar un bloque try para cada una de ellas. Algunas sentencias, en especial aquellas que invocan a otros métodos, pueden lanzar, potencialmente, muchos tipos diferentes de excepciones, por lo que un bloque try consistente en una sola sentencia requeriría varios controladores de excepciones.

También se puede dar el caso contrario, en que todas las sentencias, o varias de ellas, que puedan lanzar excepciones se encuentren en un único bloque try, con lo que habría que asociar múltiples controladores a ese bloque. Aquí la experiencia del programador es la que cuenta y es el propio programador el que debe decidir qué opción tomar en cada caso.

Los controladores de excepciones deben colocarse inmediatamente después del bloque try. Si se produce una excepción dentro del bloque try, esa excepción será manejada por el controlador que esté asociado con el bloque try.

catch

Es el código que se ejecuta cuando se produce la excepción. Es como si dijésemos "controlo cualquier excepción que coincida con mi argumento". No hay código alguno entre un bloque try y un bloque catch, ni entre bloques catch. La sintaxis general de la sentencia catch en Java es la siguiente:

```
catch( UnTipoTrhowable nombreVariable ) {
    // sentencias Java
    }
```

El argumento de la sentencia declara el tipo de excepción que el controlador, el bloque catch, va a manejar.

En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

```
    catch( Excepcion e ) { ...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

Se pueden controlar grupos de excepciones, es decir, que se pueden controlar, a través del argumento, excepciones semejantes. Por ejemplo:

```
class Limites extends Exception {}
class demasiadoCalor extends Limites {}
class demasiadoFrio extends Limites {}
class demasiadoRapido extends Limites {}
class demasiadoCansado extends Limites {}
    .
    .
    .
    try {
        if( temp > 40 )
            throw( new demasiadoCalor() );
        if( dormir < 8 )
            throw( new demasiado Cansado() );
    } catch( Limites lim ) {
        if( lim instanceof demasiadoCalor ) {
            System.out.println( "Capturada excesivo calor!" );
            return;
            }
        if( lim instanceof demasiadoCansado ) {
            System.out.println( "Capturada excesivo cansancio!" );
            return;
            }
    } finally
        System.out.println( "En la clausula finally" );
```

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque. El operador instanceof se utiliza para identificar exactamente cual ha sido la identidad de la excepción.

Cuando se colocan varios controladores de excepción, es decir, varias sentencias catch, el orden en que aparecen en el programa es importante, especialmente si alguno de los controladores engloba a otros en el árbol de jerarquía. Se deben colocar primero los controladores que manejen las excepciones más alejadas en el árbol de jerarquía, porque de

otro modo, estas excepciones podrían no llegar a tratarse si son recogidas por un controlador más general colocado anteriormente.

Por lo tanto, los controladores de excepciones que se pueden escribir en Java son más o menos especializados, dependiendo del tipo de excepciones que traten. Es decir, se puede escribir un controlador que maneje cualquier clase que herede de Throwable; si se escribe para una clase que no tiene subclases, se estará implementando un controlador especializado, ya que solamente podrá manejar excepciones de ese tipo; pero, si se escribe un controlador para una clase nodo, que tiene más subclases, se estará implementando un controlador más general, ya que podrá manejar excepciones del tipo de la clase nodo y de sus subclases.

finally

Es el bloque de código que se ejecuta siempre, haya o no excepción. Hay una cierta controversia entre su utilidad, pero, por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejar grabado si se producen excepciones y si el programa se ha recuperado de ellas o no.

Este bloque finally puede ser útil cuando no hay ninguna excepción. Es un trozo de código que se ejecuta independientemente de lo que se haga en el bloque try.

A la hora de tratar una excepción, se plantea el problema de qué acciones se van a tomar. En la mayoría de los casos, bastará con presentar una indicación de error al usuario y un mensaje avisándolo de que se ha producido un error y que decida si quiere o no continuar con la ejecución del programa.

Por ejemplo, se podría disponer de un diálogo como el que se presenta en el código siguiente:

```
public class DialogoError extends Dialog {
    DialogoError( Frame padre ) {
        super( padre,true );
        setLayout( new BorderLayout() );

        // Presentamos un panel con continuar o salir
        Panel p = new Panel();
        p.add( new Button( "¿Continuar?" ) );
        p.add( new Button( "Salir" ) );

        add( "Center",new Label(
            "Se ha producido un error. ¿Continuar?" ) )
        add( "South",p );
        }
    public boolean action( Event evt,Object obj ) {
        if( "Salir".equals( obj ) ) {
            dispose();
            System.exit( 1 );
            }
        return( false );
        }
    }
```

Y la invocación, desde algún lugar en que se suponga que se generarán errores, podría ser como sigue:

```
try {
    // Código peligroso
```

```
    }
catch( AlgunaExcepcion e ) {
    VentanaError = new DialogoError( this );
    VentanaError.show();
    }
```

En el programa , se intenta demostrar el poder del bloque finally. En él, un controlador de excepciones intenta terminar la ejecución del programa ejecutando una sentencia return. Antes de que la sentencia se ejecute, el control se pasa al bloque finally y se ejecutan todas las sentencias de este bloque. Luego el programa termina. Es decir, quedaría demostrado que el bloque finally no tiene la última palabra palabra.

El programa redefine el método getMessage() de la clase Throwable, porque este método devuelve null si no es adecuadamente redefinido por la nueva clase excepción.

throw

La sentencia throw se utiliza para lanzar explícitamente una excepción. En primer lugar se debe obtener un descriptor de un objeto Throwable, bien mediante un parámetro en una cláusula catch o, se puede crear utilizando el operador new. La forma general de la sentencia throw es:

```
throw ObjetoThrowable;
```

El flujo de la ejecución se detiene inmediatamente después de la sentencia throw, y nunca se llega a la sentencia siguiente. Se inspecciona el bloque try que la engloba más cercano, para ver si tiene la cláusula catch cuyo tipo coincide con el del objeto o instancia Throwable. Si se encuentra, el control se transfiere a ese sentencia. Si no, se inspecciona el siguiente bloque try que la engloba, y así sucesivamente, hasta que el gestor de excepciones más externo detiene el programa y saca por pantalla el trazado de lo que hay en la pila hasta que se alcanzó la sentencia throw. En el programa siguiente, , se demuestra como se hace el lanzamiento de una nueva instancia de una excepción, y también cómo dentro del gestor se vuelve a lanzar la misma excepción al gestor más externo.

```
class java905 {
    static void demoproc() {
        try {
            throw new NullPointerException( "demo" );
        } catch( NullPointerException e ) {
            System.out.println( "Capturada la excepcion en demoproc" );
            throw e;
            }
        }

    public static void main( String args[] ) {
        try {
            demoproc();
        } catch( NullPointerException e ) {
            System.out.println( "Capturada de nuevo: " + e );
            }
        }
    }
```

Este ejemplo dispone de dos oportunidades para tratar el mismo error. Primero, main() establece un contexto de excepción y después se llama al método demoproc(), que establece otro contexto de gestión de excepciones y lanza inmediatamente una nueva instancia de la excepción. Esta excepción se captura en la línea siguiente. La salida que se obtiene tras la ejecución de esta aplicación es la que se reproduce:

```
% java java905
Capturada la excepcion en demoproc
Capturada de nuevo: java.lang.NullPointerException: demo
```

throws

Si un método es capaz de provocar una excepción que no maneja él mismo, debería especificar este comportamiento, para que todos los métodos que lo llamen puedan colocar protecciones frente a esa excepción. La palabra clave throws se utiliza para identificar la lista posible de excepciones que un método puede lanzar. Para la mayoría de las subclase de la clase Exception, el compilador Java obliga a declarar qué tipos podrá lanzar un método. Si el tipo de excepción es Error o RuntimeException, o cualquiera de sus subclases, no se aplica esta regla, dado que no se espera que se produzcan como resultado del funcionamiento normal del programa. Si un método lanza explícitamente una instancia de Exception o de sus subclases, a excepción de la excepción de runtime, se debe declarar su tipo con la sentencia throws. La declaración del método sigue ahora la sintaxis siguiente:

```
type NombreMetodo( argumentos ) throws excepciones { }
```

En el ejemplo siguiente, , el programa intenta lanzar una excepción sin tener código para capturarla, y tampoco utiliza throws para declarar que se lanza esta excepción. Por tanto, el código no será posible compilarlo.

```
class java906 {
    static void demoproc() {
        System.out.println( "Capturada la excepcion en demoproc" );
        throw new IllegalAccessException( "demo" );
        }
    }

    public static void main( String args[] ) {
        demoproc();
    }
```

El error de compilación que se produce es lo suficientemente explícito:

```
% javac java906.java
java906.java:30:  Exception  java.lang.IllegalAccessException  must  be
caught, or it must be declared in the throws clause of this method.
        throw new IllegalAccessException( "demo" );
          ^
```

Para hacer que este código compile, se convierte en el ejemplo siguiente, , en donde se declara que el método puede lanzar una excepción de acceso ilegal, con lo que el problema asciende un nivel más en la jerarquía de llamadas. Ahora main() llama a demoproc(), que se ha declarado que lanza una IllegalAccessException, por lo tanto colocamos un bloque try que pueda capturar esa excepción.

```
class java907 {
    static void demoproc() throws IllegalAccessException {
        System.out.println( "Dentro de demoproc" );
        throw new IllegalAccessException( "demo" );
        }

    public static void main( String args[] ) {
        try {
            demoproc();
        } catch( IllegalAccessException e ) {
            System.out.println( "Capturada de nuevo: " + e );
            }
        }
```

```
}
```

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque y sigue el flujo de control por el bloque finally (si lo hay) y concluye el control de la excepción.

Si ninguna de las cláusulas catch coincide con la excepción que se ha producido, entonces se ejecutará el código de la cláusula finally (en caso de que la haya). Lo que ocurre en este caso, es exactamente lo mismo que si la sentencia que lanza la excepción no se encontrase encerrada en el bloque try.

El flujo de control abandona este método y retorna prematuramente al método que lo llamó. Si la llamada estaba dentro del ámbito de una sentencia try, entonces se vuelve a intentar el control de la excepción, y así continuamente.

Cuando una excepción no es tratada en la rutina en donde se produce, lo que sucede es lo siguiente. El sistema Java busca un bloque try..catch más allá de la llamada, pero dentro del método que lo trajo aquí. Si la excepción se propaga de todas formas hasta lo alto de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje. Es decir, podemos suponer que Java nos está proporcionando un bloque catch por defecto, que imprime un mensaje de error, indica las últimas entradas en la pila de llamadas y sale.

No hay ninguna sobrecarga en el sistema por incorporar sentencias try al código. La sobrecarga se produce cuando se genera la excepción.

Se ha indicado ya que un método debe capturar las excepciones que genera, o en todo caso, declararlas como parte de su llamada, indicando a todo el mundo que es capaz de generar excepciones. Esto debe ser así para que cualquiera que escriba una llamada a ese método esté avisado de que le puede llegar una excepción, en lugar del valor de retorno normal. Esto permite al programador que llama a ese método, elegir entre controlar la excepción o propagarla hacia arriba en la pila de llamadas. La siguiente línea de código muestra la forma general en que un método declara excepciones que se pueden propagar fuera de él, tal como se ha visto a la hora de tratar la sentencia throws:

```
tipo_de_retorno( parametros ) throws e1,e2,e3 { }
```

Los nombres e1,e2,... deben ser nombres de excepciones, es decir, cualquier tipo que sea asignable al tipo predefinido Throwable. Observar que, como en la llamada al método se especifica el tipo de retorno, se está especificando el tipo de excepción que puede generar (en lugar de un objeto Exception).

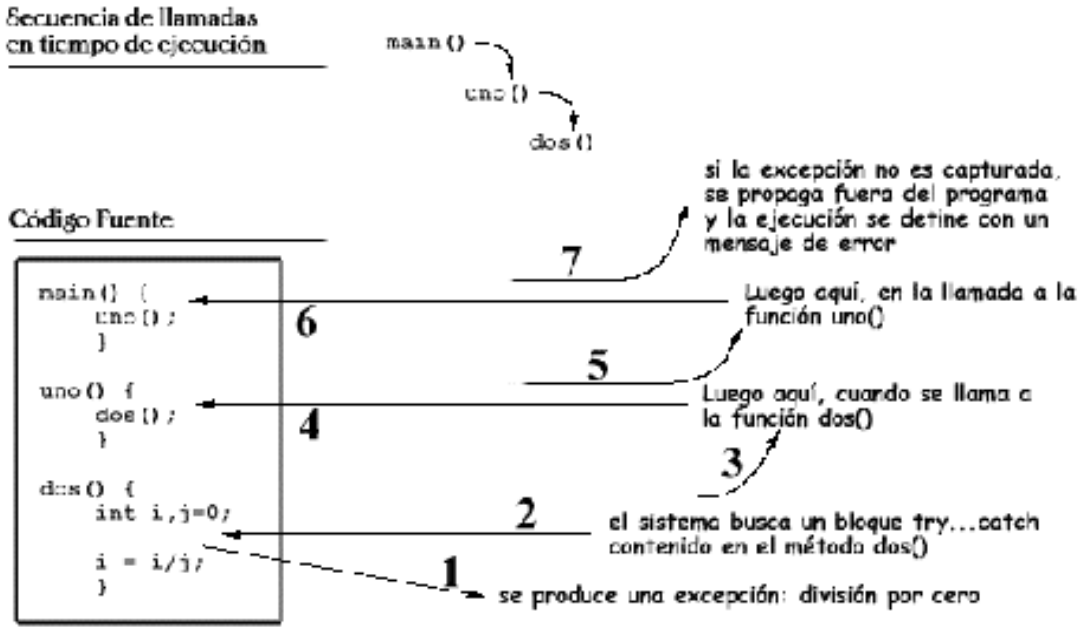
He aquí un ejemplo, tomado del sistema Java de entrada/salida:

```
byte readByte() throws IOException;
short readShort() throws IOException;
char readChar() throws IOException;

void writeByte( int v ) throws IOException;
void writeShort( int v ) throws IOException;
void writeChar( int v ) throws IOException;
```

Lo más interesante aquí es que la rutina que lee un char, puede devolver un char; no el entero que se requiere en C. C necesita que se devuelva un int, para poder pasar cualquier valor a un char, y además un valor extra (-1) para indicar que se ha alcanzado el

final del fichero. Algunas de las rutinas Java lanzan una excepción cuando se alcanza el fin del fichero.

En el siguiente diagrama se muestra gráficamente cómo se propaga la excepción que se genera en el código, a través de la pila de llamadas durante la ejecución del código:



Cuando se crea una nueva excepción, derivando de una clase Exception ya existente, se puede cambiar el mensaje que lleva asociado. La cadena de texto puede ser recuperada a través de un método. Normalmente, el texto del mensaje proporcionará información para resolver el problema o sugerirá una acción alternativa. Por ejemplo:

```
class SinGasolina extends Exception {
    SinGasolina( String s ) {    // constructor
        super( s );
        }
    ....

// Cuando se use, aparecerá algo como esto
    try {
        if( j < 1 )
            throw new SinGasolina( "Usando deposito de reserva" );
    } catch( SinGasolina e ) {
        System.out.println( o.getMessage() );
        }
```

Esto, en tiempo de ejecución originaría la siguiente salida por pantalla:

```
> Usando deposito de reserva
```

Otro método que es heredado de la superclase Throwable es **printStackTrace**(). Invocando a este método sobre una excepción se volcará a pantalla todas las llamadas hasta el momento en donde se generó la excepción (no donde se maneje la excepción). Por ejemplo:

```
// Capturando una excepción en un método
class testcap {
    static int slice0[] = { 0,1,2,3,4 };

    public static void main( String a[] ) {
        try {
            uno();
        } catch( Exception e ) {
            System.out.println( "Captura de la excepcion en main()" );
            e.printStackTrace();
            }
        }

    static void uno() {
        try {
            slice0[-1] = 4;
        } catch( NullPointerException e ) {
            System.out.println( "Captura una excepcion diferente" );
            }
        }
    }
```

Cuando se ejecute ese código, en pantalla observaremos la siguiente salida:

```
> Captura de la excepcion en main()
> java.lang.ArrayIndexOutOfBoundsException: -1
        at testcap.uno(test5p.java:19)
        at testcap.main(test5p.java:9)
```

Con todo el manejo de excepciones podemos concluir que se proporciona un método más seguro para el control de errores, además de representar una excelente herramienta para organizar en sitios concretos todo el manejo de los errores y, además, que se pueden proporcionar mensajes de error más decentes al usuario indicando qué es lo que ha fallado y por qué, e incluso podemos, a veces, recuperar al programa automáticamente de los errores.

La degradación que se produce en la ejecución de programas con manejo de excepciones está ampliamente compensada por las ventajas que representa en cuanto a seguridad de funcionamiento de esos mismos programas.

MODELO DE EVENTOS

Revisión del Modelo de Propagación

El modelo de gestión de eventos del AWT está basado en la Herencia. Para que un programa capture eventos de un interfaz, los Componentes deben ser subclases del interfaz y sobreescribir los métodos **action**() y **handleEvent**().

Cuando uno de los dos métodos anteriores devuelve true, el evento ya no es procesado más allá, en caso contrario, el evento se propaga a través de la jerarquía de componentes del interfaz hasta que el evento sea tratado o alcance la raíz de la jerarquía. El resultado de este modelo es que los programas tienen dos elecciones para estructurar su código de manejo de eventos:

- Cada componente individual puede hacerse subclase para manejar específicamente un conjunto de eventos
- Todos los eventos para una jerarquía completa (o subconjunto de ella) pueden ser manejados por un contenedor determinado

En este modelo de Herencia, no hay posibilidad de filtrar eventos. Los eventos son recibidos por los Componentes, independientemente de que los manejen o no. Este es un problema general de rendimiento, especialmente con eventos que se producen con mucha frecuencia, como son los eventos de ratón. Con el nuevo modelo, todos los sistemas debería ver incrementado su rendimiento, especialmente los sistemas basados en Solaris.

Modelo de delegación de eventos

De acuerdo con Javasoft, las principales características de partida que han originado el nuevo modelo de manejo de eventos en el AWT, son:

- Que sea simple y fácil de aprender
- Que soporte una clara separación entre el código de la aplicación y el código del interfaz
- Que facilite la creación de robustos controladores de eventos, con menos posibilidad de generación de errores (chequeo más potente en tiempo de compilación)
- Suficientemente flexible para permitir el flujo y propagación de eventos
- Para herramientas visuales, permitir en tiempo de ejecución ver cómo se generan estos eventos y quien lo hace
- Que soporte compatibilidad binaria con el modelo anterior

A continuación se muestra una revisión por encima del nuevo modelo de eventos, antes de entrar en el estudio detallado, y también se expone un ejemplo sencillito para poder entender de forma más fácil ese estudio detallado del modelo de Delegación de Eventos.

Los eventos ahora están organizados en jerarquías de clases de eventos.

El nuevo modelo hace uso de fuentes de eventos (Source) y receptores de eventos (Listener). Una **fuente de eventos** es un objeto que tiene la capacidad de detectar eventos y notificar a los receptores de eventos que se han producido esos eventos. Aunque el programador puede establecer el entorno en que se producen esas notificaciones, siempre hay un escenario por defecto.

Un objeto **receptor de eventos** es una clase (o una subclase de una clase) que implementa un interfaz receptor específico. Hay definidos un determinado número de interfaces receptores, donde cada interfaz declara los métodos adecuados al tratamiento de los eventos de su clase. Luego, hay un emparejamiento natural entre clases de eventos y definiciones de interfaces. Por ejemplo, hay una clase de eventos de ratón que incluye muchos de los eventos asociados con las acciones del ratón, y hay un interfaz que se utiliza para definir los receptores de esos eventos.

Un objeto receptor puede estar registrado con un objeto fuente para ser notificado de la ocurrencia de todos los eventos de la clase para los que el objeto receptor está diseñado. Una vez que el objeto receptor está registrado para ser notificado de esos eventos, el suceso de un evento en esta clase automáticamente invocará al método sobreescrito del objeto receptor. El código en el método sobreescrito debe estar diseñado por el programador para realizar las acciones específicas que desee cuando suceda el evento.

Algunas clases de eventos, como los de ratón, involucran a un determinado conjunto de eventos diferentes. Una clase receptor que implemente el interfaz que recoja estos eventos debe sobreescribir todos los métodos declarados en el interfaz. Para prevenir esto, de forma que no sea tan tedioso y no haya que sobreescribir métodos que no se van a utilizar, se han definido un conjunto de clases intermedias, conocidas como Adaptadores (Adapter).

Estas clases **Adaptadores** implementan los interfaces receptor y sobreescriben todos los métodos del interfaz con métodos vacíos. Una clase receptor puede estar definida como clase que extiende una clase Adapter en lugar de una clase que implemente el interfaz. Cuando se hace esto, la clase receptor solamente necesita sobreescribir aquellos métodos que sean de interés para la aplicación, porque todos los otros métodos serán resueltos por la clase Adapter. Por ejemplo, en el programa Java1101, los dos objetos receptor instanciados desde dos clases diferentes, están registrados para recibir todos los eventos involucrados en la manipulación de un objeto de tipo Frame (apertura, cierre, minimización, etc.).

```
/**
* Este ejemplo muestra el uso de fuentes de eventos, receptores de
* eventos y adaptadores en el modelo de Delegacion de Eventos
* introducido  por Sun en el JDK 1.1
* La aplicacion instacia un objeto que crea un interfaz de usuario que
* consta de un Frame. Este objeto es la Fuente de Eventos que notifica
* a dos Receptores diferentes los eventos que se producen en la ventana.
* Uno de los receptores de eventos implementa el interfaz WindowListener
* y define todos los metodos declarados en ese interfaz.
* El otro objeto receptor extiende la clase adaptadora WindowAdapter, y
* solamente sobreescribe dos de los metodos, ya que la clase Adapter
* sobreescribe los restantes con metodos vacios.
* La aplicacion no termina por si sola, hay que forzarla a concluir.*/
```

```
import java.awt.*;
import java.awt.event.*;
public class java1101 {
   public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario, para que permita instanciar a su vez dos objetos Listener
// y registrarlos para que reciban notificacion cuando se producen
// eventos en una Ventana
class IHM {
   // Constructor de la clase
   public IHM() {
        // Se crea un objeto Frame
        Frame ventana = new Frame();
        // El metodo setSize() reemplaza al metodo resize() del JDK 1.0
        ventana.setSize( 300,200 );
        ventana.setTitle( "Tutorial de Java, Eventos" );
        // El metodo setVisible() reemplaza al metodo show() del JDK 1.0
        ventana.setVisible( true );
        // Se instancian dos objetos receptores que procesaran los
        // eventos de la ventana
        Proceso1 ventanaProceso1 = new Proceso1( ventana );
        Proceso2 ventanaProceso2 = new Proceso2();
        // Se registran los dos objetos receptores para que sean
        // notificados de los evetnos que genere la ventana, que es el
        // objeto origen de los eventos
        ventana.addWindowListener( ventanaProceso1 );
        ventana.addWindowListener( ventanaProceso2 );
        }
   }
// Las dos clases siguientes se pueden utilizar para instanciar los
// objetos receptor. Esta clase implementa el interfaz WindowListener,
// lo cual requiere que todos los metodos que estan declarados en el
// interfaz sean definidos en la clase.
// La clase define todos esos metodos y presenta un mensaje
// descriptivo cada vez que se invoca a uno de ellos.
class Proceso1 implements WindowListener {
   // Variable utilizada para guardar una referencia al objeto Frame
   Frame ventanaRef;
   // Constructor que guarda la referencia al objeto Frame
   Proceso1( Frame vent ){
        this.ventanaRef = vent;
        }
   public void windowClosed( WindowEvent evt ) {
        System.out.println( "Metodo windowClosed de Proceso1" );
        }
   public void windowIconified( WindowEvent evt ) {
        System.out.println( "Metodo windowIconified de Proceso1" );
        }
   public void windowOpened( WindowEvent evt ) {
        System.out.println( "Metodo windowOpened de Proceso1" );
        }
   public void windowClosing( WindowEvent evt ) {
        System.out.println( "Metodo windowClosing de Proceso1" );
        // Se oculta la ventana
        ventanaRef.setVisible( false );
        }
   public void windowDeiconified( WindowEvent evt ) {
        System.out.println( "Metodo windowDeiconified Proceso1" );
        }
```

```
   public void windowActivated( WindowEvent evt ) {
       System.out.println( "Metodo windowActivated de Proceso1" );
       }
   public void windowDeactivated( WindowEvent evt ) {
       System.out.println( "Metodo windowDeactivated de Proceso1" );
       }
   }
// Esta clase y la anterior se pueden utilizar para instanciar
// objetos Listener. En esta clase, se extiende la clase Adapter
// obvienado el requerimiento de tener que definir todos los
// metodos del receptor de eventos WindowListener. El objeto
// Adapter, WindowAdapter extiende a WindowListener y define
// todos los metodos con codigo vacio, que pueden ser sobreescritos
// siempre que se desee. En este clase concreta, solamente se
// sobreescriben dos de los metodos declarados en el interfaz, y
// presenta un mensaje cada vez que se invoca a uno de ellos
class Proceso2 extends WindowAdapter {
   public void windowIconified( WindowEvent evt ) {
       System.out.println( "--- Metodo windowIconified de Proceso2" );
       }
   public void windowDeiconified( WindowEvent evt ) {
       System.out.println( "---Metodo windowDeiconified de Proceso2" );
       }
   }
```

Uno de los objetos receptor implementa el interfaz WindowListener, por lo que debe sobreescribir los seis métodos del interfaz. La otra clase receptor extiende la clase WindowAdapter en vez de implementar el interfaz WindowListener. La clase WindowAdapter sobrescribe los seis métodos del interfaz con métodos vacíos, por lo que la clase receptor no necesita sobreescribir esos seis métodos.

El ejemplo se ha hecho lo más simple posible, de forma que los métodos sobreescritos solamente presentan un mensaje en pantalla indicando cuando han sido invocados.

Una de las cosas en las que debe reparar el lector es en la forma en que los objetos receptores son registrados para la notificación de los eventos; lo cual se hace en las sentencias que se reproducen de nuevo a continuación:

```
    // Se registran los dos objetos receptores para que sean
    // notificados de los eventos que genere la ventana, que es el
    // objeto origen de los eventos
    ventana.addWindowListener( ventanaProceso1 );
    ventana.addWindowListener( ventanaProceso2 );
```

La interpretación de este fragmento de código es que dos objetos receptores llamados ventanaProceso1 y ventanaProceso2 se añaden a la lista de objetos receptores que serán automáticamente notificados cuando se produzca un evento de la clase Window con respecto al objeto Frame llamado ventana.

Estos objetos receptores son notificados invocando los métodos sobrescritos de los objetos que recogen el tipo específico de evento(apertura de la ventana, cierre de la ventana, minimización de la ventana, etc.).

Los párrafos que siguen introducirán al lector en una visión más detallada del modelos de Delegación de Eventos.

En el JDK 1.0.2 los eventos se encapsulan en una sola clase llamada Event, y desde el JDK 1.1 los eventos se encapsulan en una jerarquía de clases donde la clase raíz es

java.util.EventObject; en el JDK 1.2 bajo esta misma clase se encapsulan las clases de los eventos de los APIs que se han incorporado como son **BeanContextEvent,** DragSourceEvent, DropTargetEvent y PropertyChangeEvent. La propagación de un evento desde un objeto Fuente hasta un objeto Receptor involucra la llamada a un método en el objeto receptor y el paso de una instancia de una subclase de eventos (un objeto) que define el tipo de evento generado. Cada subclase de eventos puede incluir más de un tipo de eventos.

Un objeto receptor es una instancia de una clase que implementa un interfaz específico EventListener extendido desde el receptor genérico java.util.EventListener. Un interfaz EventListener define uno o más métodos que deben ser invocados por la fuente de eventos en respuesta a cada tipo específico de evento controlado por el interfaz.

La invocación de estos métodos redefinidos es el mecanismo por el cual el objeto Fuente notifica al Receptor que uno o más eventos han sucedido. El objeto fuente mantiene una lista de objetos receptores y los tipos de eventos a los que están suscritos. El programador crea esa lista utilizando llamadas a los métodos add<TipoEvento>Listener().

Una vez que la lista de receptores está creada, el objeto fuente utiliza esta lista para notificar a cada receptor que ha sucedido un evento del tipo que controla, sin esfuerzo alguno por parte del programador. Esto es lo que se conoce como registrar Receptores específicos para recibir la notificación de eventos determinados.

La fuente de eventos es generalmente un componente del Interfaz Gráfico, por lo que se van a dejar de lado, por ahora, los eventos generados por los APIs incorporados al JDK 1.2 y centrar el estudio en los eventos que envían los componentes del AWT. Un receptor de eventos es normalmente un objeto de una clase que implementa el adecuado interfaz del receptor. El objeto receptor también puede ser otro componente del AWT que implementa uno o más interfaces receptor, con el propósito de comunicar unos objetos del interfaz gráfico con otros.

Como ya se ha mencionado, los eventos en el nuevo modelo de Delegación no están representados por una sola clase Event con identificaciones numéricas, como en el JDK 1.0.2; sino que cada tipo específico de evento es un miembro de la clase de tipos de eventos y estas clases constituyen una completa jerarquía de clases.

Como una sola clase de evento se puede utilizar para representar más de un tipo de evento, algunas clases pueden contar con un identificador (único para cada clase) que designa a cada uno de los eventos específicos; por ejemplo, MouseEvent representa el movimiento del cursor, la pulsación de un botón, el arrastre del ratón, etc.

No hay campos públicos en las nuevas clases. En su lugar, los datos del evento están encapsulados y solamente se puede acceder a ellos a través de los adecuados métodos set..() y get..(); los primeros sólo existen para modificar atributos de un evento y sólo pueden ser utilizados por un receptor.

El AWT define un conjunto determinado de eventos, aunque el programador también puede definir sus propios tipos de eventos, derivando de EventObject, o desde una de las clases de eventos del AWT.

El AWT proporciona dos tipos conceptuales de eventos: de bajo nivel y semánticos, que se introducen a continuación, aunque posteriormente se verán con detalle.

Un evento de bajo nivel es un evento que representa una entrada de bajo nivel o un suceso sobre un componente visual de un sistema de ventanas sobre la pantalla. Eventos de este tipo son:

```
java.util.EventObject
java.awt.AWTEvent
java.awt.event.ComponentEvent   //  Componente redimensionado, desplazado
java.awt.event.FocusEvent       // Pérdida, ganancia del focus por un Componente
java.awt.event.InputEvent
java.awt.event.KeyEvent    //   El Componente recoge una pulsación de teclado
java.awt.event.MouseEvent   //  El Componente recoge movimientos del ratón,  pulsación de botones
java.awt.event.ContainerEvent
java.awt.event.WindowEvent
```

Como ya se ha indicado, algunas clases de eventos engloban a varios tipos distintos de eventos. Normalmente, hay un interfaz correspondiente a cada clase de evento y hay métodos del interfaz para cada tipo distinto de evento en cada clase de evento.

Un evento semántico es un evento que se define a alto nivel y encapsula una acción de un componente del interfaz de usuario. Algunos eventos de este tipo son:

```
java.util.EventObject
    java.awt.AWTEvent
        java.awt.event.ActionEvent       // Ejecución de un comando
        java.awt.event.AdjustmentEvent  //  Ajuste de un valor
        java.awt.event.ItemEvent         //Cambio de estado de un item
        java.awt.event.TextEvent        // Cambio de valor de un texto
```

Estos eventos no están pensados para atender a Componentes específicos de la pantalla, sino para aplicarlos a un conjunto de eventos que implementen un modelo semántico similar. Por ejemplo, un objeto Button generará un evento action cuando sea pulsado y un objeto List generará un evento action cuando se pulse dos veces con el ratón sobre uno de los items que componen la lista.

Receptores de eventos

Un interfaz EventListener tendrá un método específico para cada tipo de evento distinto que trate la clase de evento. Por ejemplo, el interfaz FocusEventListener define los métodos focusGained() y focusLost(), uno para cada tipo de evento que trata la clase FocusEvent.

Los interfaces de bajo nivel que define la versión del JDK 1.2 son los siguientes:

```
java.util.EventListener
    java.awt.event.ComponentListener
    java.awt.event.ContainerListener
    java.awt.event.FocusListener
    java.awt.event.KeyListener
    java.awt.event.MouseListener
    java.awt.event.MouseMotionListener
    java.awt.event.WindowListener
```

Si se compara esta lista con la lista anterior de clases de eventos de bajo nivel, se ve claramente que hay definido un interfaz receptor por cada una de las clases más bajas en

jerarquía de las clases de eventos, excepto para la clase MouseEvent en que hay dos interfaces receptores diferentes.

Los interfaces de nivel semántico que define el AWT en la versión del JDK 1.2 son:

```
java.util.EventListener
    java.awt.event.ActionListener
    java.awt.event.AdjustmentListener
    java.awt.event.ItemListener
    java.awt.event.TextListener
```

La correspondencia aquí entre interfaces de nivel semántico y clases evento de nivel semántico es uno a uno.

Como los receptores se registran para manejar tipos de eventos determinados, solamente serán notificados de esos tipos de eventos y no llegarán a ellos otros tipos de eventos para los que no están registrados. Esto es justamente lo contrario a lo que se utilizaba en el modelo de Propagación, en donde todos los eventos se pasaban a un controlador de eventos, fuesen de su interés o no. Este filtrado de eventos mejora el rendimiento, especialmente con los eventos que se producen con mucha frecuencia, como son los de movimiento del ratón.

Fuentes de eventos

Todas las fuentes de eventos del AWT soportan el multienvío a receptores. Esto significa que se pueden añadir o quitar múltiples receptores de una sola fuente; en otras palabras, la notificación de que se ha producido un mismo evento se puede enviar a uno o más objetos receptores simultáneamente.

El API de Java no garantiza el orden en que se enviarán los eventos a los receptores que están registrados en un objeto fuente, para ser informados de esos eventos. En caso de que el orden en que se distribuyen los eventos sea un factor importante en el programa, se deberían encadenar los receptores de un solo objeto receptor registrado sobre el objeto fuente.

Como en el caso de los receptores, se puede hacer una distinción entre los eventos de bajo nivel y los eventos de tipo semántico. Las fuentes de eventos de bajo nivel serán las clases de elementos o componentes visuales del interfaz gráfico (botones, barras de desplazamiento, cajas de selección, etc.), porque cada componente de la pantalla generará sus eventos específicos. El JDK 1.2 permite registrar receptores sobre fuentes de eventos de los siguientes tipos:

```
java.awt.Component
  addComponentListener
  addFocusListener
  addKeyListener
  addMouseListener
  addMouseMotionListener
java.awt.Container
  addContainerListener
java.awt.Dialog
  addWindowListener
java.awt.Frame
  addWindowListener
```

Para determinar todos los tipos de eventos que se pueden comunicar desde un objeto fuente a un receptor, hay que tener en cuenta la herencia. Por ejemplo, como se verá en uno de los programas que se presentarán, un objeto puede detectar eventos del ratón sobre un objeto Frame y notificar a un objeto MouseListener de la ocurrencia de estos eventos, aunque en la lista anterior no se muestre un MouseListener sobre un Frame. Esto es posible porque un objeto Frame extiende indirectamente la clase Component y, MouseListener está definido en la clase Component.

Los receptores de eventos que se pueden registrar de tipo semántico sobre objetos fuentes, generadores de eventos, en el JDK 1.2 son:

```
java.awt.Button
  addActionListener
java.awt.Choice
  addItemListener
java.awt.Checkbox
  addItemListener
java.awt.CheckboxMenuItem
  addItemListener
java.awt.List
  addActionListener
  addItemListener
java.awt.MenuItem
  addActionListener
java.awt.Scrollbar
  addAdjustmentListener
java.awt.TextArea
  addTextListener
java.awt.TextField
  addActionListener
   addTextListener
```

Adaptadores

Muchos interfaces EventListener están diseñados para recibir múltiples clases de eventos, por ejemplo, el interfaz MouseListener puede recibir eventos de pulsación de botón, al soltar el botón, a la recepción del cursor, etc. El interfaz declara un método para cada uno de estos subtipos. Cuando se implementa un interfaz, es necesario redefinir todos los métodos que se declaran en ese interfaz, incluso aunque se haga con métodos vacíos. En la mayoría de las ocasiones, no es necesario redefinir todos los métodos declarados en el interfaz porque no son útiles para la aplicación.

Por ello, el AWT proporciona un conjunto de clases abstractas adaptadores (Adapter) que coinciden con los interfaces. Cada clase adaptador implementa un interfaz y redefine todos los métodos declarados por el interfaz con métodos vacíos, con lo cual se satisface ya el requerimiento de la redefinición de todos los métodos.

Se pueden definir clases Receptor extendiendo clases adaptadores, en vez de implementar el interfaz receptor correspondiente. Esto proporciona libertad al programador para redefinir solamente aquellos métodos del interfaz que intervienen en la aplicación que desarrolla.

De nuevo, hay que recordar que todos los métodos declarados en un interfaz corresponden a los tipos de eventos individuales de la clase de eventos correspondiente, y que el objeto Fuente notifica al Receptor la ocurrencia de un evento de un tipo determinado invocando al método redefinido del interfaz.

Las clases Adaptadores que se definen en el JDK 1.2 son las que se indican a continuación:

```
java.awt.ComponentAdapter
java.awt.FocusAdapter
java.awt.KeyAdapter
java.awt.MouseAdapter
java.awt.MouseMotionAdapter
java.awt.WindowAdapter
```

En el ejemplo Java1102, se modifica el primer programa de este capítulo, en que la ejecución no terminaba cuando se cerraba la ventana; ahora el programa termina cuando el usuario cierra la ventana, ejecutando la sentencia de salida en el controlador de eventos adecuado.

```
/**
* Este programa instancia un objteo receptor para porcesar los eventos
* generados por el raton. Cuando se pulsa uno de los botones del raton
* sobre el objeto Frame, el programa recoge las coordenadas de la
* posicion en que se encontraba el cursor y las presenta en pantalla
* cerca del punto en que se ha producido el click.
* La coordenada Y devuelta por getY() parece ser 30 pixels menor que
* la real
*/

import java.awt.*;
import java.awt.event.*;
public class java1102 {
   public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
   }
// Se crea una subclase de Frame para poder sobreescribir el metodo
// paint(), y presentar en pantalla las coordenadas donde se haya
// producido el click del raton
class MiFrame extends Frame {
   int ratonX;
   int ratonY;
   public void paint( Graphics g ) {
        g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
        }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
   public IHM() {
        MiFrame ventana = new MiFrame();
        ventana.setSize( 300,300 );
        ventana.setTitle( "Tutorial de Java, Eventos" );
        ventana.setVisible( true );
       // Se instancia y registra un objeto receptor de eventos
       // para terminar la ejecucion del programa cuando el
       // usuario decida cerrar la ventana
        Proceso1 procesoVentana1 = new Proceso1();
        ventana.addWindowListener( procesoVentana1 );
       // Se instancia y registra un objeto receptor de eventos
       // que sera el encargado de procesas los eventos del raton
       // para determinar y presentar las coordenadas en las que
       // se encuentra el cursor cuando el usuario pulsa el boton
       // del raton
        ProcesoRaton procesoRaton = new ProcesoRaton( ventana );
        ventana.addMouseListener( procesoRaton );
        }
```

```
    }
// Esta clase Receptora monitoriza las pulsaciones de los botones
// del raton y presenta las coordenadas en las que se ha producido
// el click
// Se trata de una clase Adpater, luego solo se redefinen los metodos
// que resulten utiles para el objetivo de la aplicacion
class ProcesoRaton extends MouseAdapter {
 MiFrame ventanaRef; // Referencia a la ventana
    // Constructor
   ProcesoRaton( MiFrame ventana ) {
        // Guardamos una referencia a la ventana
        ventanaRef = ventana;
        }
   // Se sobreescribe el metodo mousePressed para determinar y
   // presentar en pantalla las coordenadas del cursor cuando
   // se pulsa el raton
   public void mousePressed( MouseEvent evt ) {
        // Recoge las coordenadas X e Y de la posicion del cursor
        // y las almacena en el objeto Frame
        ventanaRef.ratonX = evt.getX();
        ventanaRef.ratonY = evt.getY();
        // Finalmente, presenta los valores de las coordenadas
        ventanaRef.repaint();
        }
   }
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Proceso1 extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
   }
```

El programa implementa un objeto EventSource que notifica a un objeto Listener la ocurrencia de un evento en la clase Window, y notifica a otro objeto Listener la ocurrencia de un evento en la clase Mouse.

Si se compila y ejecuta el ejemplo, cada vez que se pulse el botón del ratón con el cursor dentro de la ventana, aparecerán las coordenadas en las que se encuentra el cursor, tal como muestra la figura anterior.

En el caso más simple, los eventos de bajo nivel del nuevo modelo de Delegación de Eventos, se pueden controlas siguiendo los pasos que se indican a continuación:

- Definir una clase Listener, receptor, para una determinada clase de evento que implemente el interfaz receptor que coincida con la clase de evento, o extender la clase adaptadora correpondiente
- Redefinir los métodos del interfaz receptor para cada tipo de evento específico de la clase evento, para poder implementar la respuesta deseada del programa ante la ocurrencia de un evento. Si se implementa el interfaz receptor, hay que redefinir todos los métodos del interfaz. Si se extiende la clase adaptadora, se pueden redefinir solamente aquellos métodos que son de interés
- Definir una clase Source, fuente, que instancie un objeto de la clase receptor y registrarla para la notificación de la ocurrencia de eventos generados por cada componente específico

Por ejemplo, esto se consigue utilizando código como

```
objetoVentana.addMouseListener( procesoRaton );
```

en donde

objetoVentana es el objeto que genera el evento

procesoRaton es el nombre del objeto receptor del evento, y

addMouseListener es el método que registra el objeto receptor para recibir eventos de ratón desde el objeto llamado objetoVentana

Esta sentencia hará que el objeto procesoRaton sea notificado de todos los eventos que se produzcan sobre el objetoVentana que sean parte de la clase de eventos del ratón. La notificación tendrá lugar invocando al método redefinido en el objeto procesoRaton que corresponda con cada tipo específico de evento en la clase de eventos de ratón, aunque algunos de estos métodos pueden estar vacíos porque no interese tratar los eventos a que corresponden.

Todo lo anterior en el caso más simple, porque es posible complicar la situación a gusto, por ejemplo, si se quiere notificar a dos objetos receptor diferentes de la ocurrencia de un determinado evento sobre un mismo objeto de la pantalla, tal como muestra el código del ejemplo Java1103, en donde un objeto receptor es compartido por dos componentes visuales diferentes del mismo tipo. El programa detecta los eventos de ratón sobre dos objetos Frame diferentes, que distingue en base a su nombre, y presenta las coordenadas del cursor en cada pulsación sobre el objeto en que se encontraba el cursor.

```
/**
* Este ejemplo muestra como se comparte un solo objeto receptor entre dos
* componentes visuales del mismo tipo. El programa detecta los eventos
* del raton que se producen en cualquierea de los dos objetos Frame.
* Distingue entre un objeto y otro, en funcion del nombre del componente,
* y presenta las coordenadas en que se encuentra el cursor, en el
* objeto en que se haya picado
*
* NOTA:
* Cuando se arranca el programa, los dos componenetes visuales se
encuentran
* superpuestos, por lo que sera necesario mover uno respecto del otro
para
* poder acceder a los dos con el raton
```

```
*/

import java.awt.*;
import java.awt.event.*;
public class java1103 {
   public static void main( String args[] ) {
       // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
       IHM ihm = new IHM();
       }
   }
// Subclase de Frame para poder sobreescribir el metodo paint()
class MiFrame extends Frame {
   int ratonX;
   int ratonY;
   MiFrame( String nombre ) {
       setTitle( "Tutorial de Java, Eventos" );
       setSize( 300,200 );
       // Se asigna un nombre para distinguir entre los dos objetos
       setName( nombre );
       }
    public void paint( Graphics g ) {
       // Presenta en pantalla las coordenadas del cursos
       g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
       }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
   public IHM() {
       // Se crean dos objetos visuales de tipo Frame, se le hace
       // visibles y se les asignan los nombres Frame1 y Frame2
       MiFrame miFrame1 = new MiFrame( "Frame1" );
       miFrame1.setVisible( true );
       MiFrame miFrame2 = new MiFrame( "Frame2" );
       miFrame2.setVisible( true );
       // Se instancia y registra un objeto Receptor que hara que
       // concluya la ejecucion del programa cuando el usuario cierre
       // la ventana
       Proceso1 procesoVentana1 = new Proceso1();
       miFrame1.addWindowListener( procesoVentana1 );
       miFrame2.addWindowListener( procesoVentana1 );
       // Se instancia y registra un objeto Receptor que procesara los
       // eventos del raton que se generen en el objeto MiFrame
       ProcesoRaton procesoRaton= new ProcesoRaton( miFrame1,miFrame2 );
       miFrame1.addMouseListener( procesoRaton );
       miFrame2.addMouseListener( procesoRaton );
       }
   }
// Esta clase Receptor, monitoriza los eventos de pulsacion del
// raton y presenta en pantalla las coordenadas en donde se
// encuentra el cursor cuando el raton es pulsado. El objeto
// receptor distingue entre los dos objetos visuales en base a
// sus nombres como componentes y presenta las coordenadas sobre
// el objeto visual que ha generado el evento del raton
class ProcesoRaton extends MouseAdapter{
   // Variables para guardar referencias a los objetos
   MiFrame frameRef1,frameRef2;
   // Constructor
   ProcesoRaton( MiFrame frame1,MiFrame frame2 ) {
       frameRef1 = frame1;
       frameRef2 = frame2;
       }
   // Se sobrescribe el metodo mousePressed() para controlar la
   // respuesta cuando el raton se pulse sobre uno de los dos
   // objetos Frame
```

```
    public void mousePressed( MouseEvent evt ) {
        if( evt.getComponent().getName().compareTo( "Frame1" ) == 0 ) {
            // Recoge las coordenadas X e Y de la posicion del cursor
            // y las almacena en el objeto Frame
            frameRef1.ratonX = evt.getX();
            frameRef1.ratonY = evt.getY();
            frameRef1.repaint();
            }
        else {
            // Recoge las coordenadas X e Y de la posicion del cursor
            // y las almacena en el objeto Frame
            frameRef2.ratonX = evt.getX();
            frameRef2.ratonY = evt.getY();
            frameRef2.repaint();
            }
        }
    }
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

El código del ejemplo es realmente simple, la única parte críptica es la que trata de obtener el nombre del componente visual que ha generado el evento mousePressed().

El método main() instancia un objeto de tipo IHM, que sirve para dos propósitos, por un lado proporcionar el interfaz visual y, por otro, actuar como una fuente de eventos que notificará su ocurrencia a los objetos receptor que se registren con él.

La clase Frame es extendida en una nueva clase llamada MiFrame, para permitir la redefinición del método paint() de la clase; lo cual es imprescindible para presentar las coordenadas en donde se encuentra el cursor sobre el Frame, utilizando el método drawString().

El constructor de la clase IHM instancia dos objetos de tipo MiFrame y los hace visibles. Cuando son instanciados, se les asignan los nombre Frame1 y Frame2 a través del método setName(). Estos nombres serán los que permitan determinar posteriormente cuál ha sido el objeto que ha generado el evento.

También en ese mismo constructor se instancia a un solo objeto receptor a través del cual se procesarán todos los eventos de bajo nivel que se produzcan en cualquiera de los dos objetos visuales:

```
ProcesoRaton procesoRaton = new ProcesoRaton( miFrame1,miFrame2 );
miFrame1.addMouseListener( procesoRaton );
miFrame2.addMouseListener( procesoRaton );
```

La primera sentencia solamente instancia el nuevo objeto receptor procesoRaton, pasándole las referencias de los dos elementos visuales como parámetro. Las dos sentencias siguientes incorporan este objeto receptor (lo registran) a la lista de objetos receptor que serán automáticamente notificados cuando suceda cualquier evento de ratón sobre los objetos visuales referenciados como miFrame1 y miFrame2, respectivamente. Y ya no se requiere ningún código extra para que se produzca esa notificación.

Las notificaciones se realizan invocando métodos de instancia específicos redefinidos del objeto receptor ante la ocurrencia de tipos determinados de eventos del ratón. Las declaraciones de todos los métodos deben coincidir con todos los posibles eventos del ratón que estén definidos en el interfaz MouseListener, que deben coincidir con los definidos en la clase MouseEvent. La clase desde la que el objeto receptor es instanciado debe redefinir, bien directa o indirectamente, todos los métodos declarados en el interfaz MouseListener.

Además de registrar el objeto MouseListener para recibir objetos de ratón, el programa también instancia y registra un objeto receptor que monitoriza los eventos de la ventana, y termina la ejecución del programa cuando el usuario cierra uno cualquiera de los objetos visuales.

```
Proceso1 procesoVentana1 = new Proceso1();
miFrame1.addWindowListener( procesoVentana1 );
miFrame2.addWindowListener( procesoVentana1 );
```

En este caso, el código no intenta distinguir entre los dos objetos visuales.

La parte más complicada de la programación involucra al objeto receptor del ratón, y aún así, es bastante sencilla. Lo que intenta el código es determinar cuál de los dos elementos visuales ha sido el que ha generado el evento. En este caso el objeto receptor solamente a eventos mousePressed, aunque lo que se explica a continuación se podría aplicar a todos los eventos del ratón y, probablemente, a muchos de los eventos de bajo nivel.

La clase ProcesoRaton (receptor) en este programa extiende la clase MouseAdapter y redefine el método mousePressed() que está declarado en el interfaz MouseListener. Cuando es invocado el método mousePressed(), se le pasa un objeto de tipo MouseEvent, llamado evt, como parámetro.

Para determinar si el objeto que ha generado el evento ha sido Frame1, se utiliza la siguiente sentencia:

```
if( evt.getComponent().getName().compareTo( "Frame1" ) == 0 ) {
```

El método getComponent() es un método que devuelve el objeto donde se ha generado el evento. En este caso devuelve un objeto de tipo Component sobre el cual actúa el método getName(). Este último método devuelve el nombre del Componente como un objeto String, sobre el cual actúa el método compareTo(). Este método es estándar de la clase String y se utiliza para comparar dos objetos String. En este caso se utiliza para comparar el nombre del componente con la cadena "Frame1"; si es así, el código que se ejecuta es el que presenta las coordenadas del ratón sobre el objeto visual Frame1; si no coincide se ejecuta el código de la cláusula else que presentará las coordenadas sobre el objeto visual Frame2.

También es posible realizar la comparación directamente sobre el objeto MouseEvent, tal como se verá más adelante. Por ahora, baste hacer ver al lector que el paquete java.awt.event es diferente del paquete java.awt. El paquete java.awt.event ha sido añadido desde el JDK 1.1, aunque se mantiene parte de la documentación por compatibilidad con el JDK 1.0.2. Ha de tenerse cuidado pues a la hora de consultar la documentación, porque puede resultar altamente confuso al lector la consulta si mira la documentación del paquete java.awt cuando lo que necesita mirar se encuentra realmente en la documentación del paquete java.awt.event.

Aunque en el ejemplo anterior se utilizan dos objetos visuales del mismo tipo, no hay razón alguna para que eso sea así, ya que todos los objetos visuales comparten el mismo objeto receptor y son capaces de generar eventos para los que el receptor está registrado.

El ejemplo Java1104, es una modificación del programa anterior para utilizar un objeto Frame y un objeto Window, en vez de dos objetos Frame, para mostrar cómo un mismo objeto receptor puede recibir eventos de dos fuentes distintas.

```
/**
* El programa sirve muestra como un mismo objeto receptor puede
* ser compartido por dos componentes visuales de tipos distintos.
* El programa detecta los eventos del raton que se producen tanto
* sobre el objeto Frame como sobre el objeto Window. Distingue entre
* ellos en base al nombre de componente asignado y presenta en
* pantalla un texto indicado que objeto ha sido el que ha generado
* el evento.
* Cuando se cierre el objeto Frame, se provocara que concluya la
* ejecucion del programa.
*/
import java.awt.*;
import java.awt.event.*;
public class java1104 {
   public static void main( String args[] ) {
       // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
       IHM ihm = new IHM();
       }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
   public IHM() {
      // Se crea un objeto visual de tipo Frame y se le asigna el nombre
       Frame miFrame = new Frame();
       miFrame.setSize( 300,200 );
       miFrame.setTitle( "Tutorial de Java, Eventos" );
       miFrame.setName( "Frame" );
       miFrame.setVisible( true );
      // Se crea un objeto visual de tipo Window dentro del objeto Frame
      // y se le asigna el nombre
       Window miVentana = new Window( miFrame );
       miVentana.setSize( 100,100 );
       miVentana.setName( "Window" );
       miVentana.setVisible( true );
       // Se instancia y registra un objeto Receptor que procesara los
       // eventos del raton que se generen tanto en el objeto Frame
       // como en el objeto Window
       ProcesoRaton procesoRaton = new ProcesoRaton();
       miFrame.addMouseListener( procesoRaton );
       miVentana.addMouseListener( procesoRaton );
       // Se instancia y registra un objeto Receptor que hara que
       // concluya la ejecucion del programa cuando el usuario cierre
       // la ventana
       Proceso1 procesoVentana1 = new Proceso1();
       miFrame.addWindowListener( procesoVentana1 );
       }
   }
// Esta clase Receptor, monitoriza los eventos de pulsacion del
// raton y presenta en pantalla las coordenadas en donde se
// encuentra el cursor cuando el raton es pulsado, tanto sobre
// el objeto Frame como sobre el otro objeto, Window. El objeto
// receptor distingue entre los dos objetos visuales en base a
// sus nombres como componentes.
```

```
class ProcesoRaton extends MouseAdapter{
   // Se sobrescribe el metodo mousePressed() para controlar la
   // respuesta cuando el raton se pulse sobre uno de los dos
   // objetos visuales
   public void mousePressed( MouseEvent evt ) {
       if( evt.getComponent().getName().compareTo( "Frame" ) == 0 ) {
           System.out.println(
               "Capturado mousePressed sobre el objeto Frame" );
           }
       if( evt.getComponent().getName().compareTo( "Window" ) == 0 ) {
           System.out.println(
               "Capturado mousePressed sobre el objeto Window" );
           }
       }
   }
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Proceso1 extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
   System.out.println( "Recogido evento windowClosing desde el Frame" );
   System.exit( 0 );
       }
   }
```

Eventos de bajo nivel y semánticos

Aunque el conjunto de eventos semánticos es utilizado para propósitos diferentes que el conjunto de eventos de bajo nivel, desde el punto de vista de la programación la diferencia es muy poca.

La principal diferencia parece residir en la naturaleza del objeto evento que es pasado al controlador de eventos en el momento en que algo sucede. Utilizando la información del objeto evento, los eventos de bajo nivel pueden acceder al Componente específico que ha generado el evento, porque todas las clases de eventos de bajo nivel son subclases de la clase ComponentEvent. Una vez que la referencia a ese Componente está disponible, hay docenas de métodos de la clase Component que pueden ser invocados sobre el objeto, como getLocation(), getName(), getMaximumSize(), etc.

Los eventos de tipo semántico, por otro lado, no son subclase de la clase ComponentEvent, sino que son subclase de la superclase de ComponentEvent, es decir, son hermanos de ComponentEvent.

Al no ser subclase de ComponentEvent, los objetos evento pasados a controladores de eventos semánticos proporcionan un método para obtener una referencia al objeto que ha generado el evento y, por lo tanto, no pueden invocar los métodos de la clase Component sobre ese objeto.

Que lo anterior tenga importancia o no depende de las necesidades del programa. Por ejemplo, si se necesita determinar la posición del objeto que ha generado un evento, se puede hacer procesando un evento de bajo nivel y, probablemente, no se pueda determinar esa posición procesando un evento semántico; y se indica probablemente, porque nunca se puede decir nunca, si no queremos que alguien demuestre que estamos equivocados.

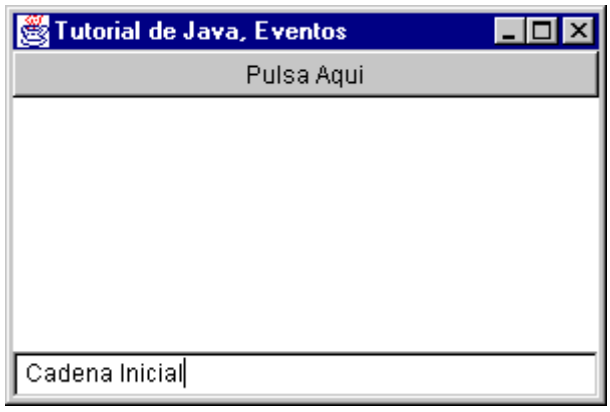
Quitando la posibilidad de acceder al objeto que ha generado el evento, el nombre del objeto está disponible tanto para los eventos de bajo nivel como para los eventos

semánticos. En ambos casos, el nombre del objeto está encapsulado en el objeto evento pasado como parámetro y puede extraerse y comprobarse utilizando métodos de la clase String. En muchas ocasiones, el saber el nombre del objeto es suficiente para conseguir el resultado apetecido.

El programa Java1105, muestra algunas de las capacidades de los eventos de bajo nivel.

```
/**
* Cuando se arranca este programa y se pulsa en alguno de los tres
* componentes visuales de que consta el interfaz de usuario, se
* presenta informacion sobre el componente que se haya seleccionado
* con la pulsacion del raton
*/
import java.awt.*;
import java.awt.event.*;
public class java1105 {
   public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
   }


//Esta clase es para instanciar un objeto de tipo interfaz de usuario
class IHM {
   public IHM() {
        // Crea un objeto visual Campo de Texto (TextField)
        TextField miTexto = new TextField( "Cadena Inicial" );
        miTexto.setName( "CampoTexto" );
        // Crea un objeto visual Boton (Button)
        Button miBoton = new Button( "Pulsa Aqui" );
        miBoton.setName( "Boton" );
        // Crea un objeto visual Frame
        Frame miFrame = new Frame();
        miFrame.setSize( 300,200 );
        miFrame.setTitle( "Tutorial de Java, Eventos" );
        miFrame.setName( "Frame" );
        // Incorpora el Campo de Texto y el Boton al objeto de tipo Frame
        miFrame.add( "North",miBoton );
        miFrame.add( "South",miTexto );
        miFrame.setVisible(true);
        // Se instancia y registra in objeto MouserListener que procesara
        // todos los eventos del raton que se produzcan o sean generados
        // por los objetos Frame, Button y TextField
        ProcesoRaton procesoRaton = new ProcesoRaton();
        miFrame.addMouseListener( procesoRaton );
        miTexto.addMouseListener( procesoRaton );
        miBoton.addMouseListener( procesoRaton );
        // Se instancia y registra un objeto Receptor que hara que
        // concluya la ejecucion del programa cuando el usuario cierre
        // la ventana
        Proceso1 procesoVentana1 = new Proceso1();
        miFrame.addWindowListener( procesoVentana1 );
        }
   }
// Monitor de eventos de Bajo-Nivel
// Esta clase Receptor monitoriza todos los eventos mousePressed() de
// bajo-nivel. Cuando se produce un evento de pulsacion del raton,
// el controlador de eventos obtiene y presenta informacion variada
// acerca del objeto que ha generado el evento
class ProcesoRaton extends MouseAdapter {
   public void mousePressed( MouseEvent evt ) {
        System.out.println( "Nombre = " + evt.getComponent().getName() );
```

```
        try {
            System.out.println(
                "Nombre del padre = " +
                evt.getComponent().getParent().getName() );
        } catch( NullPointerException exception ) {
            System.out.println( "No hay ningun padre a este nivel" );
            }
        System.out.println( "Localizacion = " +
            evt.getComponent().getLocation().toString() );
```




```
System.out.println( "Tamano Minimo = " +
            evt.getComponent().getMinimumSize().toString() );
        System.out.println( "Tamano = " +
            evt.getComponent().getSize().toString() );
        System.out.println();
        }
    }
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Proceso1 extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

Sobre un objeto Frame se coloca en la parte superior un objeto Button y en al inferior un objeto TextField, tal como muestra la figura siguiente, que corresponde a la captura de la ventana resultante de la ejecución del ejemplo.

Un objeto MouseListener es instanciado y registrado para monitorizar eventos de bajo nivel mousePressed() sobre los tres objetos. Cuando se produce un evento de este tipo, el objeto receptor obtiene y presenta información sobre el objeto que ha generado el evento. Aunque esto es lo único que hace la aplicación, todos los métodos de la clase Component están disponibles para realizar cualquier otro tipo de acciones. También hay un objeto WindowListener que es instanciado y registrado para terminar el programa cuando el usuario decida cerrar el objeto Frame.

En el ejemplo Java1106, se incluyen controladores de bajo nivel y semánticos para los mismos tres componentes del programa anterior. Como antes, se coloca un objeto Button y un objeto TextField sobre un objeto Frame. El controlador de nivel semántico trata los eventos Action y el controlador de bajo nivel trata los eventos mousePressed y los eventos Focus sobre los mismos componentes.

```
/**
* Este ejemplo ilustra el manejo de los eventos de bajo-nivel y eventos
* semanticos.
* En los comentarios que preceden a cada una de las clases se explica
* lo que hace cada una de ellas y como manejan los eventos
*/
import java.awt.*;
import java.awt.event.*;
public class java1106 {
   public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
   }
// Esta clase es para instaciar un objeto de tipo interfaz de usuario
class IHM {
   public IHM() {
        // Crea un objeto visual Campo de Texto (TextField)
        TextField miTexto = new TextField( "Cadena Inicial" );
        miTexto.setName( "CampoTexto" );
        // Crea un objeto visual Boton (Button)
        Button miBoton = new Button( "Púlsame" );
        miBoton.setName( "Boton" );

        // Crea un objeto visual Frame
        Frame miFrame = new Frame();
        miFrame.setSize( 300,200 );
        miFrame.setTitle( "Tutorial de Java, Eventos" );
        miFrame.setName( "Frame" );
        // Incorpora el Campo de Texto y el Boton al objeto de tipo Frame
        miFrame.add( "North",miBoton );
        miFrame.add( "South",miTexto );
        miFrame.setVisible( true );
        // Se instancia y registra un objeto ActionListener que
        // monitorizara todos los eventos de acciones que tengan
        // su origen en el Campo de Texto y en el Boton
        ProcesoAccion procesoAccion = new ProcesoAccion();
        miTexto.addActionListener( procesoAccion );
        miBoton.addActionListener( procesoAccion );
        // Se instancia y registra un objeto FocusListener que
        // monitorizara todos los eventos producidos por cambios
        // en el foco que tengan su origen en el Campo de Texto y
        // en el Boton
        ProcesoFoco procesoFoco = new ProcesoFoco();
        miTexto.addFocusListener( procesoFoco );
        miBoton.addFocusListener( procesoFoco );
        // Se instancia y registra un objeto MouserListener que procesara
        // todos los eventos del raton que se produzcan o sean generados
        // por los objetos Frame, Button y TextField
        ProcesoRaton procesoRaton = new ProcesoRaton();
        miFrame.addMouseListener( procesoRaton );
        miTexto.addMouseListener( procesoRaton );
        miBoton.addMouseListener( procesoRaton );
        // Se instancia y registra un objeto receptor de eventos
        // para terminar la ejecucion del programa cuando el
        // usuario decida cerrar la ventana
        Proceso1 procesoVentana1 = new Proceso1();
        miFrame.addWindowListener( procesoVentana1 );
        }
   }
// Receptor de eventos Semanticos.
// Esta clase ActionListener se utiliza para instanciar un objeto
// Receptor que monitorice todos los eventos Action que se generen en
// los objetos TextField y Button. Cuando se produce un evento de
// tipo actionPreformed(), se presenta en pantalla el ActionCommand
```

```
// y la identificacion del componente que ha generado el evento.
// El objeto receptor distingue entre los componentes que de envian
// eventos sobre la base del nombre que se ha asignado a cada uno
// de los objetos y que se encuentra embebido en el objeto que es
// pasado como parametro cuando se produce el evento
class ProcesoAccion implements ActionListener {
   public void actionPerformed( ActionEvent evt ) {
        System.out.println( "evt.getActionCommand() = " +
            evt.getActionCommand() );
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
                "Capturado actionPerformed sobre el objeto CampoTexto" );
            }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
                "Capturado actionPerformed sobre el objeto Boton" );
            }
        }
   }
// Receptor de eventos de Bajo-Nivel.
// Esta clase FocusListener se utiliza para instanciar un objeto
// Receptor que monitorice los eventos relacionados con el foco
// que se generen en los objetos TextField y Button. Cuando se
// produce un evento de tipo focusLost() o focusGained(), se
// presenta en pantalla la identificacion del componente que ha
// generado el evento. El objeto receptor distingue entre los
// componentes que de envian eventos sobre la base del nombre que
// se ha asignado a cada uno de los objetos y que se encuentra
// embebido en el objeto que es pasado como parametro cuando se
// produce el evento
class ProcesoFoco implements FocusListener{
   public void focusGained( FocusEvent evt ) {
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
                "Capturado focusGained sobre el objeto CampoTexto" );
            }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
                "Capturado focusGained sobre el objeto Boton" );
            }
        }
   public void focusLost( FocusEvent evt ) {
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
                "Capturado focusLost sobre el objeto CampoTexto" );
            }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
                "Capturado focusLost sobre el objeto Boton" );
            }
        }
   }
// Receptor de eventos de Bajo-Nivel.
// Esta clase receptor se utiliza para monitorizor los eventos
// de pulsacion de los botones del raton sobre el objeto Frame,
// el objeto Button y el objeto TextField. El mensaje indentifica
// el componente que ha generado el evento.El receptor distingue
// entre los componentes que de envian eventos sobre la base del
// nombre que se ha asignado a cada uno de los objetos y que se
// encuentra embebido en el objeto que es pasado como parametro
// cuando se produce el evento
class ProcesoRaton extends MouseAdapter {
   public void mousePressed( MouseEvent evt ) {
        if( evt.toString().indexOf("on Frame") != -1 ) {
            System.out.println(
```

```
                "Capturado mousePressed sobre el objeto Frame" );
            }
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
                "Capturado mousePressed sobre el objeto CampoTexto" );
            }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
                "Capturado mousePressed sobre el objeto Boton" );
            }
        }
    }
// Receptor de eventos de Bajo-Nivel.
// Este repector de eventos de la ventana se utiliza para concluir la
// ejecucion del programa cuando el usuario pulsa el boton de cerrar
class Proceso1 extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

Eventos de Foco

En Java, cuando se dice que un Componente tiene el foco, significa que las entradas de teclado se dirigen a ese Componente. Hay muchas razones por las que pasa el foco de un Componente a otro, y cuando esto sucede, se genera un evento focusLost() en el Componente que pierde el foco y el que recibe el foco, genera un evento focusGained(). Es base a esta pequeña explicación, es fácil comprender que haya muchos tipos de Componentes que pueden generar este tipo de eventos, ya que cualquier Componente que pueda ganar el foco también podrá perderlo y generará esos eventos.

Hay algunos Componentes como son los Botones y los Campos de Texto, que ganan el foco automáticamente cuando se pulsa sobre ellos con el ratón. En otros Componentes, sin embargo, esto no ocurre, como por ejemplo, en las Etiquetas, aunque estos Componentes pueden ganar el foco si lo solicitan.

Eventos de Acción

Un evento Action puede ser generado por muchos Componentes. Por ejemplo, pulsando con el ratón o pulsando la tecla Retorno cuando el foco está sobre un Campo de Texto, se generará un evento de este tipo. La terminología deriva de que las acciones de usuario generan acciones hacia el programa para realizar algo específico, en función de la naturaleza del Componente del que ha surgido el mensaje. Por ejemplo, si un Botón tiene la etiqueta "Salir" y es pulsado con el ratón, esto significa que el usuario está esperando que el programa realice alguna acción que el interpreta como una salida del programa.

Objeto ActionListener

En este ejemplo, un objeto ActionListener es instanciado y registrado para monitorizar de forma semántica eventos de tipo actionPerformed() sobre el Botón y el Campo de Texto. Cuando se genera un evento de este tipo, hay cierta información acerca de este evento que es encapsulada en un objeto que se le pasa al método actionPerformed() del objeto Receptor. La documentación oficial del JDK le llama a esto command name. A esta información puede accederse a través de la invocación al método getActionCommand() sobre el objeto. En este ejemplo, se accede al command name y se presenta en la pantalla.

El nombre del comando asociado con el Botón es simplemente el texto o etiqueta que figura sobre el botón. El nombre del comando asociado al Campo de Texto es el texto actual que contiene el campo. Esta información puede ser utilizada por de diferente forma por los Componentes; por ejemplo, puede ser utilizada para distinguir entre varios botones si no se permite cambiar sus etiquetas durante la ejecución del programa. También se puede utilizar para capturar la entrada del usuario desde un objeto Campo de Texto.

El objeto de tipo ActionEvent que se pasa al método actionPerformed() también incluye el nombre del componente, que en este ejemplo es utilizado en comparaciones para identificar el Componente que está generando el evento. Una forma de hacer esto es utilizar el método indexOf() de la clase String, para determinar si un Componente determinado se encuentra incluido en un objeto específico.

En este programa, cada vez que se invoca al método actionPerformed(), el código en el cuerpo del método utiliza precisamente la llamada al método indexOf() para identificar el Componente que ha generado el evento y presenta un mensaje en pantalla indicando el nombre del componente, su command name.

Objeto FocusListener

Se instancia un objeto FocusListener y también se registra, para monitorizar a bajo nivel los eventos focusGained() y focusLost(), sobre el botón y el campo de texto.

Cuando se produce un evento focusGained(), se presenta en pantalla un mensaje indicando el objeto que ha ganado el foco. De la misma forma, cuando ocurre un evento focusLost(), también se presenta un mensaje que indica el objeto que ha perdido el foco.

El objeto que gana o pierde el foco se identifica a través de comprobaciones condicionales sobre el objeto FocusEvent pasado como parámetro, del mismo modo que se hace con el objeto ActionEvent utilizado en los eventos de acción.

Objeto MouseListener

Un objeto MouseListener es instanciado y registrado para monitorizar a bajo nivel eventos de tipo mousePressed() sobre los tres objetos que conforman el interfaz. No se controlan otros de los muchos eventos que se producen por mantener una cierta simplicidad en el ejemplo.

El objeto MouseListener distingue entre los tres objetos: Frame, Button y TextField, en base al nombre del componente que se asigna a cada objeto en el momento de su instanciación. Si el programador no asigna nombres a los Componentes cuando son instanciados, el sistema le asignara nombres por defecto. Estos nombres que asigna el sistema tienen el formato frame0, frame1, frame2, etc., con la parte principal del nombre indicando el tipo de Componente y el dígito final se va asignando en el mismo orden en que se van instanciando los objetos.

En este ejemplo, se utiliza el método indexOf() sobre el objeto MouseEvent para determinar el nombre del componente. Esto es un poco menos complejo que el método utilizado en ejemplos anteriores, basado en recuperar el objeto e invocar su método getName().

Cuando se produce un evento mousePressed() sobre cualquiera de los tres objetos visuales, el objeto MouseListener presenta un mensaje en pantalla identificando el objeto que ha generado el evento.

Objeto WindowListener

Por último, para terminar la descomposición del programa, un objeto WindowListener es instanciado y registrado para terminar la ejecución de la aplicación cuando el usuario cierra el objeto Frame.

La salida por pantalla que se produce tras algunas acciones del usuario, es la que se reproduce en las siguientes secuencias.

- Si se pulsa el ratón dentro del Frame, pero fuera del botón y del campo de texto, aparece el mensaje:

```
Capturado mousePressed sobre el objeto Frame
```

- Si se pulsa el ratón sobre el campo de texto cuando el botón tiene el foco, se genera la salida:

```
Capturado mousePressed sobre el objeto CampoTexto
Capturado focusLost sobre el objeto Boton
Capturado focusGained sobre el objeto CampoTexto
```

- Pulsando la tecla Retorno cuando el campo de texto tiene el foco, aparece en pantalla el mensaje:

```
evt.getActionCommand() = Cadena Inicial
Capturado actionPerformed sobre el objeto CampoTexto
```

- Pulsando el ratón sobre el botón del ratón, cuando el campo de texto tiene el foco, se genera la salida:

```
Capturado mousePressed sobre el objeto Boton
Capturado focusLost sobre el objeto CampoTexto
Capturado focusGained sobre el objeto Boton
evt.getActionCommand() = Púlsame
Capturado actionPerformed sobre el objeto Boton
```

Para mantener la simplicidad, la respuesta a los eventos en este programa se limita a presentar información en pantalla. Obviamente, cuando en flujo de ejecución se encuentra dentro del código que controla el evento, el programador puede implementar respuestas diferentes.

También hay que hacer notar al lector, que en el ejemplo se da el hecho de que una sola acción del usuario origina varios tipos diferentes de eventos.

Control del Foco

En este apartado se entra un poco más a fondo sobre el control del Foco en el nuevo modelo de eventos del JDK, de forma que para demostrar sus posibilidades, en el programa de ejemplo Java1107, , que se utilizará para ilustrar el control del Foco, se forzará a que un objeto que normalmente no recibe el foco en los programas, como es una Etiqueta (Label), no solamente gane el foco, sino que también responda a eventos del teclado; para mostrar de esta forma el poder, consistencia y flexibilidad de este nuevo modelo de eventos.

```
/**
* Este ejemplo muestra como componentes visuales, como es un Label,
```

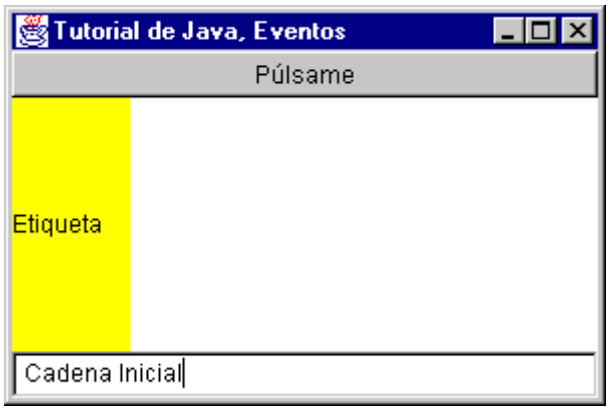
```
* pueden ser forzados a ganar el foco, y pueden responder a eventos
* del teclado; al contrario de lo que sucede en un programa normal
* al uso en entornos de ventanas
*/
import java.awt.*;
import java.awt.event.*;
public class java1107 {
   public static void main( String args[] ) {
       // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
       IHM ihm = new IHM();
       }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
   // Guardamos los nombres de los componente en variables
   String nombreMiTexto;
   String nombreMiBoton;
   String nombreMiFrame;
   String nombreMiEtiqueta;
   // Variables de referencia pasadas como parámetros
   Frame miFrame;
   Label miEtiqueta;
   public IHM() {
       // Crea un objeto visual Campo de Texto (TextField)
       TextField miTexto = new TextField( "Cadena Inicial" );
       nombreMiTexto = miTexto.getName();
       // Crea un objeto visual Etiqueta (Label)
       miEtiqueta = new Label( "Etiqueta" );
       nombreMiEtiqueta = miEtiqueta.getName();
       miEtiqueta.setBackground( Color.yellow );
       // Crea un objeto visual Boton (Button)
       Button miBoton = new Button( "Púlsame" );
       nombreMiBoton = miBoton.getName();
       // Crea un objeto visual Frame
       miFrame = new Frame();
       miFrame.setSize( 300,200 );
       miFrame.setTitle( "Tutorial de Java, Eventos" );
       nombreMiFrame = miFrame.getName();
       // Incorpora el Campo de Texto y el Boton al objeto de tipo Frame
       miFrame.add( "North",miBoton );
       miFrame.add( "South",miTexto );
       miFrame.add( "West",miEtiqueta );
       miFrame.setVisible( true );
       // Pasamos el foco al campo de texto
       miTexto.requestFocus();
       // Se instancia y registra un objeto FocusListener que
       // monitorizara todos los eventos producidos por cambios
       // en el foco. En este caso se hace
       // que el objeto receptor cambie el color del objeto para
       // indicar si ha ganado o perdido el foco
       ProcesoFoco procesoFoco = new ProcesoFoco( this );
       miFrame.addFocusListener( procesoFoco );
       miEtiqueta.addFocusListener( procesoFoco );
       // Se instancia y registra un objeto MouserListener que procesara
       // todos los eventos del raton que se produzcan o sean generados
       // por los objetos Frame y Label. El objeto receptor en este
       // caso pedirá explícitamente el foco para cada uno de estos
       // componentes cuando se pulse sobre ellos
       ProcesoRaton procesoRaton = new ProcesoRaton( this );
       miFrame.addMouseListener( procesoRaton );
       miEtiqueta.addMouseListener( procesoRaton );
      // Se instancia y registra un objeto KeyListener que procesará
      // los eventos del teclado sobre los cuatro objetos visuales que
      // componen el interfaz. Cuando se pulse una tecla, se presenta
```

```
      // un mensaje indicando qeu el componente que tiene el foco ha
      // recibido un evento de pulsación de tecla
       ProcesoTeclado procesoTeclado = new ProcesoTeclado( this );
       miFrame.addKeyListener( procesoTeclado );
       miTexto.addKeyListener( procesoTeclado );
       miBoton.addKeyListener( procesoTeclado );
       miEtiqueta.addKeyListener( procesoTeclado );
       // Se instancia y registra un objeto receptor de eventos
       // para terminar la ejecucion del programa cuando el
       // usuario decida cerrar la ventana
       Proceso1 procesoVentana1 = new Proceso1();
       miFrame.addWindowListener( procesoVentana1 );
       }
   }
// Receptor de eventos de Bajo-Nivel.
// Esta clase FocusListener se utiliza para instanciar un objeto
// Receptor que monitorice los eventos relacionados con el foco
// que se generen en los objetos Frame y Label. Cuando se
// produce un evento de tipo focusLost() o focusGained(), se
// hace que cambie el color del componente visual, para indicar
// que ha perdido o ganado el foco
class ProcesoFoco implements FocusListener{
      IHM obj;
      ProcesoFoco( IHM iObj ) {
            obj = iObj;
          }
   public void focusGained( FocusEvent evt ) {
       if( evt.toString().indexOf("on "+obj.nombreMiFrame) != -1 ) {
                obj.miFrame.setBackground( Color.blue );
           }
       if( evt.toString().indexOf("on "+obj.nombreMiEtiqueta) != -1 ) {
                obj.miEtiqueta.setForeground( Color.red );
           }
       }
   public void focusLost( FocusEvent evt ) {
       if( evt.toString().indexOf("on "+obj.nombreMiFrame) != -1 ) {
                obj.miFrame.setBackground( Color.white );
           }
       if( evt.toString().indexOf("on "+obj.nombreMiEtiqueta) != -1 ) {
                obj.miEtiqueta.setForeground( Color.black );
           }
       }
   }
// Receptor de eventos de Bajo-Nivel.
// Esta clase receptor se utiliza para monitorizor los eventos
// de pulsacion de los botones del raton sobre el objeto Frame,
// el objeto Etiqueta. Estos componentes no reciben automáticamente
// el foco cuando se pican con el ratón, por lo que la clase tiene
// que reclamarlo
class ProcesoRaton extends MouseAdapter {
      IHM obj;
      ProcesoRaton( IHM iObj ) {
            obj = iObj;
          }
   public void mousePressed( MouseEvent evt ) {
       if( evt.toString().indexOf("on "+obj.nombreMiFrame) != -1 ) {
                obj.miFrame.requestFocus();
           }
       if( evt.toString().indexOf("on "+obj.nombreMiEtiqueta) != -1 ) {
                obj.miEtiqueta.requestFocus();
           }
       }
   }
// Receptor de eventos de Bajo-Nivel.
// Este receptor de eventos se utiliza para recoger las pulsaciones del
```

```
// teclado y presentar un mensaje cada vez que se produce un enveto de
// tipo keyPressed(). El componente visual que tiene en ese momento el
// foco es el que genera el evento, que es identificado y presentado a
// través de un mensaje en la pantalla.
// Lo más significativo en este caso es que cualquier componente que
// tenga el foco puede generar un evento de teclado, incluyendo los
// componentes, que como el Label, normalmente nunca tienen el foco
class ProcesoTeclado extends KeyAdapter {
      IHM obj;
      ProcesoTeclado( IHM iObj ) {
            obj = iObj;
          }
   public void keyPressed( KeyEvent evt ) {
       if( evt.toString().indexOf("on "+obj.nombreMiFrame) != -1 ) {
                  System.out.println(
                        "Evento de teclado desde "+obj.nombreMiFrame );
          }
       if( evt.toString().indexOf("on "+obj.nombreMiTexto) != -1 ) {
                  System.out.println(
                        "Evento de teclado desde "+obj.nombreMiTexto );
          }
       if( evt.toString().indexOf("on "+obj.nombreMiBoton) != -1 ) {
                  System.out.println(
                        "Evento de teclado desde "+obj.nombreMiBoton );
          }
       if( evt.toString().indexOf("on "+obj.nombreMiEtiqueta) != -1 ) {
                  System.out.println(
                  "Evento de teclado desde "+obj.nombreMiEtiqueta );
          }
       }
   }
// Receptor de eventos de Bajo-Nivel.
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Proceso1 extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
       System.exit( 0 );
       }
   }
```

En realidad, se utiliza una combinación de eventos de bajo nivel de foco, de ratón y de teclado, para permitir que los cuatro objetos del interfaz: un Frame, un Label, un Button y un TextField, puedan capturar el foco y responder a eventos del teclado. La imagen inicial al arrancar la aplicación del interfaz, es la que muestra la figura siguiente.

Probablemente, el que una Etiqueta responda a un evento de teclado puede no resultar útil a nadie, si se sigue la convencionalidad al uso para programas basados en entornos de ventanas, pero permite hacer una cuantas locuras. El programa es bastante largo, pero muy repetitivo. El hecho de que sea repetitivo habla en favor del nuevo Modelo de Delegación de Eventos, ya que el código necesario para manejar un tipo de evento sobre un tipo de objeto es muy similar al que se necesita para manejar otros tipos de eventos sobre el mismo o diferentes objetos. Una vez que se ha comprendido la sintaxis y entendido el mecanismo, el programador se puede considerar capacitado para escribir programas que probablemente estarán libres de errores. Así de fácil lo ha puesto Sun.



Sobre el objeto Frame se coloca un Botón, un Campo de Texto y una Etiqueta. El borde de la etiqueta no puede verse porque se pinta con el color de fondo.

En el código fuente del ejemplo los comentarios previos a la definición de cada una de las clases indican las acciones que realiza.

A continuación se entra en detalles en cada uno de los diferentes tipos de eventos que se tratan en el ejemplo, teniendo en cuenta que lo especial del ejemplo es la posibilidad que se ha incorporado al objeto Label de que pueda generar eventos de foco.

Eventos del Foco

Se instancia y registra un objeto receptor del foco, FocusListener, para recibir los eventos focusGained() y focusLost() sobre el Frame y el objeto Label. Cuando se producen estos eventos, el receptor del foco hace que el Frame cambie a color azul, o cuando lo tiene el objeto Label, hace que el texto se vuelva rojo.

Una de las ventajas del modelo de Delegación de Eventos es la posibilidad de filtrar eventos, con lo que se consigue que el controlador y manejador de eventos sólo trate en su código aquellos eventos que resulten interesantes. En este ejemplo concreto, los objetos TextField y Button también generan eventos de foco; sin embargo, no hay razón alguna para procesarlos, así que lo que se hace es simplemente no registrar al receptor del foco sobre ellos, para que no sea notificado cuando se producen este tipo de eventos sobre esos objetos. Además, estos dos objetos proporcionan indicación visual de que han ganado el foco sin intervención del programador, es el propio sistema el que lo hace.

Eventos del Ratón

Sobre el objeto Frame y sobre el objeto Label, también se registra e instancia un objeto MouseListener para recibir los eventos mousePressed(). Las cuestiones de filtrado de los eventos del foco se pueden aplicar también a este caso.

Cuando se produce un evento de este tipo, el objeto MouseListener invoca al método requestFocus() sobre el componente visual que ha generado el evento. Esto hace que los objetos Frame o Label reciban el foco en el momento en que se pulse el ratón sobre ellos; aunque en el caso del objeto Label, su comportamiento no sea nada usual.

También en este caso, el botón y el campo de texto reciben automáticamente el foco cuando se pulsa sobre ellos, sin necesidad de que el programador haga nada.

Eventos del Teclado

Un objeto KeyListener es instanciado y registrado para recibir los eventos del teclado keyPressed() sobre los objetos Frame, Label, Button y TextField. Cuando se pulsa una tecla, el objeto que tenga el foco en ese momento generará un evento keyPressed(), e incluso la Etiqueta responde al teclado (en este caso). El objeto KeyListener determina el componente visual que ha generado el evento y presenta un mensaje en pantalla.

Eventos de la Ventana

Finalmente, se instancia y registra un objeto WindowListener para poder terminar la ejecución del programa cuando el usuario cierre el objeto Frame.

Asignación Automática de Nombres

Siempre que se instancia un objeto de la clase Component, el sistema le proporciona automáticamente un nombre. A este nombre se puede acceder utilizando el método getName() de la clase Component, y se puede utilizar el método setName() de la misma clase para asignarle un nombre específico a un objeto. En este ejemplo se utiliza el nombre del Componente para determinar qué objeto ha generado un evento.

En el ejemplo anterior se establecían los nombres de varios Componentes utilizando el método setName(). Sin embargo, el JDK no fuerza a que los nombres sean únicos cuando se fijan por parte desde el código, lo cual puede ocasionar problemas al programador si no se tiene cuidado en asignar nombres únicos a los Componentes. En el ejemplo actual, se asume que los nombres de los componentes visuales que componen el interfaz gráfico son únicos y asignados automáticamente por el sistema, y en vez de asignar nombres y compararlos, lo que se hace es recuperar los nombres que asigna el sistema a cada Componente para distinguir posteriormente entre los Componentes que han generado un evento.

Movimiento del foco

Los programas típicos de los entornos de ventanas que disponen de un interfaz gráfico permiten mover el foco entre los diversos Componentes del interfaz mediante la tecla del Tabulador en el sentido de las agujas del reloj y mediante la combinación Mayúsculas y Tabulador, realiza el movimiento del foco en sentido inverso a través de los Componentes. Esto es lo que se conoce como Movimiento Transversal.

En el JDK este método de desplazamiento está soportado sin que el programador tenga que realizar esfuerzo alguno. El orden en que se mueve el foco entre los Componentes viene determinado, aparentemente, por el orden en que se han instanciado y parece no poder modificarse, tal como sucede en otros entornos de desarrollo.

En el esquema estándar de movimiento del foco, se excluyen algunos tipos de Componentes para que no puedan ganar el foco (una Etiqueta, por ejemplo) y, en principio no puede forzarse a que sea de forma diferente. Sin embargo, en el ejemplo que se presenta, se implementa un esquema de desplazamiento del foco utilizando una tecla

distinta a la estándar. Aunque nunca se utilice esto en un programa normal, sí que proporciona una buena demostración del uso de ciertos aspectos del modelo de Delegación de Eventos que se ha introducido.

En el programa Java1108 , se implementa la capacidad de movimiento del foco utilizando las teclas F12 y Mayúsculas y F12. También se hace que aquellos Componentes que normalmente no se tienen en cuenta a la hora de pasarles el foco, puedan adquirirlo. También se hace que el movimiento del foco sea independiente del orden en que se instancian los Componentes, sin demasiado esfuerzo adicional en lo que a programación se refiere.

```
/**
* Este programa implementa un método especial de desplazamiento del
* foco a través de los elemntos visuales de un interfaz, utilizando
* la tecla F12 y la combinación Mayúsculas+F12, como alternativa
* al uso de las teclas de desplazamiento estándar Tab y Mayúsculas+Tab
* Las teclas del Tabulador y Mayúsculas+Tab también funcionan porque
* son implementadas automáticamente por el sistema, y se saltan los
* objetos Frame y Label.
* El esquema que implementa este código sí hace pasar el foco por los
* objetos Frame y Label, aunque por simplicidad no realiza ninguna
* respuesta al teclado cuando el componente tiene el foco. Cuando el
* foco está en la Etiqueta, su texto es rojo, y en caso de no tener
* el foco, será negro. La misma indicación se implementa para el
* objeto Frame que será azul cuando tiene el foco y blanco si no lo
* tiene. El Campo de Texto y el Botón implementan su propia indicación
* de que tienen o no el foco, sin necesidad de que se indique nada
* desde el programa.
* El programa también recupera y guarda los nombres de los
* componentes que son asignados por el sistema, asumiendo que siempre
* el sistema asignará nombres únicos a los Componentes, y utiliza
* estos nombres para determinar cual ha sido el componente que ha
* generado un evento cuando lo necesita.
*/
import java.awt.*;
import java.awt.event.*;
public class java1108 {
   public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
   // Guardamos los nombres de los componente en variables
   String nombreMiTexto;
   String nombreMiBoton;
   String nombreMiFrame;
   String nombreMiEtiqueta;
   // Variables de referencia pasadas como parámetros
   Frame miFrame;
      TextField miTexto;
   Label miEtiqueta;
      Button miBoton;
      int indiceFoco;
   public IHM() {
        // Crea un objeto visual Campo de Texto (TextField)
        miTexto = new TextField( "Cadena Inicial" );
        nombreMiTexto = miTexto.getName();
        // Crea un objeto visual Etiqueta (Label)
        miEtiqueta = new Label( "Etiqueta" );
        nombreMiEtiqueta = miEtiqueta.getName();
```

```
        miEtiqueta.setBackground( Color.yellow );
        // Crea un objeto visual Boton (Button)
        miBoton = new Button( "Púlsame" );
        nombreMiBoton = miBoton.getName();
        // Crea un objeto visual Frame
        miFrame = new Frame();
        miFrame.setSize( 300,200 );
        miFrame.setTitle( "Tutorial de Java, Eventos" );
        nombreMiFrame = miFrame.getName();
        // Incorpora el Campo de Texto y el Boton al objeto de tipo Frame
        miFrame.add( "North",miBoton );
        miFrame.add( "South",miTexto );
        miFrame.add( "West",miEtiqueta );
        miFrame.setVisible( true );
        // Pasamos el foco al campo de texto
        miTexto.requestFocus();
        indiceFoco = 0;
        // Se instancia y registra un objeto FocusListener que
        // monitorizara todos los eventos producidos por cambios
        // en el foco. En este caso se hace que el objeto receptor
        // cambie el color para indicar que ha ganado o perdido
        // la posesión del foco
        ProcesoFoco procesoFoco = new ProcesoFoco( this );
        miFrame.addFocusListener( procesoFoco );
        miEtiqueta.addFocusListener( procesoFoco );
        // Se instancia y registra un objeto KeyListener que procesará
        // los eventos del teclado sobre los cuatro objetos visuales que
        // componen el interfaz. En este caso se hace que el foco se
              // mueva según se haya definido en el programa cuando el
        // usuario pulse la tecla F12, y que se mueva en sentido
        // contrario cuando pulse Mayúsculas+F12
        ProcesoTeclado procesoTeclado = new ProcesoTeclado( this );
        miFrame.addKeyListener( procesoTeclado );
        miTexto.addKeyListener( procesoTeclado );
        miBoton.addKeyListener( procesoTeclado );
        miEtiqueta.addKeyListener( procesoTeclado );
        // Se instancia y registra un objeto receptor de eventos
        // para terminar la ejecucion del programa cuando el
        // usuario decida cerrar la ventana
        Conclusion procesoVentana = new Conclusion();
        miFrame.addWindowListener( procesoVentana );
        }
    }
// Receptor de eventos de Bajo-Nivel.
// Esta clase FocusListener se utiliza para instanciar un objeto
// Receptor que monitorice los eventos relacionados con el foco
// que se generen en los objetos Frame y Label. Cuando se
// produce un evento de tipo focusLost() o focusGained(), se
// hace que cambie el color del componente visual, para indicar
// que ha perdido o ganado el foco
class ProcesoFoco implements FocusListener{
      IHM obj;
      ProcesoFoco( IHM iObj ) {
            obj = iObj;
          }
    public void focusGained( FocusEvent evt ) {
        if( evt.toString().indexOf("on "+obj.nombreMiFrame) != -1 ) {
                  obj.miFrame.setBackground( Color.blue );
            }
        if( evt.toString().indexOf("on "+obj.nombreMiEtiqueta) != -1 ) {
                  obj.miEtiqueta.setForeground( Color.red );
            }
        }
    public void focusLost( FocusEvent evt ) {
        if( evt.toString().indexOf("on "+obj.nombreMiFrame) != -1 ) {
```

```
                    obj.miFrame.setBackground( Color.white );
                }
            if( evt.toString().indexOf("on "+obj.nombreMiEtiqueta) != -1 ) {
                    obj.miEtiqueta.setForeground( Color.black );
                }
            }
    }
// Receptor de eventos de Bajo-Nivel.
// Este receptor de eventos se utiliza para recoger las pulsaciones del
// del teclado e ir moviendo el foco entre los componentes visuales si
// las teclas pulsadas son F12 o Mayúsculas+F12. Si se pulsa solamente
// F12, el foco se moverá por los componentes según las agujas del
// reloj, incluyendo a la Etiqueta y al Frame entre los componentes
// por los que pasa. Si se pulsa Mayúsculas+F12, el movimiento es en
// la dirección contraria.
// Las teclas estándar de movimiento del foco Tab y Mayúsculas+Tab,
// también funcionan, pero no incluyen en la lista de objetos por los
// que mueven el foco al Frame ni a la Etiqueta
class ProcesoTeclado extends KeyAdapter {
      IHM obj;
      ProcesoTeclado( IHM iObj ) {
            obj = iObj;
          }
   public void keyPressed( KeyEvent evt ) {
       int codigoTecla = evt.getKeyCode();
            // Comprobamos primero que se haya pulsado solamente la
            // tecla F12 en solitario
       if( (codigoTecla == KeyEvent.VK_F12) &&
           (evt.getModifiers() != InputEvent.SHIFT_MASK) ) {
                  // Se mueve el foco en la dirección de las agujas del
                  // reloj al siguiente componente de la lista
           if( ++obj.indiceFoco > 3 )
                       obj.indiceFoco = 0;
           switch( obj.indiceFoco ) {
               case 0:
                              obj.miTexto.requestFocus();
                              break;
               case 1:
                              obj.miEtiqueta.requestFocus();
                              break;
               case 2:
                              obj.miFrame.requestFocus();
                              break;
               case 3:
                              obj.miBoton.requestFocus();
                              break;
               }
           }
            // Ahora se comprueba que se hayan pulsado las dos teclas a
            // la vez, F12 y la tecla de las Mayúsculas
       if( (codigoTecla == KeyEvent.VK_F12) &&
           (evt.getModifiers() == InputEvent.SHIFT_MASK) ) {
       // Se mueve el foco en la dirección opuesta a las agujas del
       // reloj al siguiente componente de la lista
           if( --obj.indiceFoco < 0 )
                       obj.indiceFoco = 3;
           switch( obj.indiceFoco ) {
               case 0:
                              obj.miTexto.requestFocus();
                              break;
               case 1:
                              obj.miEtiqueta.requestFocus();
                              break;
               case 2:
                              obj.miFrame.requestFocus();
```

```
                                break;
                    case 3:
                                obj.miBoton.requestFocus();
                                break;
                    }
               }
          }
     }
// Receptor de eventos de Bajo-Nivel.
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Conclusion extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
       System.exit( 0 );
       }
   }
```

En el ejemplo Java1107 , se mostraba cómo un Componente podía ganar el foco y responder al teclado; y cómo ganaba el foco cuando recibía la pulsación del ratón. Por simplicidad, en el ejemplo que sigue se ha eliminado esta capacidad del programa, pero podría implementarse del mismo modo.

Aunque el listado el programa es largo, vuelve a ser repetitivo. Se coloca un objeto Button, un objeto TextField y un objeto Label sobre un objeto Frame para componer el interfaz gráfico que presenta la aplicación.

Eventos del Foco

Se instancia y registra un objeto receptor de eventos del foco, FocusListener, para recibir los eventos focusGained() y focusLost() sobre los objetos Frame y Label. Cuando se producen estos eventos, el objeto FocusListener hace que el Frame, cuando tiene el foco, cambie su color de fondo a azul y, cuando es la Etiqueta quien tiene el foco, el color del texto se vuelve rojo.

Eventos del Teclado

Un objeto KeyListener es instanciado y registrado para recibir eventos keyPressed() desde el Frame, etiqueta, botón y campo de texto.

Cuando se pulsa una tecla, el objeto que tiene el foco genera un evento keyPressed(), aunque sea un objeto, como la Etiqueta, que normalmente no genere este tipo de eventos. Si se pulsa la tecla F12 o Mayúsculas-F12, el objeto KeyListener implementa un esquema definido en el programa que mueve el foco al Componente siguiente o al anterior, respectivamente.

Al contrario que el método estándar de control del foco con la tecla del Tabulador, la determinación de quién es el Componente siguiente o anterior al que se desplazará el foco, es independiente del orden en que se han instanciado los Componentes. Y también, al contrario que en el movimiento estándar del foco, en este ejemplo, se han incluido los objetos Label y Frame en la secuencia de movimiento.

Barras de desplazamiento

El control de los eventos producidos por las barras de desplazamiento es diferente a lo que se ha presentado en los ejemplos que se han visto hasta ahora, como es el caso de la pulsación de botones del ratón.

Para crear un objeto receptor de eventos del ratón, se implementa un interfaz MouseListener, o se extiende la clase MouseAdapter; sin embargo, para crear un objeto receptor de eventos para una Barra de Desplazamiento, no se puede implementar un objeto ScrollbarListener o extender una clase ScrollbarAdapter, porque no existen. En su lugar, hay que implementar un interfaz AdjustmentListener; que no dispone de una clase AdjustmentAdapter, porque no es necesaria ya que el interfaz solamente declara el método adjustmentValueChanged().

En el interfaz MouseListener se declaran cinco tipos diferentes de eventos de ratón:

```
mouseClicked()
mouseEntered()
mouseExited()
mousePressed()
mouseReleased()
```

Aunque no se han tratado todavía, hay dos tipos de eventos adicionales que están declarados en el interfaz MouseMotionListener:

```
mouseDragged()
mouseMoved()
```

Cada uno de los eventos de ratón citados, están representados por la declaración de un método en uno u otro de los interfaces definidos para crear clases receptoras de la actividad del ratón. El programador puede sobreescribir aquellos métodos de los eventos que realmente le interesen.

La clase AdjustmentEvent define varios métodos que se pueden utilizar para extraer información del objeto, en caso de necesitar acceder a esa información. Datos como el valor de la barra o el nombre del Componente Scrollbar, también están codificados en el objeto.

Los tipos de eventos de ajuste están definidos como variables estáticas en la clase AdjustmentEvent, y son:

```
UNIT_INCREMENT
UNIT_DECREMENT
BLOCK_INCREMENT
BLOCK_DECREMENT
TRACK
```

Los dos eventos UNIT se generan al pulsar con el botón sobre las flechas de los extremos de la barra de desplazamiento . Los dos eventos de tipo BLOCK se generan al pulsar dentro de la barra, a ambos lados del marcador. El evento TRACK se produce al desplazar el marcador de la barra.

Hay métodos para fijar los diferentes parámetros de la barra de desplazamiento, incluyendo el rango, el tamaño del marcador, el número de unidades de desplazamiento, tanto como unidad o como bloque, etc.

Lo que sorprende un poco de este planteamiento, es que parece que está definido de forma general, como si estuviesen definidos pensando en el soporte de una familia de Componentes que operasen sobre la barra de desplazamiento. Sin embargo, en ninguna otra parte de la documentación del JDK se habla de que haya Componentes que utilicen este interfaz de ajuste. Quizá Sun esté pensando en alguna sorpresa.

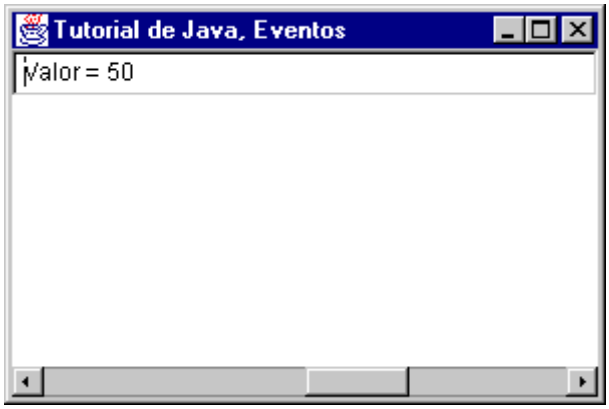
En el ejemplo Java1109 , se coloca un objeto Scrollbar y un objeto TextField sobre un objeto Frame.

```
/**
* Este programa permite comprobar los eventos de Ajuste que se generan
* en una barra de desplazamiento.
* Si se ejecuta este ejemplo se podrá comprobar que los botones situados
* el los bordes de la barra, las zonas entre estos botones y los
* límites del marcador, y el marcador en sí mismo, generar eventos de
* tipo diferente
*/
import java.awt.*;
import java.awt.event.*;
public class java1109 {
   public static void main( String args[] ) {
       // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
       IHM ihm = new IHM();
       }
   }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
   // Variables de referencia pasadas como parámetros
   Frame miFrame;
      TextField miTexto;
   Scrollbar miBarra;
   public IHM() {
       // Crea un objeto visual Frame
       miFrame = new Frame( "Tutorial de Java, Eventos" );
            miFrame.setSize( 300,200 );
       // Crea un objeto visual Barra de Desplazamiento (Scrollbar)
       // con un rango de 1 a 100, la posición incial en 50, un
       // marcador de anho 20 y un incremento de 15. Cuando un
       // objeto AdjustmentEvent se presenta en pantalla, la anchura
       // del marcador (bubble) o tamaño de paginación se identifica
       // como "vis" que quiere decir "visible"
       // La barra no centra el indicador sobre el valor que se le
       // indica, sino que se alinea con la parte izquierda, por lo
       // que no se podrán alcanzar los valores de rango superior, en
       // este caso no se podrá ir más allá de 80, porque la
       // anchura del marcador es 20
       miBarra = new Scrollbar( Scrollbar.HORIZONTAL,50,20,0,100 );
       miBarra.setBlockIncrement( 15 );
       // Crea un objeto visual Campo de texto (TextField)
       // solamente para visualización, así que lo hacemos
       // no editable
       miTexto = new TextField( "Cadena Inicial" );
       miTexto.setEditable( false );
       // Incorporamos los componentes al objeto de tipo Frame
       miFrame.add( "North",miTexto );
       miFrame.add( "South",miBarra );
       miFrame.setVisible( true );
       // Se instancia un objeto para recibir los eventos de la
       // barra de desplazamiento y se registra para ser notificado
       // de los eventos de ajuste
       ProcesoBarra procesoBarra = new ProcesoBarra( miTexto );
       miBarra.addAdjustmentListener( procesoBarra );
```

```
        // Se instancia y registra un objeto receptor de eventos
        // para terminar la ejecucion del programa cuando el
        // usuario decida cerrar la ventana
        miFrame.addWindowListener( new Conclusion() );
        }
    }
// Receptor de eventos de Bajo-Nivel.
// Al contrario que otros componentes que disponen de su propio
// interfaz receptor, esta clase no puede implementar un interfaz
// del tipo ScrollbarListener porque no existe. En su lugar,
// implementa el interfaz AdjustmentListener que es el utilizado
// como receptor de los eventos de las barras de desplazamiento
class ProcesoBarra implements AdjustmentListener {
// A través de esta variable se indicará la psoición del
// indicador en la barra de desplazamiento
      TextField posicion;

      ProcesoBarra( TextField iPosicion ) {
            posicion = iPosicion;
          }
   public void adjustmentValueChanged( AdjustmentEvent evt ) {
            // Presentamos información sobre el evento en la pantalla
       System.out.println( evt );
       System.out.println( "Ajustable = "+evt.getAdjustable() );
       System.out.println( "Tipo de Ajuste = "+evt.getAdjustmentType() );
            // Presentamos el valor que representa la posición del
            // marcador en el campo de texto
            posicion.setText( "Valor = "+evt.getValue() );
       }
    }
// Receptor de eventos de Bajo-Nivel.
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Conclusion extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
       System.exit( 0 );
       }
   }
```

Cuando el marcador de la barra de desplazamiento se mueve, utilizando cualquiera de los cinco métodos físicos de movimiento de ese marcador, el valor de la barra, que representa la posición del marcador, se extrae del evento y se presenta en el campo de texto. Además, en cada movimiento del ratón, se presenta información adicional en la pantalla, como es el tipo de evento, el valor del desplazamiento, etc. Si se compila y ejecuta el ejemplo, aparecerá una ventana en la pantalla como la que reproduce la imagen que se muestra a continuación.

En esa imagen se puede observar una cosa que puede resultar un poco chocante, y es que el valor que indica el campo de texto es 50, justo la mitad del rango de la barra; sin embargo, el marcador de la barra está alineado con ese valor por su borde izquierdo. Este comportamiento parece diferente a lo que se acostumbra en otros entornos como Visual Basic o Delphi, en donde el valor corresponde al punto central del marcador. En este caso del JDK, su comportamiento hace que si tenemos un marcador de tamaño finito, los valores superiores no se podrán alcanzar jamás; en este caso, por ejemplo, la barra no podrá sobrepasar el valor 80, cuando su rango va de 0 a 100.



El código del ejemplo, los comentarios incluidos ahondan en el funcionamiento y control de la barra de desplazamiento.

Se puede observar que hay cinco parámetros diferentes que se pueden fijar en el constructor de la barra de desplazamiento, aunque hay disponibles algunos más en la inicialización, como es el BlockIncrement, que no está incluido en la lista de parámetros. Este parámetro debe fijarse utilizando el método setBlockIncrement(). Debería haber valores por defecto para todos estos parámetros, pero no se localiza nada al respecto en la documentación de Sun.

Reiterando lo citado en la presentación del ejemplo, el objeto receptor de eventos de la barra de desplazamiento no está añadido como un ScrollbarListener, sino que está definido y luego añadido como un AdjustmentListener.

Movimientos del ratón

En secciones anteriores ya se han presentado los eventos del ratón y se indicó que hay dos interfaces diferentes para tratarlos: MouseListener y MouseMotionListener.

En el ejemplo Java1110 que ilustrará esta sección, , se verán solamente los métodos del interfaz MouseMotionListener: mouseDragged() y mouseMoved(), para mover un objeto Label sobre un objeto Panel, pinchando y arrastrando la Etiqueta. También se utilizarán los eventos mousePressed() y mouseReleased() del interfaz MouseListener. No se intenta hacer una demostración de la técnica de arrastrar-y-soltar; sino que solamente se pretende mostrar cómo instanciar, registrar, recibir y utilizar eventos declarados en el interfaz MouseMotionListener.

```
/**
* Este ejemplo permite desplazar una etiqueta por la ventana que se
* genera. Se utilizan los eventos de posicionamiento del rattón,
* tanto cuando se está arrastrando como cuando se ha movido de
* sitio el puntero
*/
import java.awt.*;
import java.awt.event.*;
public class java1110 {
   public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
```

```
        }
    }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
    // Variables de referencia pasadas como parámetros
    Label miEtiqueta;
    LabelInfo miEtiquetaInfo;
       // Posición inicial de la etiqueta
       int posicionX = 50;
       int posicionY = 25;
    public IHM() {
        // Crea un objeto visual Etiqueta (Label)
        miEtiqueta = new Label( "Etiqueta" );
        miEtiqueta.setBackground( Color.yellow );
        // Crea un objeto Panel para contener la etiqueta
        Panel miPanel = new Panel();
        miPanel.setLayout( null );
        miPanel.add( miEtiqueta );
             // Fijamos la posición y ancho y alto de la etiqueta
             miEtiqueta.setBounds( posicionX,posicionY,125,100 );
        // Crea un objeto visual Frame para contenerlo todo
        Frame miFrame = new Frame();
        miFrame.setSize( 300,200 );
        miFrame.setTitle( "Tutorial de Java, Eventos" );
        miFrame.add( "Center",miPanel );
        miFrame.setVisible( true );
        // Crea un objeto para mantener información acerca de la
             // etiqueta
        miEtiquetaInfo = new LabelInfo();
        miEtiquetaInfo.labelX = posicionX;
        miEtiquetaInfo.labelY = posicionY;
        // Se instancia y registra un objeto MouseMotionListener
        // y otro MouseListener que procesarán los eventos de
        // movimiento del ratón y de los botones del ratón
        miEtiqueta.addMouseMotionListener(
                    new ProcesoMovimientoRaton( miEtiquetaInfo ) );
        miEtiqueta.addMouseListener(
                    new ProcesoRaton( miEtiquetaInfo,miEtiqueta ) );
        // Se instancia y registra un objeto receptor de eventos
        // para terminar la ejecucion del programa cuando el
        // usuario decida cerrar la ventana
        Conclusion procesoVentana = new Conclusion();
        miFrame.addWindowListener( procesoVentana );
        }
    }
// Esta es una clase auxiliar que se utiliza para guardar las
// posiciones en que se encuentra la etiqueta y también las
// coordenadas en las que se mueve el ratón y las coordenadas
// en que se encontraba el cursor al pulsar uno de los botones
// del ratón
class LabelInfo {
       int labelX;
       int labelY;
       int ratonPulsadoX;
       int ratonPulsadoY;
       int ratonMovidoX;
       int ratonMovidoY;
    }
// Esta clase receptor se utiliza para monitorizor los eventos
// de pulsacion de los botones del raton y los eventos que se
// producen al soltar el botón. Los dos eventos se utilizan para
// determinar la posición inicial y final en que se encuentra la
// etiqueta
class ProcesoRaton extends MouseAdapter {
```

```
   LabelInfo labelInfo;
      Label label;
      ProcesoRaton( LabelInfo iLabelInfo,Label iLabel ) {
            labelInfo = iLabelInfo;
            label = iLabel;
          }
   public void mousePressed( MouseEvent evt ) {
            // Guarda la posición inicial en que se pulsa el ratón
            labelInfo.ratonPulsadoX = evt.getX();
            labelInfo.ratonPulsadoY = evt.getY();
       }
      // Al soltar el botón se calcula la nueva posición de la
      // etiqueta, en función de su posición inicial, el punto en
      // que se encontraba el ratón al pulsar el botón y el punto
      // final cuando se suelta el botón. Hay pulsar y soltar dentro
      // de la etiqueta, sino no se realiza ningún movimiento
   public void mouseReleased( MouseEvent evt ) {
            int nuevaPosicionX = labelInfo.labelX +
                  labelInfo.ratonMovidoX - labelInfo.ratonPulsadoX;
            int nuevaPosicionY = labelInfo.labelY +
                  labelInfo.ratonMovidoY - labelInfo.ratonPulsadoY;
            // Movemos la etiqueta a la nueva posición
            label.setLocation( nuevaPosicionX,nuevaPosicionY );
            // Guarda la posición actual de la etiqueta
            labelInfo.labelX = nuevaPosicionX;
            labelInfo.labelY = nuevaPosicionY;
       }
   }
// Esta clase receptor se utiliza para monitorizor los eventos
// de movimiento del raton
class ProcesoMovimientoRaton extends MouseMotionAdapter {
   LabelInfo labelInfo;
      ProcesoMovimientoRaton( LabelInfo iLabelInfo ) {
            labelInfo = iLabelInfo;
          }
   public void mouseDragged( MouseEvent evt ) {
            System.out.println( "Arrastre = "+evt );
            // Guarda la posición durante el movimiento del ratón
            labelInfo.ratonMovidoX = evt.getX();
            labelInfo.ratonMovidoY = evt.getY();
       }
   public void mouseMoved( MouseEvent evt ) {
            System.out.println( "Movido = "+evt );
          }
   }
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del Frame
class Conclusion extends WindowAdapter {
   public void windowClosing( WindowEvent evt ) {
       System.exit( 0 );
       }
   }
```

El interfaz MouseMotionListener declara dos métodos:

```
public abstract void mouseDragged( MouseEvent evt )
public abstract void mouseMoved( MouseEvent evt )
```

El primer método es invocado cuando uno de los botones del ratón es pulsado sobre un componente y luego arrastrado. Según la documentación, los eventos de arrastre del ratón continuarán siendo enviados al Componente donde se haya originado en primer lugar, hasta que el botón se suelte, considerando que la posición del ratón esté dentro de los

límites del Componente. Sin embargo, si se inserta una sentencia println() en el ejemplo y se observa la salida, se puede comprobar que los eventos de arrastre dejan de generarse tan pronto como el ratón abandona los límites del objeto para el cual se ha registrado el receptor. Además, si el ratón vuelve a entrar dentro de los límites del objeto, no se recupera la emisión de eventos. Como este comportamiento no está claro, no sé si es un bug, un error en la documentación o simplemente una mala interpretación de la documentación por mi parte.

El segundo método es invocado cuando el ratón se mueve sobre un Componente, sin ningún botón pulsado. Si también se coloca una llamada a println() en el método mouseMoved() del ejemplo, se puede observar que se generan eventos cuando el ratón se mueve dentro de la Etiqueta y dejan de generarse cuando el ratón abandona los límites de esa Etiqueta. El tren de eventos vuelve a producirse cuando el ratón reentra en el campo de acción de la Etiqueta.

El programa ejemplo Java1110, utiliza una combinación de eventos mousePressed(), mouseReleased() y mouseDragged() para implementar una forma muy cruda de la técnica de arrastrar-y-soltar.

Se coloca una Etiqueta amarilla sobre un Panel que está situado sobre un Frame. Un objeto MouseListener y otro MouseMotionListener son instanciados y registrados para recibir eventos del ratón generados sobre el objeto Label.

El usuario puede cambiar la posición del objeto Label pulsando el botón con el cursor dentro de la Etiqueta y arrastrándolo. El camino que siga el ratón debe comenzar y terminar dentro de los límites de la Etiqueta y no debe abandonar esos límites en ningún momento. La etiqueta se mueve la cantidad de pixels que resulte de la diferencia neta entre los puntos final e inicial de la posición del ratón.

La posición inicial está determinada por las coordenadas del evento mousePressed() y el punto final está fijado por las coordenadas finales que envía el evento mouseDragged().

El propósito de este ejemplo es simplemente demostrar el uso del interfaz MouseMotionListener, en comparación con el interfaz MouseListener. No se ha pretendido que sea un programa que sirva de base para ejemplos de arrastrar-y-soltar. Hay una gran cantidad de código adicional que sería necesario incorporar a este programa para convertirlo en un verdadero ejemplo de este tipo de técnica, aunque resulta mucho más sencillo utilizar las clases incorporadas en el JDK 1.2 para estos menesteres.

El programa produce una salida por pantalla que va mostrando el tren de eventos que está siendo generado por los métodos sobrecargados mouseDragged() y mouseMoved(). Si el lector observa detenidamente esa información mientras mueve o arrastra el ratón, tendrá una idea bastante clara de cómo está funcionando el sistema de eventos.

Eventos generados por el usuario

Hasta ahora se han descrito los eventos que pueden generar los Componentes que se integran en in interfaz gráfico. Ahora se va a abordar la creación y lanzamiento de

eventos bajo el control del programa que se está ejecutando, que producirán las mismas respuestas que si los eventos tuviesen su origen en alguno de los Componentes del interfaz.

Aunque esta técnica no es imprescindible para entender el funcionamiento del Modelo de Delegación de Eventos, sí es un material crítico a la hora de entrar en el estudio de los componentes Lightweight, o componentes que no necesitan tener un soporte en la plataforma en que se ejecuta la aplicación, por lo que es necesario sentar sólidamente las bases del entendimiento de los eventos generados por programa para luego atacar el uso de los componentes Lightweight. Además, también esto servirá de mucho para entender más fácilmente lo que está sucediendo realmente cuando se implemente el Modelo de Delegación de Eventos utilizando componentes visuales desde el AWT.

Para poder utilizar los eventos generados por programa con las técnicas que se van a describir, será necesario definir una clase que sea capaz de generar este tipo de eventos. Aquí se centrará el estudio en los eventos de tipo Action, aunque no hay razón alguna para que la misma técnica se aplique a eventos de bajo nivel como puedan ser los eventos de ratón o del teclado.

La clase en cuestión debe ser una subclase de Component y, al menos, debe incluir los siguientes tres miembros:

- Una variable de instancia que es una referencia a la lista de objetos Listener registrados. En el ejemplo siguiente, Java1111, estos objetos receptores de eventos son de tipo ActionListener. La variable de instancia es de tipo ActionListener y puede contener una referencia a un solo objeto de este tipo o una referencia a una lista de objetos de este tipo.

- Un método para crear la lista anterior, que en el ejemplo es generaListaReceptores(), que se llama a sí para ilustrar que el nombre no es técnicamente importante, aunque por consistencia con la documentación del Modelo de Delegación de Eventos, debiera llamarse addActionListener(). Esta lista ha de ser generada a través de una llamada al método AWTEventMulticaster.add(), que devuelve una referencia a la lista.

- Un método que invocará al método correspondiente al tipo de evento en la clase Listener de la lista de objetos receptores de los eventos. En el ejemplo, los objetos receptores son de tipo ActionListener, así que el método en cuestión es actionPerformed(), y el método que lo invoca es generaEventoAction(). En este ejemplo solamente hay un objeto receptor, luego se verá otro en el que habrá varios objetos receptores de este tipo de eventos en la lista.

En el programa Java1111, se instancia un solo objeto de la clase NoVisualizable. También se define un solo objeto de la clase ActionListener, que es instanciado y registrado para recibir objetos de tipo Action generados sobre el objeto NoVisualizable.

La generación del evento Action se produce al invocar el método generaEventoAction() del objeto de la clase NoVisualizable. Este evento es atrapado y procesado por el objeto ActionListener, y como resultado del procesado aparecerá en pantalla la información siguiente:

```
% java java1111
Tutorial de Java, Eventos
```

```
metodo actionPerformed() invocado sobre ObjetoNoVisual
```

El listado del código completo de este ejemplo es el que se reproduce a continuación.

```
/**
* Este es un ejemplo muy sencillo que muestra como se pueden
* generar eventos desde programa en el modelo de Delegacion
* de Eventos
*/
import java.awt.*;
import java.awt.event.*;
public class java1111 {
   public static void main( String args[] ) {
        // Se instancia un objeto de este tipo
        new java1111();
        }
   // Constructor
   public java1111() {
        System.out.println( "Tutorial de Java, Eventos" );

        NoVisualizable objNoVisual=new NoVisualizable( "ObjetoNoVisual");
        objNoVisual.generaListaReceptores( new ClaseActionListener() );
        objNoVisual.generaEventoAction();
        }
   }
// Clase para responder a los eventos de accion
class ClaseActionListener implements ActionListener {
   public void actionPerformed( ActionEvent evt ) {
        System.out.println(
            "metodo actionPerformed() invocado sobre " +
            evt.getActionCommand() );
        }
   }
// Clase para crear un objeto que sea capaz de generar eventos de
// accion (Action)
class NoVisualizable extends Component {
   // La identificacion del objeto
   String ID;
   // Lista de los objetos receptor registrados
   ActionListener receptorAction;
   // Constructor para el objeto No Visual
   public NoVisualizable( String ID ) {
        this.ID = ID;
        }
   public void generaListaReceptores( ActionListener listener ) {
        receptorAction = AWTEventMulticaster.add(
            receptorAction,listener );
        }
   public void generaEventoAction() {
        receptorAction.actionPerformed(
            new ActionEvent( this,ActionEvent.ACTION_PERFORMED,ID ) );
        }
   }
```

Es interesante repasar un poco más detalladamente el funcionamiento del programa y el código que lo implementa. Y, empezando por el constructor, se observa que instancia un objeto del tipo NoVisualizable, registra un objeto receptor de eventos sobre ese objeto NoVisualizable e invoca al método que hace que el objeto NoVisualizable genere un evento de tipo Action. Todo esto se hace en las líneas siguientes:

```
NoVisualizable objNoVisual = new NoVisualizable( "ObjetoNoVisual" );
objNoVisual.generaListaReceptores( new ClaseActionListener() );
objNoVisual.generaEventoAction();
```

La siguiente sentencia es la declaración de la variable de instancia en la definición de la clase NoVisualizable que referenciará la lista de objetos Listener registrados, que como ya se ha indicado puede apuntar a uno o a varios objetos de tipo ActionListener.

```
ActionListener receptorAction;
```

En la línea de código que se reproduce a continuación se encuentra la sentencia que construye la lista de objetos Listener registrados añadiendo un nuevo objeto a esa lista. La primera vez que se ejecuta la sentencia en el programa, devuelve una referencia al objeto que se añade. Cuando se ejecute posteriormente, devolverá una referencia a la lista de objetos que está siendo mantenida separadamente.

```
receptorAction = AWTEventMulticaster.add( receptorAction,listener );
```

En este caso, solamente se añade un objeto a la lista y si se examina la referencia que devuelve el método add() se observará que es la referencia a un solo objeto.

La última sentencia interesante de este ejemplo es la invocación del método actionPerformed() del objeto ActionListener, o más propiamente, la invocación de este método en todos los objetos que se encuentren registrados en la lista. Afortunadamente, todo lo que hay que hacer es invocar el método sobre la referencia y el sistema se encarga de realizar la invocación en cada uno de los objetos que están integrados en la lista. Esta es la característica principal de la clase AWTEventMulticaster.

```
receptorAction.actionPerformed( new ActionEvent( this,
ActionEvent.ACTION_PERFORMED ,ID ) );
```

Y a esto es a lo que se reduce, en esencia, la generación de eventos desde programa. Aunque se puede complicar todo lo complicable, como se muestra en el ejemplo siguiente, Java1112, que ya tiene un poco más de consistencia y está destinado a ilustrar la capacidad de la clase AWTEventMulticaster de despachar eventos a más de un objeto a la vez, en concreto a todos los que se encuentren registrados en la lista de objetos Listener.

```
/**
* Este es un ejemplo mas consistente que ilustra la forma en que se
* generan eventos desde Programa.
* Se generan dos objetos no visibles que son capaces de enviar eventos
* de tipo Action
*/
import java.awt.*;
import java.awt.event.*;
public class java1112 {
   public static void main( String args[] ) {
       new java1112();
       }
   public java1112() {
       System.out.println( "Tutorial de Java, Eventos" );
       System.out.println( "Instancia dos objetos NoVisualizable que son
" + "capaces de generar eventos de tipo Action." );
       NoVisualizable aNoVisualizable = new NoVisualizable(
"ObjNoVisualizable A" );
       aNoVisualizable.setName( "ObjNoVisualizableA" );
       NoVisualizable bNoVisualizable = new NoVisualizable(
"ObjNoVisualizable B" );
       bNoVisualizable.setName( "ObjNoVisualizableB" );

       System.out.println( "Nombre del primer objeto NoVisualizable:  " +
aNoVisualizable.getName() );
       System.out.println( "Nombre del segundo objeto NoVisualizable: " +
bNoVisualizable.getName() );
```

```
      // Se registran objetos ActionListener para que recojan los
      // eventos que se generen sobre los objetos NoVisualizable. Uno de
      // los objetos NoVisualizable es registrado solamente sobre un
      // objeto ActionListener, mientras que el otro se registra
      // sobre dos objetos ActionListener, uno de los cuales es
      // el mismo que se registro para el otro objeto NoVisualizable
       System.out.println(
           "Registra objetos ActionListener sobre los objetos
NoVisualizable" );
       aNoVisualizable.addActionListener( new PrimerReceptorAction() );
       bNoVisualizable.addActionListener( new PrimerReceptorAction() );
       bNoVisualizable.addActionListener( new SegundoReceptorAction() );
       // Se hace que cada uno de los dos objetos NoVisualizable generen
       // un evento Action
       System.out.println( "Metodo generaEventoAction() invocado " +
           " sobre el objeto " + aNoVisualizable.getName() );
       System.out.println(
           " que solo tiene un objeto ActionListener registrado." );
       aNoVisualizable.generaEventoAction();
       System.out.println();
       System.out.println( "Metodo generaEventoAction() invocado " +
           " sobre el objeto " + bNoVisualizable.getName() );
       System.out.println(
           " que tiene dos objetos ActionListener registrados." );
       bNoVisualizable.generaEventoAction();
       }
   }
// Las dos clases siguientes son clases ActionListener estandar. Los
// objetos instanciados de estas clases simplemente capturan los
// eventos de tipo Action y presentan informacion en pantalla sobre
// ellos
// Primera clase para responder a los eventos action
class PrimerReceptorAction implements ActionListener {
   public void actionPerformed( ActionEvent evt ) {
       System.out.println(
           "En el metodo actionPerformed() del objeto " +
           "PrimerReceptorAction" );
       System.out.println(
           "Metodo actionPerformed() invocado sobre " +
           evt.getActionCommand() );
       }
   }

// Segunda clase para responder a los eventos action
class SegundoReceptorAction implements ActionListener {
   public void actionPerformed( ActionEvent evt ) {
       System.out.println(
           "En el metodo actionPerformed() del objeto " +
           "SegundoReceptorAction" );
       System.out.println(
           "Metodo actionPerformed() invocado sobre " +
           evt.getActionCommand() );
       }
   }
// Clase para crear un objeto que sea capaz de generar eventos de
// accion (Action)
class NoVisualizable extends Component {
   // El estado de un objeto NoVisualizable en cualquier momento se
   // puede conocer a traves de sus variables de instancia
   // La identificacion del objeto
   String ID;
   // Lista de los objetos receptor registrados
   ActionListener receptorAction;
   // estos objetos seran notificados cuando se produzca
   // un evento Action (ver la descripcion de addActionListener)
```

```
    // Constructor para el objeto No Visual
    public NoVisualizable( String ID ) {
        this.ID = ID;
        }

    //-------------------------------------------------------------
    // El entorno de un objeto NoVisualizable esta definido por los
    // siguientes metodos de instancia
    //-------------------------------------------------------------
    /**
     * El siguiente metodo incorpora los objetos ActionListener pasados
     *  como parametro a la lista de objetos ActionListener designados
     *  como receptores que deben ser notificados cuando se produzca
     *  un evento sobre un objeto NoVisualizable
     * La notificacion tiene lugar en un metodo diferente invocando
     *  al metodo actionPerformed() de cada uno de los objetos
     *  ActionListener de la lista
     * Se incorporan nuevos objetos a la lista llamando al metodo
     *  estatico add() de la clase java.awt.AWTEventMulticaster y
     *  pasandole la variable de instancia que referencia a la lista
     *  a la que se quiere incorporar el nuevo receptor
     * Para el primer objeto receptor de eventos incorporado a la
     *  lista, se devuelve una referencia a si mismo. Luego, en
     *  este caso, la lista seria simplemente una referencia al
     *  objeto Receptor
     * Cuando se incorporan objetos receptores adicionales, el
     *  metodo add() de la clase AWTEventMulticaster devuelve una
     *  referencia a un objeto de tipo java.awt.AWTEventMulticaster
     * Segun la documentacion del JDK sobre esta clase, se dice:
     *  "Esta clase manejara la estructura de una cadena de
     *  receptores de eventos y despachara eventos a esos
     *  receptores"
     * Cuando el metodo actionPerformed() es invocado posteriormente
     *  sobre la referencia a la lista, oel metodo es invocado sobre
     *  un objeto simple, o el objeto AWTEventMulticaster asume la
     *  responsabilidad de invocar a todos los metodos actionPerformed()
     *  de todos los objetos Listener que se mantienen en su lista de
     *  objetos receptores de esos eventos
     */
    public void addActionListener( ActionListener receptor ) {
        System.out.println();
        System.out.println( "Invocado el metodo addActionListener()" );
        System.out.println( ID + ":   El Receptor incorporado es: " +
            receptor );
        System.out.println(
            "Invocado AWTEventMulticaster.add() para recuperar " +
            "la referencia a ActionListener");
        receptorAction = AWTEventMulticaster.add( receptorAction,
            receptor);
        System.out.println( ID + ":   La Ref a ActionListener es: " +
            receptorAction );
        System.out.println();
        }
    // El proposito de este metodo es invocar al metodo actionPerformed()
    // sobre todos los objetos Receptor que estan contenidos en la lista
    // de objetos Receptor que estan registrados para recibir eventos
    // Action que estan siendo generados por este objeto NoVisualizable.
    // Esto se consigue invocando el metodo actionPerformed() sobre la
    // referencia a la lista. Cuando esto esta hecho, un objeto
ActionEvent
    // es instanciado y pasado como parametro
    public void generaEventoAction() {
        // Confirma que hay algun receptor registrado
        if( receptorAction != null ) {
            System.out.println(
```

```
                "En el metodo generaEventoAction(), lanzando " +
                "un evento ACTION_PERFORMED a " );
            System.out.println( receptorAction + " para " + ID );
            receptorAction.actionPerformed( new ActionEvent(
                this,ActionEvent.ACTION_PERFORMED,ID ) );
            }
        }
    }
```

En este ejemplo se registran dos objetos diferentes ActionListener sobre un solo objeto NoVisualizable, con lo cual la clase AWTEventMulticaster deberá lanzar eventos Action a dos objetos Listener distintos. El código del ejemplo es el que se muestra a continuación y le sigue un repaso detallado de las sentencias de código que resultan más interesantes.

La mayor parte del código del ejemplo son simples sentencias de impresión de información por pantalla, para explicar qué es lo que está sucediendo durante la ejecución del programa, así que solamente se revisará a continuación las sentencias que resultan interesantes para comprender el funcionamiento y utilización de la clase AWTEventMulticaster.

Creación de eventos propios

Tras la amplia introducción a la clase AWTEventMulticaster que se ha visto, se puede atacar la creación y envío de eventos propios, que esencialmente consta de las siguientes fases:

- Definir una clase para que los objetos del nuevo tipo de evento puedan ser instanciados. Debería extender la clase EventObject.
- Definir un interfaz EventListener para el nuevo tipo de evento. Este interfaz debería ser implementado por las clases Listener del nuevo tipo y debería extender a EventListener.
- Definir una clase Listener que implemente el interfaz para el nuevo tipo de evento.
- Definir una clase para instanciar objetos capaces de generar el nuevo tipo de evento. En el ejemplo que se verá a continuación, se extiende la clase Component.
- Definir un método add<nombre_del_evento>Listener() para que pueda mantener la lista de objetos Listener registrados para recibir eventos del nuevo tipo.
- Definir un método que envíe un evento del nuevo tipo a todos los objetos Listener registrados en la lista anterior, invocando al método de cada uno de esos objetos. Este método es un método declarado en el interfaz EventListener para el nuevo tipo de evento.

El modelo de Delegación de Eventos puede soportar el registro de múltiples objetos Listener sobre un solo objeto generador de eventos. El modelo también puede soportar el registro de objetos desde una sola clase Listener sobre múltiples objetos generadores de eventos. Y también, el modelo puede soportar combinaciones de los dos.

En la sección anterior se presentó el método add() de la clase AWTEventMulticaster para crear y mantener una lista de objetos Listener registrados. Desafortunadamente, este método no puede utilizarse directamente para mantener una lista de objetos registrados para los eventos creados por el usuario, porque no hay una versión sobrecargada del método cuya signature coincida con el evento nuevo.

Una forma de crear y mantener estas listas de objetos registrados es embarcarse en un ejercicio de programación de estructuras de datos en una lista. Hay ejemplos de esto en algunos libros, pero aquí no se va a ver, porque parece más interesante una aproximación distinta, que consistirá en la construcción de subclases de AWTEventMulticaster y sobrecargar el método add() para que tenga una signature que coincida con el evento que se está creando. Por supuesto, el cuerpo del método sobrecargado todavía tiene que proporcionar código para porcesar la lista, lo cual lo hace de nuevo complicado. Así que, el siguiente ejemplo, se limita a tener un solo objeto registrado, para evitar la dificultad añadida que supondría, en un primer contacto por parte del lector con este tipo de eventos, el encontrarse con el código de proceso de la lista de eventos.

El programa de ejemplo, Java1113, muestra como se crean, se capturan y se procesan eventos creados por el programador, capacidad que ya se ha introducido en la sección anterior.

```
/**
 * Este ejemplo muestra una forma sencilla de crear, capturar y
 * procesar eventos propios creados por el programador
 */
import java.awt.*;
import java.awt.event.*;
import java.util.EventListener;
import java.util.EventObject;
public class java1113 {
   public static void main(String[] args){
        // Se intancia un objeto de este tipo
        new java1113();
        }
   // Constructor de la clase
   public java1113(){
        System.out.println( "Tutorial de Java, Eventos" );
        NoVisual primerObjNoVisual = new NoVisual( "Primer ObjetoNoVisual"
);
       primerObjNoVisual.addMiEventoListener(new MiClaseEventListener());
        // Crea un evento
        primerObjNoVisual.generarMiEvento();
        NoVisual segundoObjNoVisual = new NoVisual( "Segundo
ObjetoNoVisual" );
        segundoObjNoVisual.addMiEventoListener( new MiClaseEventListener()
);
        // La siguiente sentencia hace que el programa termine con el
        // mensaje "No se soportan multiples Receptores" porque se
    // viola la restriccion que establece este programa de que solamente
        // haya un objeto receptor registrado para recoger sus eventos.
        // La sentencia esta deshabilitada mediante un comentario y se
        // muestra solo a efectos de informacion
        //
   // segundoObjNoVisual.addMiEventoListener(newMiClaseEventListener());
       // Crea un evento
        segundoObjNoVisual.generarMiEvento();
        }
   }
// Clase para definir un nuevo tipo de evento
class MiEvento extends EventObject {
```

```
   // Variable de instancia para diferencia a cada objeto de este tipo
   String id;
   // Constructor parametrizado
   MiEvento( Object obj,String id ) {
        // Se le pasa el objeto como parametro a la superclase
        super( obj );
        // Se guarda el identificador del objeto
        this.id = id;
        }
   // Metodo para recuperar el identificador del objeto
   String getEventoID() {
        return( id );
        }
   }
// Define el interfaz para el nuevo tipo de receptor de eventos
interface MiEventoListener extends EventListener {
   void capturarMiEvento( MiEvento evt );
   }
// Clase Receptor para responder a los eventos que se crean
class MiClaseEventListener implements MiEventoListener {
   public void capturarMiEvento( MiEvento evt ) {
        System.out.println(
            "Metodo capturarMiEvento() invocado sobre " +
            evt.getEventoID() );
        System.out.println(
            "El origen del evento fue " + evt.getSource() );
        }
   }
// Clase para crear eventos de nuestro tipo. Esta es una version
// muy sencilla que solo permite que se registre un Receptor
// para el tipo de evento que hemos creado
class NoVisual extends Component {
   // El identificador de este objeto
   String ID;
   // Referencia al receptor unico
   MiClaseEventListener miReceptor;
   // Construye un objeto NoVisual
   public NoVisual( String ID ) {
        this.ID = ID;
        }
   public void addMiEventoListener( MiClaseEventListener receptor ) {
        // No se permite que intente incorporar mas de un receptor
        if( miReceptor == null )
            miReceptor = receptor;
        else {
            System.out.println( "No se soportan multiples Receptores" );
            // Se sale, si se intentan registrar varios objetos Receptor
            System.exit( 0 );
            }
        }
   public void generarMiEvento() {
        miReceptor.capturarMiEvento( new MiEvento( this,ID ) );
        }
   }
```

Se define una clase NoVisual, cuyos objetos son capaces de generar eventos del nuevo tipo que se va a crear siguiendo los pasos que se han indicado en párrafos anteriores. En el programa, se instancian dos objetos de la clase NoVisual y se define una clase MiClaseEventListener que implementa el interfaz MiEventoListener y define un método para procesar los objetos de tipo MiEvento llamando al método capturarMiEvento().

Los objetos de la clase MiClaseEventListener son instanciados y registrados para recibir eventos propios desde los objetos NoVisual, es decir, se instancian y registran

objetos desde una sola clase Listener sobre múltiples objetos generadores de eventos. El objeto NoVisual contiene un método de instancia llamado generarMiEvento(), diseñado para enviar un evento del nuevo tipo al objeto Listener registrado sobre el objeto fuente al invocar el método capturarMiEvento() del objeto Listener.

El método generarMiEvento() es invocado sobre ambos objetos NoVisual, haciendo que se generen los eventos y que sean capturados y procesados por los objetos registrados de la clase MiClaseEventListener. El procesado del evento es muy simple. El identificador y la fuente de la información se extraen del objeto MiEvento que es pasado como parámetro al método capturarMiEvento() y se presenta esta información en la pantalla.

Como en ejemplos anteriores, ahora es el momento de revisar y comentar las líneas de código que resultan más interesantes, o que son nuevas, y merecen un repaso de entre todo el código del ejemplo. El primer fragmento de código en el que hay que reparar es la definición de la clase desde la cual se pueden instanciar los eventos del nuevo tipo. Esta clase extiende la clase EventObject y se reproduce a continuación.

```
class MiEvento extends EventObject {
    // Variable de instancia para diferencia a cada objeto de este tipo
    String id;

    // Constructor parametrizado
    MiEvento( Object obj,String id ) {
        // Se le pasa el objeto como parametro a la superclase
        super( obj );
        // Se guarda el identificador del objeto
        this.id = id;
        }
    // Método para recuperar el identificador del objeto
    String getEventoID() {
        return( id );
        }
    }
```

El constructor de la clase recibe dos parámetros. El primero es una referencia al objeto que ha generado el evento, que es pasada al constructor de la superclase, EventObject, donde se almacena para poder acceder a ella a través del método getSource() de esa superclase EventObject. El segundo parámetro es una cadena de identificación proporcionada al constructor cuando el objeto MiEvento es instanciado. En este ejemplo, este identificador es simplemente un String almacenado en una variable de instancia en el objeto fuente cuando se instancia, pero se puede pasar cualquier identificación o comando de este modo. La clase también proporciona un método para poder recuperar esa identificación de la variable de instancia del objeto.

El siguiente código interesante es la definición del nuevo interfaz Listener llamado MiEventoListener que extiende a EventListener. Este interfaz declara el método capturarMiEvento() que es el principal en el procesado de eventos de este tipo.

```
interface MiEventoListener extends EventListener {
    void capturarMiEvento( MiEvento evt );
    }
```

El código que se reproduce a continuación es el que implementa el interfaz anterior y define una clase receptora de eventos del nuevo tipo. La clase Listener sobreescribe el evento capturarMiEvento() declarado en el interfaz y utiliza los métodos de la superclase

para acceder y presentar en pantalla la identificación del evento y del objeto que lo ha generado.

```
class MiClaseEventListener implements MiEventoListener {
    public void capturarMiEvento( MiEvento evt ) {
        System.out.println(
            "Metodo capturarMiEvento() invocado sobre " +
            evt.getEventoID() );
        System.out.println(
            "El origen del evento fue " + evt.getSource() );
        }
    }
```

En el fragmento que sigue es donde se define la clase NoVisual que extiende la clase Component y mantiene dos variables de instancia; una de ellas es el identificador String inicializado por el constructor de la clase y la otra es una referencia al objeto Listener registrado sobre la fuente de eventos.

```
class NoVisual extends Component {
    // El identificador de este objeto
    String ID;
    // Referencia al receptor unico
    MiClaseEventListener miReceptor;
```

El código siguiente es el que registra un objeto Listener y que, como se puede observar es muy similar al de ejemplos anteriores, excepto que no se utiliza el método add() de la clase AWTEventMulticaster para crear y mantener la lista de objetos registrados. Este código soporta solamente un objeto Listener registrado y contiene la lógica para concluir la ejecución si se intenta registrar más de un objeto receptor de eventos.

```
public void addMiEventoListener( MiClaseEventListener receptor ) {
    // No se permite que intente incorporar mas de un receptor
    if( miReceptor == null )
        miReceptor = receptor;
    else {
        System.out.println( "No se soportan multiples Receptores" );
        // Se sale, si se intentan registrar varios objetos Receptor
        System.exit( 0 );
        }
    }
```

Y ya sólo resta repasar el código que corresponde al método que genera el evento del nuevo tipo. En este caso, la generación de eventos va acompañada de la invocación del método capturarMiEvento() del objeto Listener registrado al que se le pasa un objeto de tipo MiEvento como parámetro, que es la forma típica de lanzar eventos en el nuevo Modelo de Delegación de eventos del JDK.

```
public void generarMiEvento() {
    miReceptor.capturarMiEvento( new MiEvento( this,ID ) );
    }
```

La cola de eventos del sistema

Una de las cosas más complicadas de entender del nuevo modelo de gestión de eventos es la creación de eventos y el envío de estos eventos a la cola de eventos del Sistema, para que lo lance a un determinado componente. El concepto en sí no es difícil, sin embargo, como la documentación es escasa, se vuelve una ardua tarea el entender el funcionamiento del sistema en esta circunstancia.

En este modelo para la creación y envío de eventos a la cola del Sistema, solamente es necesario invocar a un método, tal como muestra la siguiente sentencia:

```
Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
    new MouseEvent( miComponente,MouseEvent.MOUSE_CLICKED,0,0,-1,-
1,2,false ) );
```

Si se examina la sentencia de dentro hacia fuera, se observa que hay una instanciación de un nuevo objeto evento del tipo que se desea; se envía el objeto a la cola de eventos del Sistema que es devuelta por la invocación al método *getSystemEvent*Queue(), que es un método de Toolkit devuelto por la invocación al método getDefaultToolkit(), que es un método estático de la clase Toolkit.

Ahora se presenta el ejemplo Java1114 , en donde se utiliza la cola de eventos del Sistema para interceptar eventos del teclado y convertirlos en eventos del ratón.

```
/**
* Este ejemplo demuestra el uso del metodo postEvent() para
* colocar eventos en la cola de eventos del sistema, EventQueue
*/
import java.awt.*;
import java.awt.event.*;
public class java1114 extends Frame {
   public static void main( String args[] ) {
        java1114 ventana = new java1114();
        }
   public java1114() {
        MiComponente miComponente = new MiComponente();
        this.add( miComponente );
        setTitle( "Tutorial de Java, Eventos" );
        setSize( 250,100 );
        setVisible( true );
        // El siguiente objeto KeyListener, cuando recibe un evento del
        // teclado lo convierte en un evento de raton
        miComponente.addKeyListener( new MiKeyListener( miComponente ) );
        // Cuando se cierre el Frame, se concluye la aplicacion
        this.addWindowListener( new Conclusion() );
        }
   }
// Esta clase recibe los eventos del teclado sobre los componentes
// que se han creado. Cuando se atrapa un evento de teclado, el
// codigo del metodo keyPressed() presenta el caracter que
// corresponde a esa tecla (siempre que se pueda, porque hay
// teclas que no tienen un caracter visible asociado).
// Tambien crea un evento de raton artificail y lo coloca en
// la cola de eventos indicando que ha sido el mismo componente
// el que lo ha generado. De este modo, los objetos Receptores
// de este tipo convierten los eventos de teclado en eventos de
// raton
class MiKeyListener extends KeyAdapter {
   // Referencia al componente creado
   MiComponente miComponente;
   // Constructor parametrizado
   MiKeyListener( MiComponente miComp ) {
        // Guarda una referencia a nuestro componente
        miComponente = miComp;
        }
   // Metodo sobreescrito de pulsacion del teclado
   public void keyPressed( KeyEvent evt ) {
        System.out.println(
           "Metodo keyPressed(), tecla pulsada -> " + evt.getKeyChar() );
        // El parametro de identificacion en la construccion del
        // evento de raton debe ser un identificador de evento de
```

```
        // raton valido. Este evento se construye con las coordenadas
        // x e y a -1, para hacer que sea mas identificable. La
        // referencia al componente que se ha creado es el primer
        // parametro del constructor.
        Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
            new MouseEvent( miComponente,MouseEvent.MOUSE_CLICKED,
            0,0,-1,-1,2,false ) );
        }
    }
// En esta clase se crea un nuevo componente extendiendo un Label.
// Puede responder a eventos del teclado si se registra un
// objeto KeyListener adecuado sobre el. Sobreescribe el metodo
// processMouseEvent() para poder capturar y presentar en pantalla
// objetos MouseEvent creados y enviados por el objeto KeyListener
// con una referencia aun objeto de este tipo como el primer
// parametro del constructor del MouseEvent

class MiComponente extends Label {
    MiComponente() {
        this.setText( "Mi Componente" );
        // La siguiente sentencia es necesaria para provocar que
        // el metodo processMouseEvent() sea invocado siempre que
        // se encole un evento de raton para un objeto de este tipo
        enableEvents( AWTEvent.MOUSE_EVENT_MASK );
        }
    public void processMouseEvent( MouseEvent evt ) {
        // Se indica que se ha invocado a este metodo y se presenta
        // el identificador y las coordenadas del objeto MouseEvent
        // que se haya pasado como parametro
        System.out.println(
            "Metodo processMouseEvent(), MiComponente ID = " +
            evt.getID() + " " + evt.getPoint() );
        // SIEMPRE hay que hacer esto si se sobreescribe el metodo
        super.processMouseEvent( evt );
        }
    }
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Concluye la ejecucion del programa cuando se cierre la ventana
        System.exit( 0 );
        }
    }
```

El objeto principal del interfaz gráfico es instanciado desde la clase que extiende a la clase Frame. Se crea un componente propio extendiendo un Label. Los objetos de esta nueva clase así creada son capaces de responder a eventos del teclado y del ratón. Extendiendo la clase Label es posible sobreescribir el método processMouseEvent() de la clase Label que se proporciona a la clase MiComponente.

Los eventos del ratón son habilitados sobre los objetos de esta clase para que cualquier evento del ratón sea enviado al método processMouseEvent() y, como siempre en estos casos, hay que pasar el objeto MouseEvent al método del mismo nombre de la superclase antes de que termine el flujo del programa en el método.

Los clicks sobre el ratón con el cursor posicionado en el objeto que se ha creado son eviados al método processMouseEvent() en el cual lo que se hace es presentar la información que contiene el evneto en pantalla.

Los eventos del teclado, KeyEvent, son capturados por el receptor KeyListener. Cuando se captura un evento del teclado se genera un objeto MouseEvent sintético y se

coloca en la cola de eventos del Sistema. La documentación del JDK 1.2 para la creación de un objeto MouseEvent es la siguiente:

```
public MouseEvent(Component source,
                  int id,
                  long when,
                  int modifiers,
                  int x,
                  int y,
                  int clickCount,
                  boolean popupTrigger)
```

En el programa que nos ocupa, source es una referencia al objeto que se ha creado. Se asignan valores negativos a los parámetros x e y para que el objeto sea fácilmente reconocible cuando se genere, ya que un click real nunca podrá tener valores negativos en las coordenadas. Para el resto de parámetros se asignan valores aleatorios, excepto para id. Este parámetro es crítico a la hora de crear el objeto MouseEvent ya que debe coincidir con alguna de las constantes simbólicas que están definidas en la clase MouseEvent para los diferentes eventos del ratón que reconoce el Sistema. Estas constantes son:

MOUSE_FIRST Primer entero usado como id en el rango de eventos

MOUSE_LAST Ultimo entero usado como id en el rango de eventos

MOUSE_CLICKED

MOUSE_PRESSED

MOUSE_RELEASED

MOUSE_ENTERED

MOUSE_EXITED

MOUSE_DRAGGED

Si el identificador id asignado al nuevo evento no se encuentra en el rango anterior, no se produce ningún aviso ni se genera ninguna excepción, ni en la compilación ni en la ejecución del programa, simplemente no se envía ningún evento.

Cuando se ejecuta el programa aparece una ventana en la pantalla totalmente ocupada con el componente propio creado al extender la etiqueta. Si se muevo el ratón se producen mensajes normales en la pantalla del sistema generados por el método processMouseEvent(); pero si se pulsa una tecla, aparecerá un mensaje indicando la tecla pulsada y a continuación se indica que hay una llamada al método de procesado de eventos del ratón, al haberse generado un evento sintético provocado por la pulsación de la tecla. En las líneas siguientes se observa la salida por pantalla que se acaba de describir.

```
Metodo processMouseEvent(), MiComponente ID = 505 java.awt.Point
[x=246,y=54]
Metodo processMouseEvent(), MiComponente ID = 504 java.awt.Point
[x=239,y=28]
Metodo keyPressed(), tecla pulsada -> x
Metodo processMouseEvent(), MiComponente ID = 500 java.awt.Point [x=-
1,y=-1]
Metodo processMouseEvent(), MiComponente ID = 505 java.awt.Point
[x=246,y=47]
Metodo processMouseEvent(), MiComponente ID = 504 java.awt.Point
[x=227,y=23]
Metodo keyPressed(), tecla pulsada -> y
Metodo processMouseEvent(), MiComponente ID = 500 java.awt.Point [x=-
1,y=-1]
Metodo processMouseEvent(), MiComponente ID = 505 java.awt.Point
[x=188,y=96]
```

Echando un vistazo al código del programa se observa que hay gran parte ya vista en ejemplos anteriores; así que, siguiendo la tónica de otras secciones, se revisan a continuación los trozos de código que se han introducido nuevos o que merece la pena volver a ver. El primer fragmento son las sentencias en que se instancia un objeto del tipo MiComponente y se registra un objeto KeyListener para recibir los eventos del teclado sobre ese componente. Posteriormente, será el código de ese receptor KeyListener el que atrapará los objetos KeyEvent, creará objetos MouseEvent sintéticos y los enviará a la cola del Sistema como porvenientes de MiComponente.

```
MiComponente miComponente = new MiComponente();
this.add( miComponente );
. . .
miComponente.addKeyListener( new MiKeyListener( miComponente ) );
```

La sentencia siguiente es la ya vista al comienzo, en la que se proporcionan todos los parámetros necesarios para generar el evento del ratón sintético y enviarlo a la cola de eventos del Sistema. Entre esos parámetros está la referencia al componente creado, que en la descripción anterior era el parámetro source, luego está el parámetro id del nuevo evento, que es uno de la lista presentada antes y, finalmente, están las coordenadas x e y, que se fijan a –1 para poder reconocer fácilmente el evento sintético entre los mensajes que aparezcan por la pantalla.

```
Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent( new
MouseEvent( miComponente,MouseEvent.MOUSE_CLICKED,0,0,-1,-1,2,false ) );
```

Lo que ocurre aquí es que hay un gran problema de seguridad si se permite que applets no certificados manipulen libremente la cola de eventos del Sistema. Por ello, el método getSystemEventQueue() está protegido por comprobaciones de seguridad que impiden a los applets acceso directo a la cola de eventos.

Ahora se encuentra la sentencia que permite la generación de eventos y que está contenida en el constructor de la clase MiComponente. Esta sentencia es necesaria para poder invocar al método sobreescrito processMouseEvent() siempre que un evento de ratón sea enviado a un objeto de la clase MiComponente.

```
enableEvents( AWTEvent.MOUSE_EVENT_MASK );
```

Finalmente, para concluir el repaso al código del ejemplo, está el método sobreescrito processMouseEvent() que anuncia la invocación del método y presenta en pantalla información del id del evento y las coordenadas que contiene. La sentencia final es la llamada al mismo método de la superclase, si no se hace esta llamada, la salida probablemente no sea del todo correcta.

```
public void processMouseEvent( MouseEvent evt ) {
    // Se indica que se ha invocado a este metodo y se presenta
    // el identificador y las coordenadas del objeto MouseEvent
    // que se haya pasado como parametro
    System.out.println(
        "Metodo processMouseEvent(), MiComponente ID = " +
        evt.getID() + " " + evt.getPoint() );
    // SIEMPRE hay que hacer esto si se sobreescribe el metodo
    super.processMouseEvent( evt );
    }
```

Y el resto del código es el mismo que el de muchos de los ejemplos anteriores.

Intercambio de componentes

En este caso, se va a tratar de hacer que un objeto Label funcione como un objeto Button, provocando que genere eventos de tipo ActionEvent atribuibles a un botón; es decir, los eventos serán enviados a un objeto receptor ActionListener registrado sobre un Button, donde serán procesados como si fuesen eventos originados por ese objeto Button, cuando en realidad su origen es el objeto Label.

Al contrario que en el ejemplo anterior, en este ejemplo, Java1115, no se va a sobreescribir ningún método processXxxEvent(), sino que se centrará el código completamente al modelo fuente/receptor del Modelo de Delegación de Eventos. Dos objetos Label y un objeto Button son instanciados y añadidos a un objeto Frame. Cuando se pica sobre el botón, se genera un evento de tipo ActionEvent que es atrapado por un objeto ActionListener registrado sobre el objeto Button. El código del método actionPerformed() del objeto ActionListener cambia el color de fondo del objeto Label Muestra entre azul y amarillo, y viceversa.

```
import java.awt.*;
import java.awt.event.*;
public class java1115 extends Frame {
   public static void main( String args[] ) {
       java1115 displayWindow = new java1115();
       }
   public java1115() {
       setTitle( "Tutorial de Java, Eventos" );
       setLayout( new FlowLayout() );
       Button miBoton = new Button( "Pulsar" );
       Label miColorLabel = new Label( "  Muestra  " );
       Label miClickLabel = new Label( "Pulsar" );
       // Se añaden los componentes al Frame
       add( miBoton );
       add( miColorLabel );
       add( miClickLabel );
       // Se fija el tamaño del Frame y se presenta
       setSize( 250,100 );
       setVisible( true );
       // Se registran los objetos receptores de eventos
       miClickLabel.addMouseListener( new MiMouseListener( miBoton ) );
       miBoton.addActionListener(new MiActionListener( miColorLabel ) );
       // Concluye la aplicacion cuando se cierra la ventana
       this.addWindowListener( new Conclusion() );
       }
   }
// Esta clase MouseListener se utiliza para monitorizar los clicks del
// raton sobre un objeto de tipo Label. Cada vez que el usuario pulsa
// sobre la etiqueta, el codigo del objeto de esta clase crea un objeto
// ActionEvent y lo coloca en la cola de eventos del sistema. El origen
// del evento se especifica para que sea un objeto Button, que se pasa
// cuando se instancia un objeto de esta clase.
// De este modo, el objeto Label simula ser un Boton y genera eventos
// de tipo ActionEvent, que son interpretados por el sistema runtime
// como si estuviesen originados en un verdadero boton. El tipo de
// ActionEvents generados son eventos ACTION_PERFORMED, que son
// enviados automaticamente por el metodo actionPerformed() de un
// objeto ActionListener registrado sobre el boton
class MiMouseListener extends MouseAdapter {
   Button miBoton;
   MiMouseListener( Button boton ) {
       miBoton = boton;
       }
   // Sobreescribe el metodo mouseClicked()
```

```
    public void mouseClicked( MouseEvent evt ) {
        Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
            new ActionEvent( miBoton,
            ActionEvent.ACTION_PERFORMED,"evento" ) );
        }
    }
// Esta clase ActionListener se utiliza para instanciar un objeto
// Listener para el objeto Button. Cadavez que el boton sea pulsado,
// se envia un evento de tipo ActionEvent con el Boton como origen.
// El codigo de un objeto de esta clase va a cambiar el color de
// fondo de un objeto Label, intercambiandolo entre amarillo y azul
class MiActionListener implements ActionListener {
    int cambio = 0;
    Label miLabel;
    MiActionListener (Label label ) {
        miLabel = label;
        }

    public void actionPerformed( ActionEvent evt ) {
        if( cambio == 0 ) {
            cambio = 1;
            miLabel.setBackground( Color.yellow );
        } else {
            cambio = 0;
            miLabel.setBackground( Color.blue );
            }
        }
    }
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Termina la ejecucion del programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

Hasta aquí todo es normal. Sin embargo, se registra un objeto MouseListener sobre el objeto Label, para que cuando se pique sobre la etiqueta, el código del método mouseClicked() del objeto MouseListener genere un evento ActionEvent y lo coloque en la cola de eventos del Sistema. Lo que hace es simular que ha sido pulsado el botón, colocando la identificación del objeto Button en el parámetro source del objeto ActionEvent. El Sistema enviará este objeto ActionEvent al objeto ActionListener registrado sobre el objeto Button. El resultado final es que pulsando el ratón con el cursor colocado sobre la etiqueta, se invoca al método actionPerformed() registrado sobre el objeto Button, produciéndose exactamente el mismo resultado que cuando se pulsa el botón.

Durante la ejecución del programa se puede observar un efecto colateral curioso, y es que la etiqueta responde más rápido al ratón que el propio botón. Esto probablemente sea debido a los repintados que tiene que hacer el botón para simular la animación de la pulsación; quizá si se utilizasen simulaciones de los eventos pulsar y soltar sobre la etiqueta y se fuesen cambiando los colores de fondo, por ejemplo, se produjese una degradación similar.

El primer fragmento de código sobre el que hay que reparar en el ejemplo, son las dos sentencias en el constructor que registran objetos Listener sobre el botón y la etiqueta. Una cosa importante es que la referencia al objeto Button es pasada al constructor del objeto MouseListener del objeto Label, siendo éste el enlace que permite crear el evento ActionEvent desde la etiqueta y atribuírselo al botón.

También es interesante observar que el objeto Listener de la etiqueta no sabe nada sobre la otra etiqueta, Muestra, cuyo color se cambia durante la ejecución del programa. Esto es porque el objeto Label que va a simular al botón, no es el que cambia directamente el color del otro objeto Label; sino que va a ser el botón el encargado de realizar esa tarea.

```
miClickLabel.addMouseListener( new MiMouseListener( miBoton ) );
miBoton.addActionListener( new MiActionListener( miColorLabel ) );
```

La siguiente sentencia resultará conocida. Es la sentencia que va en el método mouseClicked() del objeto MouseListener que crea y envía el objeto ActionEvent y lo atribuye al objeto Button. Esto es posible, ya que tiene acceso a la referencia del objeto Button. Esta referencia es pasada como parámetro cuando el objeto MouseListener es construido, aunque puede hacerse de otras formas, por ejemplo, se podría consultar periódicamente la cola de eventos del Sistema en espera de cazar un evento de tipo ActionEvent y obtener la referencia del botón. Esta es una de las razones por las que se aplican normas estrictas de seguridad en el acceso de los applets a la cola de eventos del Sistema, porque podrían consultarla y obtener todo tipo de información si no se hiciese así.

Como el objeto ActionEvent es atribuido al objeto Button, será enviado al objeto ActionListener registrado sobre ese Button, donde será procesado como si su origen estuviese realmente en el botón.

```
Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
    new ActionEvent( miBoton,
    ActionEvent.ACTION_PERFORMED,"evento" ) );
```

Ya lo único que resta es llamar la atención sobre la sintaxis del objeto ActionEvent, en donde se utiliza la referencia al objeto Button como primer parámetro, source, y hay que recordar que se debe indicar un valor correcto para el parámetro id, sino el evento no será tratado por el Sistema.

Asistente creado por el usuario

A continuación se verá un ejemplo más, pero ya como indicación solamente, para la creación de eventos, para que el lector pueda investigar en un ejemplo útil sobre el que se proporciona suficiente información para que concluya el ejemplo y pueda comprobar que la lectura de las secciones anteriores ha sido del todo fructífera, y el concepto de funcionamiento de los eventos que se generan por programa está completamente comprendido.

Para ello lo que se va a intentar es implementar un interfaz de un Asistente, semejante al que se está popularizando en los entornos Windows. El Asistente estará constituido por un panel con un CardLayout como gestor de posicionamiento de los componentes y cuatro botones etiquetados como Siguiente, Anterior, Cancelar y Finalizar, que realizan las tareas que indican sus respectivos nombres. Para que este Asistente sea flexible, se proporciona un control total sobre las acciones de los botones, por ejemplo, cuando se pulsa el botón Siguiente, debería ser posible comprobar si los datos que son imprescindibles del panel que se está visualizando actualmente se han rellenado antes de pasar de verdad al siguiente.

En la creación del nuevo tipo de evento, se involucran cinco tareas principales, que a continuación se van a ver en detalle.

Crear un Receptor de Eventos

Una forma, entre otras, de informar a los objetos de que se ha producido una cierta acción, es crear un nuevo tipo de eventos que pueda ser enviado a los receptores que se encuentren registrados para recibirlo. En el caso del Asistente, el receptor deberá proporcionar soporte a cuatro tipos de eventos, uno para cada botón.

Se crea un interfaz para el receptor, definiendo un método para cada botón, de la forma siguiente:

```
import java.util.EventListener;

    public interface AsistenteListener exetends EventListener {
    public abstract void sigSeleccionado( AsistenteEvent evt );
    public abstract void antSeleccionado( AsistenteEvent evt );
    public abstract void cancelaSeleccionado( AsistenteEvent evt );
    public abstract void finalizaSeleccionado( AsistenteEvent evt );
    }
```

Cada método tiene un argumento de tipo AsistenteEvent, que se definirá posteriormente. El interfaz extiende a EventListener, utilizado para identificar este interfaz como un receptor de eventos del AWT.

Crear un Adaptador del Receptor de Eventos

Este es un paso opcional. En el AWT, un adaptador del receptor es una clase que proporciona una implementación por defecto para todos los métodos de un tipo determinado de receptor. Todos los adaptadores en el paquete java.awt.event proporcionan métodos vacíos que no hacen nada, así que un adaptador para el AsistenteListener sería:

```
public class AsistenteAdapter implements AsistenteListener {
    public abstract void sigSeleccionado( AsistenteEvent evt ) {}
    public abstract void antSeleccionado( AsistenteEvent evt ) {}
    public abstract void cancelaSeleccionado( AsistenteEvent evt ) {}
    public abstract void finalizaSeleccionado( AsistenteEvent evt ) {}
    }
```

Cuando se escriba una clase que sea un receptor para un Asistente, es posible extender el AsistenteAdapter y proporcionar la implementación solamente para aquellos métodos que intervengan. Es simplemente una clase de conveniencia.

Crear la Clase del Evento

Aquí es donde se crea la clase del evento, en este caso: AsistenteEvent.

```
import java.awt.AWTEvent;
public class AsistenteEvent extends AWTEvent {
    public static final int ASISTENTE_FIRST = AWTEvent.RESERVED_ID_MAX+1;
    public static final int SIG_SELECCIONADO = ASISTENTE_FIRST;
    public static final int ANT_SELECCIONADO = ASISTENTE_FIRST+1;
    public static final int CANCELA_SELECCIONADO = ASISTENTE_FIRST+2;
    public static final int FINALIZA_SELECCIONADO = ASISTENTE_FIRST+3;
    public static final int ASISTENTE_LAST = ASISTENTE_FIRST+3;
    public AsistenteEvent( Asistente source,int id ) {
        super( source,id );
        }
    }
```

Las dos constantes ASISTENTE_FIRST y ASISTENTE_LAST marcan el rango de máscaras que se utilizan en esta clase Event. Los identificadores de eventos usan la

constante RESERVED_ID_MAX de la clase AWTEvent para determinar el rango de identificadores que no presentan conflicto con los valores ya definidos por el AWT. Cuando se añadan más componentes RESERVED_ID_MAX puede estar incrementado.

Las cuatro constantes representan los identificadores de los cuatro tipos de eventos que definen la funcionalidad del Asistente. El identificador del evento y su origen son los dos argumentos que se pasan al constructor del evento del Asistente. Cada fuente de eventos debe ser de tipo Asistente, que es el tipo de componente para el que se define el evento. La razón es que solamente un panel Asistente puede ser origen de eventos de tipo Asistente. Observar que la clase AsistenteEvent extiende a AWTEvent.

Modificar el Componente

El siguiente paso es equipar al componente con métodos que permitan registrar y eliminar receptores para el nuevo evento.

Para enviar un evento a un receptor, normalmente se hace una llamada al método adecuado del receptor (dependiendo de la máscara del evento). Se puede registrar un receptor de eventos de acción desde el botón Siguiente y transmitirlo a los objetos AsistenteListener registrados. El método actionPerformed() del receptor para el botón Siguiente (u otras acciones) puede ser implementado como se muestra a continuación.

```
public void actionPerformed( ActionEvent evt ) {
    // No se hace nada si no hay receptores registrados
    if( asistenteListener == null )
        return;
    AsistenteEvent asistente;
    Asistente origen = this;
    // Control del botón "Siguiente"
    if( evt.getSource() == botonSiguiente ) {
        Asistente = new AsistenteEvent( origen,
            AsistenteEvent.SIG_SELECCIONADO );
        }
    // Los demás botones se controlan de forma similar
    }
```

Cuando se pulsa el botón Siguiente, se crea un nuevo evento AsistenteEvent con la máscara adecuada y el origen fijado en el botón Siguiente, que es el que se ha pulsado. En el ejemplo, la línea de código:

```
asistenteListener.sigSeleccionado( asistente );
```

se refiere al objeto asistenteListener, que es una variable miembro privada de Asistente y es de tipo AsistenteListener, tipo que se había definido en el primer paso de creación del evento.

En una primera impresión, el código parece que restringe el número de receptores a uno. La variable privada asistenteListener no es un array y solamente se puede hacer una llamada al método sigSeleccionado(). Para explicar el porqué de que no haya esta restricción, obliga a que haya que revisar la forma en que se añaden los receptores de eventos.

Cada nuevo componente que genera eventos predefinidos, o creados nuevos, necesita proporcionar dos métodos, uno para soportar la adición de receptores y otro para soportar la eliminación de receptores. En la clase Asistente, estos métodos son:

```
public synchronized void addAsistenteListener( AsistenteListener al ) {
```

```
    asistenteListener = AsistenteEventMulticaster.add(
        asistenteListener,al );
    }
public synchronized void removeAsistenteListener( AsistenteListener al )
{
    asistenteListener = AsistenteEventMulticaster.remove(
        asistenteListener,al );
    }
```

Ambos métodos hacen una llamada a miembros estáticos de la clase AsistenteEventMulticaster, lo que introduce el siguiente paso en que se permite el control de más de un receptor de eventos para el mismo tipo de evento que se genere.

Manejar Múltiples Receptores

Aunque es posible utilizar un Vector para manejar múltiples receptores, el JDK ahora define una clase especial para mantener esta lista de receptores, la clase AWTEventMulticaster. Una sola instancia de esta clase mantiene referencias a los objetos receptores. Como esta clase es un receptor en sí misma (implementa todos los interfaces de los receptores), cada uno de estos receptores pueden ser también multicasters, creándose así una cadena de receptores de eventos o multicasters.

Si un receptor de eventos es también un multicaster, entonces representa un eslabón de la cadena; si solamente es un receptor, representa un elemento final de la cadena. Desafortunadamente, no es posible reusar el AWTEventMulticaster en el manejo de eventos Multicaster para los nuevos tipos. La mejor forma de hacerlo es extender la clase AWTEventMulticaster, aunque esta operación sea en cierta forma cuestionable.

La clase AWTEventMulticaster contiene cincuenta y tantos métodos. De ellos, unos cincuenta proporcionan soporte a la docena de tipos de eventos y sus correspondientes receptores que son parte del AWT. De los restantes métodos, hay dos que necesitan recodificarse de nuevo, addInternal() y remove(); y se dice recodificar porque en AWTEventMulticaster, addInternal() es un método estático y no puede ser sobrecargado y, por razones que no se entienden muy bien, remove() hace una llamada a addInternal(), así que también necesita ser recodificado, de ahí que se pueda cuestionar esta técnica por parte de algunos puristas de los objetos.

Hay dos métodos, save() y saveInternal() que proporcionan soporte para ficheros o canales de comunicación que podrán ser reusados en la nueva clase Multicaster.

Por simplificar y en aras de la sencillez se hace una subclase de AWTEventMulticaster, pero muy sencilla, ya que podrían codificarse los métodos remove(), save() y saveInternal() para tener una funcionalidad completa. El código siguiente es el que implementa el multicaster para manejar el evento AsistenteEvent.

```
import java.awt.AWTEventMulticaster;
import java.util.EventListener;
public class AsistenteEventMulticaster extends AWTEventMulticaster
    implements AsistenteListener {
   protected AsistenteEventMulticaster(EventListener a,EventListener b )
{
    super( a,b );
    }
    public static AsistenteListener add( AsistenteListener a,
        AsistenteListener b ) {
        return( (AsistenteListener)addInternal( a,b ) );
```

```
    }
public static AsistenteListener remove( AsistenteListener al,
    AsistenteListener antal ) {
    return( (AsistenteListener)removeInternal( al,antal ) );
    }
public void sigSeleccionado( AsistenteEvent evt ) {
    // Aunque nunca se producirá una excepción por "casting"
    // hay que ponerlo, porque un multicaster puede manejar
    // más de un receptor
    if( a != null )
        ( (AsistenteListener)a).sigSeleccionado( evt );
    if( b != null )
        ( (AsistenteListener)b).sigSeleccionado( evt );
    }
public void antSeleccionado( AsistenteEvent evt ) {
    if( a != null )
        ( (AsistenteListener)a).antSeleccionado( evt );
    if( b != null )
        ( (AsistenteListener)b).antSeleccionado( evt );
    }
public void cancelaSeleccionado( AsistenteEvent evt ) {
    if( a != null )
        ( (AsistenteListener)a).cancelaSeleccionado( evt );
    if( b != null )
        ( (AsistenteListener)b).cancelaSeleccionado( evt );
    }
public void finalizaSeleccionado( AsistenteEvent evt ) {
    if( a != null )
        ( (AsistenteListener)a).finalizaSeleccionado( evt );
    if( b != null )
        ( (AsistenteListener)b).finalizaSeleccionado( evt );
    }
protected static EventListener addInternal( EventListener a,
    EventListener b ) {
    if( a == null )
        return( a );
    if( b == null )
        return( b );
    return( new AsistenteEventMulticaster( a,b );
    }
protected EventListener remove( EventListener antal ) {
    if( antal == a )
        return( a );
    if( antal == b )
        return( b );
    EventListener a2 = removeInternal( a,antal );
    EventListener b2 = removeInternal( b,antal );
    if( a2 == a && b2 == b )
        return( this );
    return( addInternal( a2,b2 ) );
    }
}
```

Funcionamiento del Asistente

Antes de ver el funcionamiento, se hace necesaria una revisión de los métodos que se han utilizado. El constructor de la clase Multicaster es protected, así que para obtener un nuevo AsistenteEventMulticaster hay que llamar al método add(), al que se pasan los receptores como argumentos, representando las dos piezas de una cadena de receptores que han de ser enlazados.

Para iniciar una nueva cadena, se pasará null en el primer argumento y para añadir un nuevo receptor, se pasará uno ya existente como primer argumento y el que se va a añadir como segundo argumento.

Otro método es remove(), al que se pasa un receptor (o un receptor multicaster) como primer argumento y como segundo argumento, el receptor que se va a eliminar.

Se han añadido cuatro métodos públicos para soportar la propagación de eventos a través de la cadena. Para cada tipo de AsistenteEvent, es decir, siguiente, anterior, cancelar y finalizar, hay un método. Estos métodos deben ser implementados desde el AsistenteEventMulticaster extendiendo el AsistenteListener, que es quien requiere que los cuatro métodos estén presentes.

Ahora sí es el momento de ver cómo funciona todo junto en el Asistente. Se supone que se ha construido un objeto asistente y se han añadido tres receptores de eventos, creando una cadena de receptores.

Inicialmente, la variable privada asistenteListener de la clase Asistente es null. Cuando se llama al método add(), el primer argumento es nulo y el segundo no. El método add() hace una llamada al método addInternal(). Aunque uno de los argumentos es nulo, el retorno de addInternal() es un receptor no-nulo. Esto se propaga al método add() que devuelve el receptor no-nulo al método addAsistenteListener(). Aquí, la variable asistenteListener se fija al nuevo receptor que se ha añadido.

Esto es exactamente lo que se pretendía. Si no hay receptores y se añade uno nuevo, se asignará a la variable asistenteListener. En este momento, asistenteListener es una referencia a un objeto AsistenteListener que no es un multicaster (no es necesario utilizar un multicaster si solamente hay un receptor registrado).

Cuando se hace una segunda llamada al método addAsistenteListener(), los dos argumentos que se le pasan no son nulos, así que es necesario un multicaster, por lo tanto, addInternal() devuelve una instancia a AsistenteEventListener que será asignado a la variable asistenteListener, que ahora contiene ya una cadena de dos receptores. Al añadir un tercer receptor el procedimiento es el mismo.

Si ahora se pulsa el botón Siguiente sobre el panel del Asistente, es suficiente con invocar al método sigSeleccionado() del objeto AsistenteListener representado por la variable asistenteListener y enviar un evento de tipo AsistenteEvent a todos los receptores de la cadena.

Para eliminar un receptor hay que buscarlo en la cadena de receptores de forma recursiva.

Como se puede observar por todo lo descrito, tanto en este ejemplo no desarrollado del todo, como en el anterior, el desarrollo de eventos propios o sintéticos en el nuevo modelo de eventos del JDK no es algo que se haga con simplicidad, sino que requiere código adicional. La interacción entre eventos diferentes y las clases que los soportan también es complicada de seguir.

Sin embargo, lo interesante es que no es necesario crear un nuevo multicaster para cada tipo de evento que se cree. Como un multicaster puede extender varios interfaces de

receptores, es suficiente con añadir un receptor y los métodos específicos del evento a un multicaster ya existente para conseguir que maneje más tipos de eventos.

Una cuestión que se ha dejado de lado en esta segunda explicación, pero que resulta interesante en la creación de eventos nuevos es el manejo de la cola de eventos y la habilitación de eventos, ya comentada en otra sección.

Eventos en Swing

Ahora se va a introducir el modelo de eventos en Swing, que es el modelo de Delegación. Los Componentes Swing no soportan el modelo de eventos de propagación, sino solamente el modelo de Delegación incluido desde el JDK 1.1; por lo tanto, si se van a utilizar Componentes Swing, se debe programar exclusivamente en el nuevo modelo. Aunque hay Componentes de Swing que se corresponden (o reemplazan) a componentes de AWT, hay otros que no tienen una contrapartida en el AWT. Esta circunstancia se reflejar en los dos ejemplos que se verán; en uno se hace una sustitución de Componentes y en el otro ya no hay contrapartida en el AWT.

Desde el punto de vista del manejo de eventos, los Componentes de Swing funcionan del mismo modo que los Componentes del AWT, a diferencia de los nuevos eventos que incorpora Swing. Desde otros puntos de vista, los Componentes Swing pueden parecerse ya más o menos a su contrapartida del AWT. Además, hay una serie de Componentes muy útiles en Swing, como son los árboles, la barra de progreso, los tooltips, que no tienen ningún Componente que se les pueda equiparar en el AWT.

En esta sección, solamente se va a tratar los Componentes Swing desde el punto de vista del control de eventos, dejando para otras secciones el estudio de los Componentes en sí mismos y de las características que aportan a Java.

En el ejemplo Java1116 , que es equivalente al ejemplo Java1101, solamente se reemplaza cada instancia de Frame por una instancia de JFrame, además de incorporar la sentencia import que hace que las clases sean accesibles al compilador y al intérprete. Por lo demás, el control de eventos en este nuevo ejemplo es el mismo que se ejercía en el ejemplo del AWT (si el lector ha entrado directamente en esta sección, quizá fuese conveniente que volviese atrás y le echase un vistazo a la anterior).

En el ejemplo se puede comprobar la utilización de fuentes de eventos, receptores de eventos y adaptadores del Modelo de Delegación de Eventos para Componentes Swing. Sucintamente, la aplicación instancia un objeto que crea un interfaz de usuario consistente en un JFrame. Este objeto es una fuente de eventos que notificará a dos objetos diferentes, receptores de eventos, de eventos de tipo Window.

Uno de los objetos Listener, receptores de eventos, implementa el interfaz WindowListener y define todos los métodos que se declaran en ese interfaz. El otro objeto Listener, extiende la clase Adapter, adaptador, llamada WindowAdapter, que ya no tiene porqué sobreescribir todos los métodos del interfaz, sino solamente aquellos que le resultan interesantes.

La aplicación no termina y devuelve el control al sistema operativo, sino que esto debe forzarse. El código del ejemplo se reproduce a continuación:

```
import java.awt.*;
```

```
import java.awt.event.*;
import java.awt.swing.*;      // Este es el paquete de Swing

public class java1116 {
    public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
    }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario, para que permita instanciar a su vez dos objetos Listener
// y registrarlos para que reciban notificacion cuando se producen
// eventos en una Ventana
class IHM {
    // Constructor de la clase
    public IHM() {
        // Se crea un objeto JFrame
        JFrame ventana = new JFrame();
        // El metodo setSize() reemplaza al metodo resize() del JDK 1.0
        ventana.setSize( 300,200 );
        ventana.setTitle( "Tutorial de Java, Eventos" );
        // El metodo setVisible() reemplaza al metodo show() del JDK 1.0
        ventana.setVisible( true );
        // Se instancian dos objetos receptores que procesaran los
        // eventos de la ventana
        Proceso1 ventanaProceso1 = new Proceso1( ventana );
        Proceso2 ventanaProceso2 = new Proceso2();
        // Se registran los dos objetos receptores para que sean
        // notificados de los evetnos que genere la ventana, que es el
        // objeto origen de los eventos
        ventana.addWindowListener( ventanaProceso1 );
        ventana.addWindowListener( ventanaProceso2 );
        }
    }

// Las dos clases siguientes se pueden utilizar para instanciar los
// objetos receptor. Esta clase implementa el interfaz WindowListener,
// lo cual requiere que todos los metodos que estan declarados en el
// interfaz sean definidos en la clase.
// La clase define todos esos metodos y presenta un mensaje
// descriptivo cada vez que se invoca a uno de ellos.
class Proceso1 implements WindowListener {
    // Variable utilizada para guardar una referencia al objeto JFrame
    JFrame ventanaRef;
    // Constructor que guarda la referencia al objeto JFrame
    Proceso1( JFrame vent ){
        this.ventanaRef = vent;
        }
    public void windowClosed( WindowEvent evt ) {
        System.out.println( "Metodo windowClosed de Proceso1" );
        }
    public void windowIconified( WindowEvent evt ) {
        System.out.println( "Metodo windowIconified de Proceso1" );
        }
    public void windowOpened( WindowEvent evt ) {
        System.out.println( "Metodo windowOpened de Proceso1" );
        }
    public void windowClosing( WindowEvent evt ) {
        System.out.println( "Metodo windowClosing de Proceso1" );
        // Se oculta la ventana
        ventanaRef.setVisible( false );
        }
    public void windowDeiconified( WindowEvent evt ) {
        System.out.println( "Metodo windowDeiconified Proceso1" );
        }
```

```
    public void windowActivated( WindowEvent evt ) {
        System.out.println( "Metodo windowActivated de Proceso1" );
        }
    public void windowDeactivated( WindowEvent evt ) {
        System.out.println( "Metodo windowDeactivated de Proceso1" );
        }
    }

// Esta clase y la anterior se pueden utilizar para instanciar
// objetos Listener. En esta clase, se extiende la clase Adapter
// obvienado el requerimiento de tener que definir todos los
// metodos del receptor de eventos WindowListener. El objeto
// Adapter, WindowAdapter extiende a WindowListener y define
// todos los metodos con codigo vacio, que pueden ser sobreescritos
// siempre que se desee. En este clase concreta, solamente se
// sobreescriben dos de los metodos declarados en el interfaz, y
// presenta un mensaje cada vez que se invoca a uno de ellos
class Proceso2 extends WindowAdapter {
    public void windowIconified( WindowEvent evt ) {
        System.out.println( "--- Metodo windowIconified de Proceso2" );
        }
    public void windowDeiconified( WindowEvent evt ) {
        System.out.println( "---Metodo windowDeiconified de Proceso2" );
        }
    }
```

Lo único destacable del código es el fragmento del comienzo del constructor de la clase IHM en que se utiliza la clase JFrame para instanciar al contenedor principal del interfaz gráfico.

```
class IHM {
    // Constructor de la clase
    public IHM() {
        // Se crea un objeto JFrame
        JFrame ventana = new JFrame();
        // El metodo setSize() reemplaza al metodo resize() del JDK 1.0
        ventana.setSize( 300,200 );
        ventana.setTitle( "Tutorial de Java, Eventos" );
```

Si se compila y ejecuta el programa, el lector podrá comprobar que no hay diferencias aparentes con el ejemplo , presentado en la sección anterior.

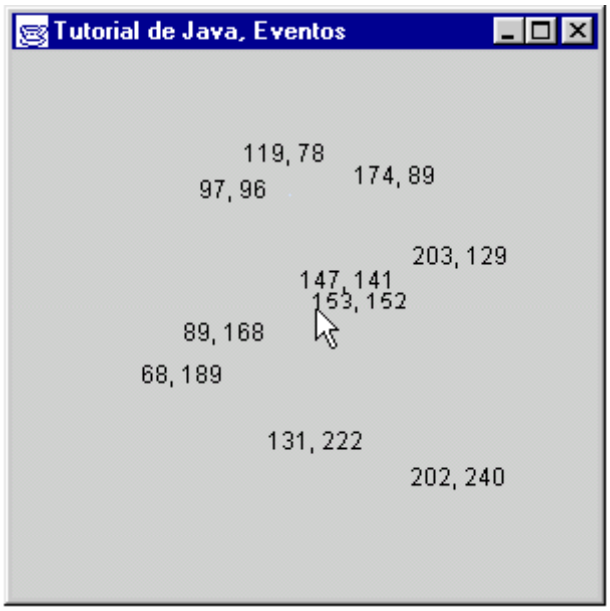
Pero tampoco las cosas son tan sencillas como parecen, porque hay ocasiones en que la conversión de un programa de AWT a Swing no es tan simple como una sustitución e importar la librería. Esto es lo que se muestra en el siguiente ejemplo,Java1117 , que es la contrapartida en Swing del ejemplo Java1102, ya visto al tratar del AWT. Se ha hecho lo mismo que en el caso anterior, sustituir Frame por JFrame e incorporar la sentencia que importa la librería de Swing.

La intención del programa era presentar las coordenadas del cursor en la posición en que se picase, dentro del área de influencia del Frame, en este caso del JFrame.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;      // Este es el paquete de Swing

public class java1117 {
    public static void main( String args[] ) {
        // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
    }
// Se crea una subclase de JFrame para poder sobreescribir el metodo
```

```
// paint(), y presentar en pantalla las coordenadas donde se haya
// producido el click del raton
class MiFrame extends JFrame {
    int ratonX;
    int ratonY;
    public void paint( Graphics g ) {
        g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
        }
    }
// Esta clase se utiliza para instaciar un objeto de tipo interfaz de
// usuario
class IHM {
    public IHM() {
        MiFrame ventana = new MiFrame();
        ventana.setSize( 300,300 );
        ventana.setTitle( "Tutorial de Java, Eventos" );
        ventana.setVisible( true );
        // Se instancia y registra un objeto receptor de eventos
        // para terminar la ejecucion del programa cuando el
        // usuario decida cerrar la ventana
        Proceso1 procesoVentana1 = new Proceso1();
        ventana.addWindowListener( procesoVentana1 );
        // Se instancia y registra un objeto receptor de eventos
        // que sera el encargado de procesas los eventos del raton
        // para determinar y presentar las coordenadas en las que
        // se encuentra el cursor cuando el usuario pulsa el boton
        // del raton
        ProcesoRaton procesoRaton = new ProcesoRaton( ventana );
        ventana.addMouseListener( procesoRaton );
        }
    }

// Esta clase Receptora monitoriza las pulsaciones de los botones
// del raton y presenta las coordenadas en las que se ha producido
// el click
// Se trata de una clase Adpater, luego solo se redefinen los metodos
// que resulten utiles para el objetivo de la aplicacion
class ProcesoRaton extends MouseAdapter {
  MiFrame ventanaRef; // Referencia a la ventana
    // Constructor
    ProcesoRaton( MiFrame ventana ) {
        // Guardamos una referencia a la ventana
        ventanaRef = ventana;
        }
    // Se sobreescribe el metodo mousePressed para determinar y
    // presentar en pantalla las coordenadas del cursor cuando
    // se pulsa el raton
    public void mousePressed( MouseEvent evt ) {
        // Recoge las coordenadas X e Y de la posicion del cursor
        // y las almacena en el objeto JFrame
        ventanaRef.ratonX = evt.getX();
        ventanaRef.ratonY = evt.getY();
        // Finalmente, presenta los valores de las coordenadas
        ventanaRef.repaint();
        }
    }
// Este repector de eventos de la ventana se utiliza para concluir
// la ejecucion del programa cuando el usuario pulsa sobre el boton
// de cierre del JFrame
class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

Si se compila y ejecuta el programa, inicialmente todo parece ir bien. Sin embargo, cuando se pican varias veces, se observa que las indicaciones de las coordenadas anteriores no desaparecen de la pantalla. Si se hace que el foco pase a otra de las aplicaciones que estén corriendo, entonces sí desaparecen esos textos antiguos de la ventana. Quizá sea un ejercicio interesante para el lector el descubrir por sí mismo el porqué de este funcionamiento, aparentemente anómalo.



La única parte destacable del código del programa, por indicar algo, es la que extiende la clase JFrame para dar origen a la clase MiFrame, en aras de hacer posible la sobreescritura del método paint() de la clase JFrame.

```
class MiFrame extends JFrame {
    int ratonX;
    int ratonY;

    public void paint( Graphics g ) {
        g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
        }
    }
```

El lector recordará que en el caso del ejemplo Java1102, la clase Frame se extendía de una forma similar para dar origen a otra clase donde poder sobrescribir el método paint().

Nuevos Eventos en Swing

Aunque en Swing la forma de manejar los eventos es similar a la del AWT del Modelos de Delegación, las clases Swing proporcionan una serie de nuevos tipos de eventos, algunos de los cuales se van a presentar en esta sección.

Una de las formas más fáciles de identificar estos nuevos tipos de eventos que incorpora Swing, es consultar los interfaces definidos en Swing o, también, echar una ojeada a la definición de las clases de los eventos en Swing. Ahora se presentan dos tablas,

la de la izquierda muestra una lista de los interfaces receptor definidos en el paquete Swing y la de la derecha muestra la lista de clases evento definidas en ese mismo paquete.

AncestorListener	AncestorEvent
CaretListener	CaretEvent
CellEditorListener	ChangeEvent
ChangeListener	EventListenerList
DocumentEvent	HyperlinkEvent
DocumentListener	InternalFrameAdapter
HyperlinkListener	ListDataEvent
InternalFrameListener	ListSelectionEvent
ListDataListener	MenuEvent
ListSelectionListener	PopMenuEvent
MenuListener	TableColumnModelEvent
PopMenuListener	TableModelEvent
TableColumnModelListener	TreeExpansionEvent
TableModelListener	TreeModelEvent
TreeExpansionListener	TreeSelectionEvent
TreeModelListener	UndoableEditEvent
TreeSelectionListener	
UndoableEditListener	

Se puede observar que no hay una correspondencia obvia entre los interfaces receptores y las clases de evento en todos los casos. En los dos ejemplos que siguen, se verán la clase AncestorEvent y el interfaz AncestorListener.

El primero de estos programas,Java1118 , muestra el uso de getContentPane() para añadir un objeto Swing de tipo JButton a un JFrame, de la misma forma que se ha visto en ejemplos de otras secciones, pero en este caso se profundiza en lo que a los eventos se refiere, ilustrando el uso del interfaz AncestorListener sobre un objeto Swing JButton.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;
import java.awt.swing.event.*;
public class java1118 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }
// Clase para instanciar un objeto de tipo interfaz gráfico
class IHM {
    public IHM(){
        // Crea un JFrame y le pone título, tamaño, etc.
        JFrame ventana = new JFrame();
        ventana.setSize( 300,300 );
        ventana.setTitle( "Tutorial de Java, Swing" );
       // Se añade el receptor de eventos de la ventana para concluir la
       // ejecución del programa
        ventana.addWindowListener( new Conclusion() );
        // Se crea un objeto JButton
        JButton boton = new JButton( "Boton" );
```

```
        // Se registra un objeto AncestorListener sobre cada JButton
boton.addAncestorListener( new MiAncestorListener() );
        // Se añade el botón al objeto JFrame
        ventana.getContentPane().add( boton );
        System.out.println( "Se hace visible el JFrame" );
        ventana.setVisible( true );
        }
    // Esta clase se utiliza para concluir el programa cuando el
    // usuario decide cerrar la ventana
    class Conclusion extends WindowAdapter {
        public void windowClosing( WindowEvent evt ) {
            System.exit( 0 );
            }
        }
    // Definicion de la clase AncestorListener
    class MiAncestorListener implements AncestorListener{
        // Se definen los tres métodos declarados en el interfaz
        // AncestorListener
        public void ancestorAdded( AncestorEvent evt ) {
            System.out.println( "Llamada al metodo ancestorAdded" );
            System.out.println( "Origen Evento: " + evt.getSource() );
            System.out.println( "Ancestor: " + evt.getAncestor() );
            System.out.println( "Padre: " + evt.getAncestorParent() );
            System.out.println( "Componente: " + evt.getComponent() );
            System.out.println( "ID: " + evt.getID() );
            }
        public void ancestorRemoved( AncestorEvent evt ) {
            System.out.println( "Metodo ancestorRemoved" );
            }
        public void ancestorMoved( AncestorEvent evt ) {
            System.out.println( "Metodo ancestorMoved" );
            }
        }
    }
```

A la hora de repasar el código de la aplicación, en este caso hay que empezar desde las sentencias import, porque ya en ellas está el código que permite que tanto compilador como intérprete Java, puedan acceder a las clases Swing.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;
import java.awt.swing.event.*;
```

El método main(), en este caso es tan simple, que no merece comentario alguno. No obstante, ahí se construye un objeto de la clase IHM que será el que se presente en pantalla. El constructor de esta clase, reproducido en las siguiente líneas, es muy sencillo.

```
class IHM {
    public IHM(){
        // Crea un JFrame y le pone título, tamaño, etc.
        JFrame ventana = new JFrame();
        ventana.setSize( 300,300 );
        ventana.setTitle( "Tutorial de Java, Swing" );
       // Se añade el receptor de eventos de la ventana para concluir la
       // ejecución del programa
        ventana.addWindowListener( new Conclusion() );
        // Se crea un objeto JButton
        JButton boton = new JButton( "Boton" );
        // Se registra un objeto AncestorListener sobre cada JButton
boton.addAncestorListener( new MiAncestorListener() );
        // Se añade el botón al objeto JFrame
        ventana.getContentPane().add( boton );
        System.out.println( "Se hace visible el JFrame" );
        ventana.setVisible( true );
```

```
}
```

Instancia un objeto Swing de tipo JFrame, fijando su tamaño, proporcionándole un título, etc. Además, también se añade un objeto WindowListener para recoger los eventos de la ventana y terminar el programa cuando el usuario cierre el objeto JFrame.

Luego se instancia un objeto Swing de tipo JButton y se registra un objeto de tipo AncestorListener sobre ese botón. A continuación, se añade el objeto JButton al objeto JFrame ventana, invocando al método getContentPane() y luego al método add() sobre el contenido anterior. Por fin, se presenta un mensaje y se hace visible el objeto JFrame, concluyendo el construtor.

El método getContentPane() no existe en el AWT, donde la única forma de añadir Componentes es manipular directamente el área cliente del objeto Frame. En Swing, sin embargo, algunos paneles son colocados automáticamente sobre el ásrea cliente de un objeto JFrame, con lo cual se pueden añadir Componentes, o manipular, estos paneles en vez de manipular el área cliente del objeto JFrame directamente.

La definición de la clase JFrame es la siguiente:

```
public class JFrame extends Frame implements WindowConstants,
Accesible,RootPaneContainer
```

Es decir, es una extensión de la clase Frame del AWT que le añade soporte para recibir entradas y pintar sobre objetos Frame hijos, soporte para hijos especiales controlados por un LayeredPanel y soporte para las barras de menú de Swing. Esta clase JFrame es ligeramente incompatible con la clase Frame del AWT. JFrame contiene un objeto JRootPane como único hijo. El contentPane debería ser el padre de cualquier hijo del JFrame. Esto difiere con respecto al Frame del AWT; por ejemplo, para añadir un hijo a un Frame, se escribiría:

```
frame.add( hijo );
```

mientras que en el caso del JFrame, es necesario añadir el hijo al contentPane, de la siguiente forma:

```
frame.getContentPane().add( hijo );
```

Esto mismo es válido a la hoja de fijar el controlador de posicionamiento de los Componentes, eliminar Componentes, listar los hijos, etc. Todos estos métodos normalmente, deberían ser enviados al contentPane en vez de directamente al JFrame. El contentPane siempre será distinto de nulo y, por defecto, tendrá un BorderLayout como controlador de posicionamiento. Si se intenta hacer que contentPane sea nulo, el sistema generará una excepción. En este ejemplo es suficiente con insertar una llamada al método entre la referencia al objeto JFrame y las llamada a add(), setLayout(), etc. En programas más complejos, las ramificaciones probablemente creasen mayores problemas.

El interfaz de usuario del programa tiene dos clases anidadas. Una de ellas es una clase WindowListener que se utiliza para concluir la ejecución del programa cuando el usuario cierra el JFrame. Es muy simple, y ya se ha visto en otras secciones, aquí se incluye como anidada para mostrar la versatilidad de Java y ver que se puede hacer lo mismo de varias formas diferentes.

La segunda clase anidada se utiliza para instancia un objeto de tipo AncestorListener que será registrado sobre el objeto JButton. Esto ya es un poco más

interesante. El interfaz AncestorListener declara tres métodos, así que la clase debe implementar estos tres métodos, que son:

ancestorAdded(AncestorEvent), llamado cuando el origen o uno de sus antecesores se hace visible, bien porque se llame al método setVisible() pasándole el parámetro true, o porque se haya añadido el Componente a la jerarquía.

ancestorMoved(AncestorEvent), llamado cuando el origen o uno de sus antecesores es movido.

ancestorRemoved(AncestorEvent), llamado cuando el origen o uno de sus antecesores se hace invisible, bien porque se llame al método setVisible() pasándole el parámetro false, o porque se haya eliminado el Componente de la jerarquía.

Como se puede observar, cuando alguno de estos método es llamado, se le pasa un objeto de tipo AncestorEvent como parámetro. Cuando se llama al primero de ellos, invoca a su vez a métodos del AncestorEvent que se le pasa para presentar en pantalla información sobre el antecesor.

```
class MiAncestorListener implements AncestorListener{
    // Se definen los tres métodos declarados en el interfaz
    // AncestorListener
    public void ancestorAdded( AncestorEvent evt ) {
        System.out.println( "Llamada al metodo ancestorAdded" );
        System.out.println( "Origen Evento: " + evt.getSource() );
        System.out.println( "Ancestor: " + evt.getAncestor() );
        System.out.println( "Padre: " + evt.getAncestorParent() );
        System.out.println( "Componente: " + evt.getComponent() );
        System.out.println( "ID: " + evt.getID() );
        }
```

La verdad es que si se compila y ejecuta el programa, se obtendría algo como la salida de pantalla capturada a continuación:

```
% java java1118
Se hace visible el JFrame
Llamada al metodo ancestorAdded
Origen Evento: java.awt.swing.JButton[,0,0,0x0,invalid,
    layout=java.awt.swing.OverlayLayout]
Ancestor: java.awt.swing.JButton[,0,0,0x0,invalid,
    layout=java.awt.swing.OverlayLayout]
Padre: java.awt.swing.JPanel[null.contentPane,0,0,0x0,invalid,
    layout=java.awt.swing.JRootPane$1]
Componente: java.awt.swing.JButton[,0,0,0x0,invalid,
    layout=java.awt.swing.OverlayLayout]
ID: 1
Metodo ancestorMoved
```

Y esto no parece coincidir demasiado con las descripciones que JavaSoft proporciona en su documentación sobre los métodos getAncestor() y getAncestorParent(). La salida parece referirse al objeto JButton como antecesor y al objeto JRootPane como padre del antecesor. Pero lo cierto es que el objeto JButton es un hijo, no un antecesor; aunque es de imaginar que esto dependerá de la interpretación que se le quiera dar. El caso es que no está demasiado claro en la documentación.

El fragmento final del código es el que muestra la definición de los otros dos métodos del interfaz AncestorListener, y que se reproduce a continuación:

```
public void ancestorRemoved( AncestorEvent evt ) {
    System.out.println( "Metodo ancestorRemoved" );
    }
```

```
public void ancestorMoved( AncestorEvent evt ) {
    System.out.println( "Metodo ancestorMoved" );
    }
```

Si se ejecuta el programa, aparte de observar la salida referida anteriormente, se observa que cuando se mueve, iconiza o desiconiza la ventana, se generan llamadas al método ancestorMoved(). Cuando se hace visible el JFrame, se llaman a los dos métodos, ancestorAdded() y ancestorMoved().

A continuación se muestra otro ejemplo, Java1119, que ilustra el uso de un receptor de tipo AncestorListener sobre un JButton, y lo que es más importante, ilustra el hecho de que objetos como JButton pueden ser contenedores de otros objetos, incluyendo entre ellos a otros objeto JButton.

El programa apila tres objetos JButton apilados y colocados sobre un objeto JFrame, tal como muestra la figura anterior. Los objetos ActionListener se registran sobre cada uno de los botones para atrapar los eventos de tipo Action cuando se pulsa el botón y poder presentar la información correspondiente al origen del evento. Los objetos AncestorListener también son registrados sobre los objetos JButton.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;
import java.awt.swing.event.*;
public class java1119 {
    public static void main( String args[] ){
        IHM ihm = new IHM();
        }
    }
// Clase para instanciar un objeto de tipo interfaz gráfico
class IHM {
    public IHM() {
        // Crea un JFrame y le pone título, tamaño, etc.
        JFrame ventana = new JFrame();
        ventana.setSize( 300,100 );
        ventana.setTitle( "Tutorial de Java, Swing" );
        // Obsérvese la utilización de getContentPane() en la siguiente
        // sentencia
        ventana.getContentPane().setLayout( new FlowLayout() );
       // Se añade el receptor de eventos de la ventana para concluir la
       // ejecución del programa
        ventana.addWindowListener( new Conclusion() );
        // Se crean los tres objetos JButton
        JButton primerBoton = new JButton( "Primer Boton" );
        JButton segundoBoton = new JButton( "Segundo Boton" );
        JButton tercerBoton = new JButton( "Tercer Boton" );
        // Se apilan los botones uno sobre otro
        primerBoton.add( segundoBoton );
        segundoBoton.add( tercerBoton );
        // Se registra un objeto AncestorListener sobre cada JButton
primerBoton.addAncestorListener( new MiAncestorListener() );
```

```
        segundoBoton.addAncestorListener( new MiAncestorListener() );
        tercerBoton.addAncestorListener( new MiAncestorListener() );
        // Se registra un objeto ActionListener sobre cada JButton
        primerBoton.addActionListener( new MiActionListener() );
        segundoBoton.addActionListener( new MiActionListener() );
        tercerBoton.addActionListener( new MiActionListener() );

        // Se añade el primer botón, que contiene a los demás, al
        // objeto JFrame
        ventana.getContentPane().add( primerBoton );
        System.out.println( "Se hace visible el JFrame" );
        ventana.setVisible( true );
        }
    // Esta clase se utiliza para concluir el programa cuando el
    // usuario decide cerrar la ventana
    class Conclusion extends WindowAdapter{
        public void windowClosing( WindowEvent evt ) {
            System.exit( 0 );
            }
        }
    // Definicion de la clase AncestorListener
    class MiAncestorListener implements AncestorListener {
        // Se definen los tres métodos declarados en el interfaz
        // AncestorListener, incorporando el moldeo necesario para
        // que nadie se queje
        public void ancestorAdded( AncestorEvent evt ) {
            System.out.println( "Metodo ancestorAdded" );
            System.out.println( " Origen del evento: " +
                ( (JButton)evt.getSource() ).getActionCommand() );
            }
        public void ancestorRemoved( AncestorEvent evt ) {
            System.out.println( "Metodo ancestorRemoved" );
            System.out.println( " Origen del evento: " +
                ( (JButton)evt.getSource() ).getActionCommand() );
            }
        public void ancestorMoved( AncestorEvent evt ) {
            System.out.println( "Metodo ancestorMoved" );
            System.out.println( " Origen del evento: " +
                ( (JButton)evt.getSource() ).getActionCommand() );
            }
        }
    // Definicion de la clase ActionListener
    class MiActionListener implements ActionListener {
        public void actionPerformed( ActionEvent evt ) {
            System.out.println( "Metodo actionPerformed" );
            System.out.println( " Origen del evento: " +
                ( (JButton)evt.getSource() ).getActionCommand() );
            }
        }
    }
```

La mayor parte del código de este programa es semejante al de los anteriores. Cuando se instancian los tres botones, se apilan añadiendo el segundoBoton al primerBoton y añadiendo el tercerBoton al segundoBoton.

Se registra un AncestorListener sobre los tres botones y luego un objeto ActionListener también sobre ellos. La clase AncestorListener es muy similar a la del ejemplo anterior; no obstante, es de notar la necesidad de moldeo en esta versión del método. Esto se debe a la invocación del método getSource() que devuelve un objeto de tipo Object, que debe ser moldeado a un JButton para que pueda ser utilizado.

```
class MiAncestorListener implements AncestorListener {
    // Se definen los tres métodos declarados en el interfaz
    // AncestorListener, incorporando el moldeo necesario para
```

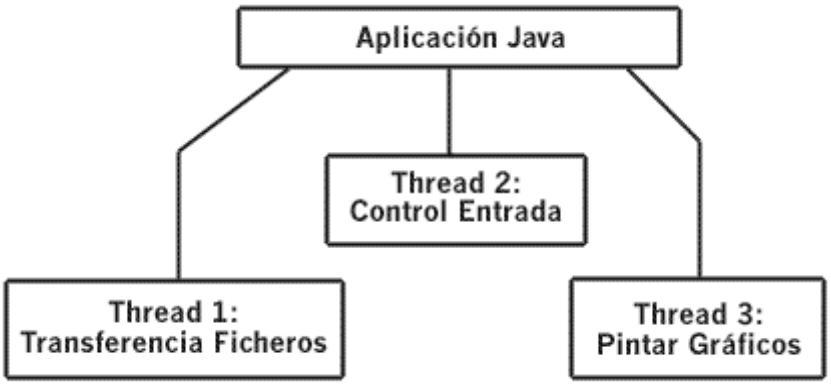
```
    // que nadie se queje
    public void ancestorAdded( AncestorEvent evt ) {
        System.out.println( "Metodo ancestorAdded" );
        System.out.println( " Origen del evento: " +
            ( (JButton)evt.getSource() ).getActionCommand() );
        }
```

Y ya, lo más interesante es ver cómo la clase ActionListener atrapa los eventos de tipo Action que se producen en los botones cuando se pulsan.

```
class MiActionListener implements ActionListener {
    public void actionPerformed( ActionEvent evt ) {
        System.out.println( "Metodo actionPerformed" );
        System.out.println( " Origen del evento: " +
            ( (JButton)evt.getSource() ).getActionCommand() );
        }
    }
```

Hilos y Multihilos

Considerando el entorno multithread (multihilo), cada thread (hilo, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. A veces se les llama procesos ligeros o contextos de ejecución. Típicamente, cada hilo controla un único aspecto dentro de un programa, como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todos los hilos comparten los mismos recursos, al contrario que los procesos, en donde cada uno tiene su propia copia de código y datos (separados unos de otros). Gráficamente, los hilos (threads) se parecen en su funcionamiento a lo que muestra la figura siguiente:



Hay que distinguir multihilo (multithread) de multiproceso. El multiproceso se refiere a dos programas que se ejecutan "aparentemente" a la vez, bajo el control del Sistema Operativo. Los programas no necesitan tener relación unos con otros, simplemente el hecho de que el usuario desee que se ejecuten a la vez.

Multihilo se refiere a que dos o más tareas se ejecutan "aparentemente" a la vez, dentro de un mismo programa.

Se usa "aparentemente" en ambos casos, porque normalmente las plataformas tienen una sola CPU, con lo cual, los procesos se ejecutan en realidad "concurrentemente", sino que comparten la CPU. En plataformas con varias CPU, sí es posible que los procesos se ejecuten realmente a la vez.

Tanto en el multiproceso como en el multihilo (multitarea), el Sistema Operativo se encarga de que se genere la ilusión de que todo se ejecuta a la vez. Sin embargo, la multitarea puede producir programas que realicen más trabajo en la misma cantidad de tiempo que el multiproceso, debido a que la CPU está compartida entre tareas de un mismo proceso. Además, como el multiproceso está implementado a nivel de sistema operativo, el programador no puede intervenir en el planteamiento de su ejecución; mientras que en el caso del multihilo, como el programa debe ser diseñado expresamente para que pueda soportar esta característica, es imprescindible que el autor tenga que planificar adecuadamente la ejecución de cada hilo, o tarea.

Actualmente hay diferencias en la especificación del intérprete de Java, porque el intérprete de Windows '95 conmuta los hilos de igual prioridad mediante un algoritmo circular (round-robin), mientras que el de Solaris 2.X deja que un hilo ocupe la CPU indefinidamente, lo que implica la inanición de los demás.

Programas de flujo único

Un programa de flujo único o mono-hilvanado (single-thread) utiliza un único flujo de control (thread) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchos de los applets y aplicaciones son de flujo único.

Por ejemplo, en la archiconocida aplicación estándar de saludo:

```
public class HolaMundo {
    static public void main( String args[] ) {
        System.out.println( "Hola Mundo!" );
        }
    }
```

Aquí, cuando se llama a main(), la aplicación imprime el mensaje y termina. Esto ocurre dentro de un único hilo de ejecución (thread).

Debido a que la mayor parte de los entornos operativos no solían ofrecer un soporte razonable para múltiples hilos de control, los lenguajes de programación tradicionales, tales como C++, no incorporaron mecanismos para describir de manera elegante situaciones de este tipo. La sincronización entre las múltiples partes de un programa se llevaba a cabo mediante un bucle de suceso único. Estos entornos son de tipo síncrono, gestionados por sucesos. Entornos tales como el de Macintosh de Apple, Windows de Microsoft y X11/Motif fueron diseñados en torno al modelo de bucle de suceso.

Programas de flujo múltiple

En la aplicación de saludo, no se ve el hilo de ejecución que corre el programa. Sin embargo, Java posibilita la creación y control de hilos de ejecución explícitamente. La utilización de hilos (threads) en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar hilos de ejecución, permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se puede con otros lenguajes de tercera generación. En un lenguaje orientado a Internet como es Java, esta herramienta es vital.

Si se ha utilizado un navegador con soporte Java, ya se habrá visto el uso de múltiples hilos en Java. Habrá observado que dos applets se pueden ejecutar al mismo tiempo, o que puede desplazar la página del navegador mientras el applet continúa ejecutándose. Esto no significa que el applet utilice múltiples hilos, sino que el navegador es multihilo, multihilvanado o multithreaded.

Los navegadores utilizan diferentes hilos ejecutándose en paralelo para realizar varias tareas, "aparentemente" concurrentemente. Por ejemplo, en muchas páginas web, se puede desplazar la página e ir leyendo el texto antes de que todas las imágenes estén presentes en la pantalla. En este caso, el navegador está trayéndose las imágenes en un hilo de ejecución y soportando el desplazamiento de la página en otro hilo diferente.

Las aplicaciones (y applets) multihilo utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo de ejecución para cada subtarea.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multihilo permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

Vamos a modificar el programa de saludo creando tres hilos de ejecución individuales, que imprimen cada uno de ellos su propio mensaje de saludo, :

```
// Definimos unos sencillos hilos. Se detendrán un rato
// antes de imprimir sus nombres y retardos

class TestTh extends Thread {
    private String nombre;
    private int retardo;

    // Constructor para almacenar nuestro nombre
    // y el retardo
    public TestTh( String s,int d ) {
        nombre = s;
        retardo = d;
        }

    // El metodo run() es similar al main(), pero para
    // threads. Cuando run() termina el thread muere
    public void run() {
          // Retasamos la ejecución el tiempo especificado
        try {
            sleep( retardo );
        } catch( InterruptedException e ) {
            ;
        }

        // Ahora imprimimos el nombre
        System.out.println( "Hola Mundo! "+nombre+" "+retardo );
        }
    }

public class MultiHola {
    public static void main( String args[] ) {
        TestTh t1,t2,t3;

        // Creamos los threads
        t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
        t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
        t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );

        // Arrancamos los threads
        t1.start();
        t2.start();
        t3.start();
        }
    }
```

La clase Thread

Es la clase que encapsula todo el control necesario sobre los hilos de ejecución (threads). Hay que distinguir claramente un objeto Thread de un hilo de ejecución o thread. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto

Thread como el panel de control de un hilo de ejecución (thread). La clase Thread es la única forma de controlar el comportamiento de los hilos y para ello se sirve de los métodos que se exponen en las secciones siguientes.

Métodos de Clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase Thread.

currentThread()

Este método devuelve el objeto thread que representa al hilo de ejecución que se está ejecutando actualmente.

yield()

Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran inanición.

sleep(long)

El método sleep() provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

Métodos de Instancia

Aquí no están recogidos todos los métodos de la clase Thread, sino solamente los más interesantes, porque los demás corresponden a áreas en donde el estándar de Java no está completo, y puede que se queden obsoletos en la próxima versión del JDK, por ello, si se desea completar la información que aquí se expone se ha de recurrir a la documentación del interfaz de programación de aplicación (API) del JDK.

start()

Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método run() de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método start() más de una vez sobre un hilo determinado.

run()

El método run() constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz Runnable. Es llamado por el método start() después de que el hilo

apropiado del sistema se haya inicializado. Siempre que el método run() devuelva el control, el hilo actual se detendrá.

stop()

Este método provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método stop() no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que stop() devuelva el control. Una forma más elegante de detener un hilo es utilizar alguna variable que ocasione que el método run() termine de manera ordenada. En realidad, nunca se debería recurrir al uso de este método.

suspend()

El método suspend() es distinto de stop(). suspend() toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a resume() sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

resume()

El método resume() se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de mayor prioridad en ejecución actualmente, pero resume() ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.

setPriority(int)

El método setPriority() asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la clase Thread, tales como MIN_PRIORITY, NORM_PRIORITY y MAX_PRIORITY, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a NORM_PRIORITY. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a MIN_PRIORITY. Con las tareas a las que se fije la máxima prioridad, en torno a MAX_PRIORITY, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a sleep() o yield(), se puede provocar que el intérprete Java quede totalmente fuera de control.

getPriority()

Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.

setName(String)

Este método permite identificar al hilo con un nombre mnemónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

getName()

Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante setName().

Creación de un Thread

Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando el interfaz Runnable, la otra es extender la clase Thread.

La implementación del interfaz Runnable es la forma habitual de crear hilos. Los interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar. El interfaz define el trabajo y la clase, o clases, que implementan el interfaz para realizar ese trabajo. Los diferentes grupos de clases que implementen el interfaz tendrán que seguir las mismas reglas de funcionamiento.

Hay una cuantas diferencias entre interfaz y clase, que ya son conocidas y aquí solamente se resumen. Primero, un interfaz solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes. Segundo, un interfaz no puede implementar cualquier método. Una clase que implemente un interfaz debe implementar todos los métodos definidos en ese interfaz. Un interfaz tiene la posibilidad de poder extenderse de otros interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces. Además, un interfaz no puede ser instanciado con el operador new; por ejemplo, la siguiente sentencia no está permitida:

```
Runnable a = new Runnable();    // No se permite
```

El primer método de crear un hilo de ejecución es simplemente extender la clase Thread:

```
class MiThread extends Thread {
    public void run() {
        . . .
        }
```

El ejemplo anterior crea una nueva clase MiThread que extiende la clase Thread y sobreescribe el método Thread.run() por su propia implementación. El método run() es donde se realizará todo el trabajo de la clase. Extendiendo la clase Thread, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Esta limitación de Java puede ser superada a través de la implementación de Runnable:

```
public class MiThread implements Runnable {
    Thread t;
    public void run() {
          // Ejecución del thread una vez creado
        }
    }
```

En este caso necesitamos crear una instancia de Thread antes de que el sistema pueda ejecutar el proceso como un hilo. Además, el método abstracto run() está definido en el interfaz Runnable y tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía está la oportunidad de extender la clase MiThread, si fuese necesario. La mayoría de las clases

creadas que necesiten ejecutarse como un hilo, implementarán el interfaz Runnable, ya que probablemente extenderán alguna de su funcionalidad a otras clases.

No pensar que el interfaz Runnable está haciendo alguna cosa cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual es una clase para dar idea sobre el diseño de la clase Thread. De hecho, si se observan los fuentes de Java, se puede comprobar que solamente contiene un método abstracto:

```
package java.lang;
public interface Runnable {
    public abstract void run() ;
    }
```

Y esto es todo lo que hay sobre el interfaz Runnable. Como se ve, un interfaz sólo proporciona un diseño para las clases que vayan a ser implementadas. En el caso de Runnable, fuerza a la definición del método run(), por lo tanto, la mayor parte del trabajo se hace en la clase Thread. Un vistazo un poco más profundo a la definición de la clase Thread da idea de lo que realmente está pasando:

```
public class Thread implements Runnable {
    ...
    public void run() {
        if( tarea != null )
            tarea.run() ;
            }
        }
    ...
    }
```

De este trocito de código se desprende que la clase Thread también implemente el interfaz Runnable. tarea.run() se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un hilo) no sea nula y ejecuta el método run() de esa clase. Cuando esto suceda, el método run() de la clase hará que corra como un hilo.

A continuación se presenta el ejemplo java1001 , que implementa el interfaz Runnable para crear un programa multihilo.

```
class java1001 {
    static public void main( String args[] ) {
        // Se instancian dos nuevos objetos Thread
        Thread hiloA = new Thread( new MiHilo(),"hiloA" );
        Thread hiloB = new Thread( new MiHilo(),"hiloB" );

          // Se arrancan los dos hilos, para que comiencen su
ejecución
        hiloA.start();
        hiloB.start();

          // Aquí se retrasa la ejecución un segundo y se captura
la
          // posible excepción que genera el método, aunque no se
hace
        // nada en el caso de que se produzca
        try {
            Thread.currentThread().sleep( 1000 );
        }catch( InterruptedException e ){}

          // Presenta información acerca del Thread o hilo principal
        // del programa
        System.out.println( Thread.currentThread() );
```

```
          // Se detiene la ejecución de los dos hilos
        hiloA.stop();
        hiloB.stop();
        }
    }

class NoHaceNada {
// Esta clase existe solamente para que sea heredada por la clase
// MiHilo, para evitar que esta clase sea capaz de heredar la clase
// Thread, y se pueda implementar el interfaz Runnable en su
// lugar
}

class MiHilo extends NoHaceNada implements Runnable {
    public void run() {
          // Presenta en pantalla información sobre este hilo en
particular
        System.out.println( Thread.currentThread() );
        }
    }
```

Como se puede observar, el programa define una clase MiHilo que extiende a la clase NoHaceNada e implementa el interfaz Runnable. Se redefine el método run() en la clase MiHilo para presentar información sobre el hilo.

La única razón de extender la clase NoHaceNada es proporcionar un ejemplo de situación en que haya que extender alguna otra clase, además de implementar el interfaz.

En el ejemplo java1002 muestra el mismo programa básicamente, pero en este caso extendiendo la clase Thread, en lugar de implementar el interfaz Runnable para crear el programa multihilo.

```
class java1002 {
    static public void main( String args[] ) {
        // Se instancian dos nuevos objetos Thread
        Thread hiloA = new Thread( new MiHilo(),"hiloA" );
        Thread hiloB = new Thread( new MiHilo(),"hiloB" );

          // Se arrancan los dos hilos, para que comiencen su ejecución
        hiloA.start();
        hiloB.start();

        // Aquí se retrasa la ejecución un segundo y se captura la
       // posible excepción que genera el método, aunque no se hace
       // nada en el caso de que se produzca
        try {
            Thread.currentThread().sleep( 1000 );
        }catch( InterruptedException e ){}

          // Presenta información acerca del Thread o hilo principal
        // del programa
        System.out.println( Thread.currentThread() );

          // Se detiene la ejecución de los dos hilos
        hiloA.stop();
        hiloB.stop();
        }
    }

class MiHilo extends Thread {
    public void run() {
          // Presenta en pantalla información sobre este hilo en particular
        System.out.println( Thread.currentThread() );
```

```
        }
    }
```

En ese caso, la nueva clase MiHilo extiende la clase Thread y no implementa el interfaz Runnable directamente (la clase Thread implementa el interfaz Runnable, por lo que indirectamente MiHilo también está implementando ese interfaz). El resto del programa es similar al anterior.

Y todavía se puede presentar un ejemplo más simple, utilizando un constructor de la clase Thread que no necesita parámetros, tal como se presenta en el ejemplo java1003. En los ejemplos anteriores, el constructor utilizado para Thread necesitaba dos parámetros, el primero un objeto de cualquier clase que implemente el interfaz Runnable y el segundo una cadena que indica el nombre del hilo (este nombre es independiente del nombre de la variable que referencia al objeto Thread).

```
class java1003 {
    static public void main( String args[] ) {
        // Se instancian dos nuevos objetos Thread
        Thread hiloA = new MiHilo();
        Thread hiloB = new MiHilo();

          // Se arrancan los dos hilos, para que comiencen su ejecución
        hiloA.start();
        hiloB.start();

          // Aquí se retrasa la ejecución un segundo y se captura la
       // posible excepción que genera el método, aunque no se hace
       // nada en el caso de que se produzca
        try {
            Thread.currentThread().sleep( 1000 );
        }catch( InterruptedException e ){}

          // Presenta información acerca del Thread o hilo principal
       // del programa
        System.out.println( Thread.currentThread() );

          // Se detiene la ejecución de los dos hilos
        hiloA.stop();
        hiloB.stop();
        }
    }

class MiHilo extends Thread {
    public void run() {
          // Presenta en pantalla información sobre este hilo en particular
        System.out.println( Thread.currentThread() );
        }
    }
```

Las sentencias en este ejemplo para instanciar objetos Thread, son mucho menos complejas, siendo el programa, en esencia, el mismo de los ejemplos anteriores.

Arranque de un Thread

Las aplicaciones ejecutan main() tras arrancar. Esta es la razón de que main() sea el lugar natural para crear y arrancar otros hilos. La línea de código:

```
    t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
```

crea un nuevo hilo de ejecución. Los dos argumentos pasados representan el nombre del hilo y el tiempo que se desea que espere antes de imprimir el mensaje.

Al tener control directo sobre los hilos, hay que arrancarlos explícitamente. En el ejemplo con:

```
t1.start();
```

start(), en realidad es un método oculto en el hilo de ejecución que llama a run().

Manipulación de un Thread

Si todo fue bien en la creación del hilo, t1 debería contener un thread válido, que controlaremos en el método run().

Una vez dentro de run(), se pueden comenzar las sentencias de ejecución como en otros programas. run() sirve como rutina main() para los hilos; cuando run() termina, también lo hace el hilo. Todo lo que se quiera que haga el hilo de ejecución ha de estar dentro de run(), por eso cuando se dice que un método es Runnable, es obligatorio escribir un método run().

En este ejemplo, se intenta inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada a través del constructor):

```
sleep( retardo );
```

El método sleep() simplemente le dice al hilo de ejecución que duerma durante los milisegundos especificados. Se debería utilizar sleep() cuando se pretenda retrasar la ejecución del hilo. sleep() no consume recursos del sistema mientras el hilo duerme. De esta forma otros hilos pueden seguir funcionando. Una vez hecho el retardo, se imprime el mensaje "Hola Mundo!" con el nombre del hilo y el retardo.

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un hilo sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un hilo de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de los hilos de ejecución se puede utilizar el método suspend().

```
t1.suspend();
```

Este método no detiene la ejecución permanentemente. El hilo es suspendido indefinidamente y para volver a activarlo de nuevo se necesita realizar una invocación al método resume():

```
t1.resume();
```

Parada de un Thread

El último elemento de control que se necesita sobre los hilos de ejecución es el método stop(). Se utiliza para terminar la ejecución de un hilo:

```
t1.stop();
```

Esta llamada no destruye el hilo, sino que detiene su ejecución. La ejecución no se puede reanudar ya con t1.start(). Cuando se desasignen las variables que se usan en el hilo, el objeto Thread (creado con new) quedará marcado para eliminarlo y el garbage collector se encargará de liberar la memoria que utilizaba.

En el ejemplo, no se necesita detener explícitamente el hilo de ejecución. Simplemente se le deja terminar. Los programas más complejos necesitarán un control sobre cada uno de los hilos que lancen, el método stop() puede utilizarse en esas situaciones.

Si se necesita, se puede comprobar si un hilo está vivo o no; considerando vivo un hilo que ha comenzado y no ha sido detenido.

```
t1.isAlive();
```

Este método devolverá true en caso de que el hilo t1 esté vivo, es decir, ya se haya llamado a su método run() y no haya sido parado con un stop() ni haya terminado el método run() en su ejecución.

En el ejemplo no hay problemas de realizar una parada incondicional, al estar todos los hilos vivos. Pero si a un hilo de ejecución, que puede no estar vivo, se le invoca su método stop(), se generará una excepción. En este caso, en los que el estado del hilo no puede conocerse de antemano es donde se requiere el uso del método isAlive().

Grupos de Hilos

Todo hilo de ejecución en Java debe formar parte de un grupo. La clase ThreadGroup define e implementa la capacidad de un grupo de hilos.

Los grupos de hilos permiten que sea posible recoger varios hilos de ejecución en un solo objeto y manipularlo como un grupo, en vez de individualmente. Por ejemplo, se pueden regenerar los hilos de un grupo mediante una sola sentencia.

Cuando se crea un nuevo hilo, se coloca en un grupo, bien indicándolo explícitamente, o bien dejando que el sistema lo coloque en el grupo por defecto. Una vez creado el hilo y asignado a un grupo, ya no se podrá cambiar a otro grupo.

Si no se especifica un grupo en el constructor, el sistema coloca el hilo en el mismo grupo en que se encuentre el hilo de ejecución que lo haya creado, y si no se especifica en grupo para ninguno de los hilos, entonces todos serán miembros del grupo "main", que es creado por el sistema cuando arranca la aplicación Java.

En la ejecución de los ejemplos de esta sección, se ha podido observar la circunstancia anterior. Por ejemplo, el resultado en pantalla de uno de esos ejemplos es el que se reproduce a continuación:

```
% java java1002
Thread[hiloA,5,main]
Thread[hiloB,5,main]
Thread[main,5,main]
```

Como resultado de la ejecución de sentencias del tipo:

```
System.out.println( Thread.currentThread() );
```

Para presentar la información sobre el hilo de ejecución. Se puede observar que aparece el nombre del hilo, su prioridad y el nombre del grupo en que se encuentra englobado.

La clase Thread proporciona constructores en los que se puede especificar el grupo del hilo que se esta creando en el mismo momento de instanciarlo, y también métodos

como **setThreadGroup(),** que permiten determinar el grupo en que se encuentra un hilo de ejecución.

Arrancar y Parar Threads

Ahora que ya se ha visto por encima como se arrancan, paran, manipulan y agrupan los hilos de ejecución, el ejemplo un poco más gráfico, , implementa un contador.

El progama arranca un contador en 0 y lo incrementa, presentando su salida tanto en la pantalla gráfica como en la consola. Una primera ojeada al código puede dar la impresión de que el programa empezará a contar y presentará cada número, pero no es así. Una revisión más profunda del flujo de ejecución del applet, revelará su verdadera identidad.

En este caso, la clase java1004 está forzada a implementar Runnable sobre la clase Applet que extiende. Como en todos los applets, el método init() es el primero que se ejecuta. En init(), la variable contador se inicializa a cero y se crea una nueva instancia de la clase Thread. Pasándole this al constructor de Thread, el nuevo hilo ya conocerá al objeto que va a correr. En este caso this es una referencia a java1004. Después de que se haya creado el hilo, necesitamos arrancarlo. La llamada a start(), llamará a su vez al método run() de la clase, es decir, a java1004.run(). La llamada a start() retornará con éxito y el hilo comenzará a ejecutarse en ese instante. Observar que el método run() es un bucle infinito. Es infinito porque una vez que se sale de él, la ejecución del hilo se detiene. En este método se incrementará la variable contador, se duerme 10 milisegundos y envía una petición de refresco del nuevo valor al applet.

Es muy importante dormirse en algún lugar del hilo, porque sino, el hilo consumirá todo el tiempo de la CPU para su proceso y no permitirá que entren métodos de otros hilos a ejecutarse. Otra forma de detener la ejecución del hilo sería hacer una llamada al método stop(). En el contador, el hilo se detiene cuando se pulsa el ratón mientras el cursor se encuentre sobre el applet. Dependiendo de la velocidad del ordenador, se presentarán los números consecutivos o no, porque el incremento de la variable contador es independiente del refresco en pantalla. El applet no se refresca a cada petición que se le hace, sino que el sistema operativo encolará las peticiones y las que sean sucesivas las convertirá en un único refresco. Así, mientras los refrescos se van encolando, la variable contador se estará todavía incrementando, pero no se visualiza en pantalla.

El uso y la conveniencia de utilización del método stop() es un poco dudoso y algo que quizá debería evitarse, porque puede haber objetos que dependan de la ejecución de varios hilos, y si se detiene uno de ellos, puede que el objeto en cuestión estuviese en un estado no demasiado consistente, y si se le mata el hilo de control puede que definitivamente ese objeto se dañe. Una solución alternativa es el uso de una variable de control que permita saber si el hilo se encuentra en ejecución o no, por ello, en el ejemplo se utiliza la variable miThread que controla cuando el hilo está en ejecución o parado.

La clase anidada ProcesoRaton es la que se encarga de implementar un objeto receptor de los eventos de ratón, para detectar cuando el usuario pulsa alguno de los botones sobre la zona de influencia del applet.

Suspender y Reanudar Threads

Una vez que se para un hilo de ejecución, ya no se puede rearrancar con el comando start(), debido a que stop() concluirá la ejecución del hilo. Por ello, en ver de parar el hilo, lo que se puede hacer es dormirlo, llamando al método sleep(). El hilo estará suspendido un cierto tiempo y luego reanudará su ejecución cuando el límite fijado se alcance. Pero esto no es útil cuando se necesite que el hilo reanude su ejecución ante la presencia de ciertos eventos. En estos casos, el método suspend() permite que cese la ejecución del hilo y el método resume() permite que un método suspendido reanude su ejecución. En la versión modificada del ejemplo anterior, , se modifica el applet para que utilice los métodos suspend() y resume():

El uso de suspend() es crítico en ocasiones, sobre todo cuando el hilo que se va a suspender está utilizando recursos del sistema, porque en el momento de la suspensión los va a bloquear, y esos recursos seguirán bloqueados hasta que no se reanude la ejecución del hilo con resume(). Por ello, deben utilizarse métodos alternativos a estos, por ejemplo, implementando el uso de variables de control que vigiles periódicamente el estado en que se encuentra el hilo actual y obren el consecuencia.

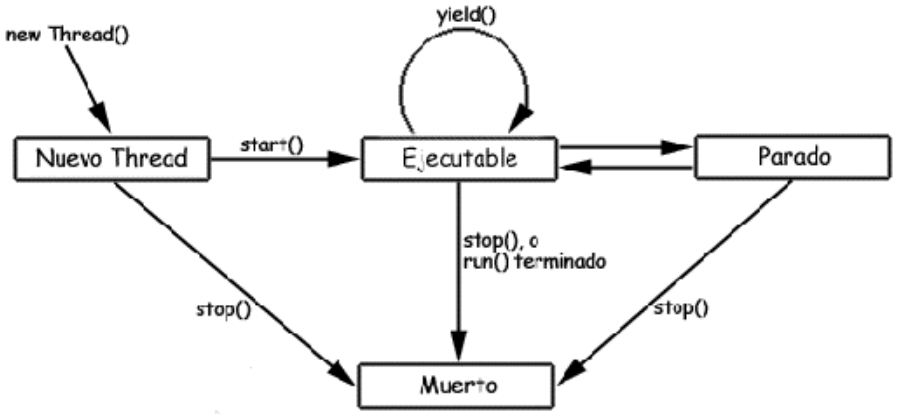
```
public class java1005 extends Applet implements Runnable {

    ...

class ProcesoRaton extends MouseAdapter {
    boolean suspendido;

    public void mousePressed( MouseEvent evt ) {
    if( suspendido )
            t.resume();
        else
            t.suspend();
        suspendido = !suspendido;
        }
    }
     ...
```

Para controlar el estado del applet, se ha modificado el funcionamiento del objeto Listener que recibe los eventos del ratón, en donde se ha introducido la variable suspendido. Diferenciar los distintos estados de ejecución del applet es importante porque algunos métodos pueden generar excepciones si se llaman desde un estado erróneo. Por ejemplo, si el applet ha sido arrancado y se detiene con stop(), si se intenta ejecutar el método start(), se generará una excepción IllegalThreadStateException.

Aquí podemos poner de nuevo en cuarentena la idoneidad del uso de estos métodos para el control del estado del hilo de ejecución, tanto por lo comentado del posible bloqueo de recursos vitales del sistema, como porque se puede generar un punto muerto en el sistema si el hilo de ejecución que va a intentar revivir el hilo suspendido necesita del recurso bloqueado. Por ello, es más seguro el uso de una variable de control como suspendido, de tal forma que sea ella quien controle el estado del hilo y utilizar el método notify() para indicar cuando el hilo vuelve a la vida.

Estados de un hilo de ejecución

Durante el ciclo de vida de un hilo, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de un hilo.



Nuevo Thread

La siguiente sentencia crea un nuevo hilo de ejecución pero no lo arranca, lo deja en el estado de Nuevo Thread:

```
Thread MiThread = new MiClaseThread();
Thread MiThread = new Thread( new UnaClaseThread,"hiloA" );
```

Cuando un hilo está en este estado, es simplemente un objeto Thread vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método start(), o detenerse definitivamente, llamando al método stop(); la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo IllegalThreadStateException.

Ejecutable

Ahora obsérvense las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método start() creará los recursos del sistema necesarios para que el hilo puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del hilo de ejecución. En este momento se encuentra en el estado Ejecutable del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el hilo está aquí no esta corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los hilos estén corriendo al mismo tiempo. Java implementa un tipo de scheduling o lista de procesos, que permite que el procesador sea compartido entre todos los procesos o hilos que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, se puede considerar

que este estado es realmente un estado En Ejecución, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando el hilo se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método run(), se ejecutarán secuencialmente.

Parado

El hilo de ejecución entra en estado Parado cuando alguien llama al método suspend(), cuando se llama al método sleep(), cuando el hilo está bloqueado en un proceso de entrada/salida o cuando el hilo utiliza su método wait() para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el hilo estará Parado.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
```

la línea de código que llama al método sleep():

```
MiThread.sleep( 10000 );
```

hace que el hilo se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread volvería a estar en estado Ejecutable y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado Parado, hay una forma específica de volver a estado Ejecutable. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el hilo ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado Ejecutable. Llamar al método resume() mientras esté el hilo durmiendo no serviría para nada.

Los métodos de recuperación del estado Ejecutable, en función de la forma de llegar al estado Parado del hilo, son los siguientes:

- Si un hilo está dormido, pasado el lapso de tiempo
- Si un hilo de ejecución está suspendido, después de una llamada a su método resume()
- Si un hilo está bloqueado en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución
- Si un hilo está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método notify() o notifyAll()

Muerto

Un hilo de ejecución se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). Un hilo muere normalmente cuando concluye de forma habitual su método run(). Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
    int i=0;
    while( i < 20 ) {
        i++;
        System.out.println( "i = "+i );
        }
    }
```

Un hilo que contenga a este método run(), morirá naturalmente después de que se complete el bucle y run() concluya.

También se puede matar en cualquier momento un hilo, invocando a su método stop(). En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
MiThread.stop();
```

se crea y arranca el hilo MiThread, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método stop(), lo mata.

El método stop() envía un objeto ThreadDeath al hilo de ejecución que quiere detener. Así, cuando un hilo es parado de este modo, muere asíncronamente. El hilo morirá en el momento en que reciba ese objeto *Th*readDeath.

Los applets utilizarán el método stop() para matar a todos sus hilos cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

El método isAlive()

El interfaz de programación de la clase Thread incluye el método isAlive(), que devuelve true si el hilo ha sido arrancado (con start()) y no ha sido detenido (con stop()). Por ello, si el método isAlive() devuelve false, sabemos que estamos ante un Nuevo Thread o ante un thread Muerto. Si devuelve true, se sabe que el hilo se encuentra en estado Ejecutable o Parado. No se puede diferenciar entre Nuevo Thread y Muerto, ni entre un hilo Ejecutable o Parado.

SCHEDULING

Java tiene un Scheduler, una lista de procesos, que monitoriza todos los hilos que se están ejecutando en todos los programas y decide cuales deben ejecutarse y cuales deben encontrarse preparados para su ejecución. Hay dos características de los hilos que el scheduler identifica en este proceso de decisión. Una, la más importante, es la prioridad del

hilo de ejecución; la otra, es el indicador de demonio. La regla básica del scheduler es que si solamente hay hilos demonio ejecutándose, la Máquina Virtual Java (JVM) concluirá. Los nuevos hilos heredan la prioridad y el indicador de demonio de los hilos de ejecución que los han creado. El scheduler determina qué hilos deberán ejecutarse comprobando la prioridad de todos ellos, aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

El scheduler puede seguir dos patrones, preemptivo y no-preemptivo. Los schedulers preemtivos proporcionan un segmento de tiempo a todos los hilos que están corriendo en el sistema. El scheduler decide cual será el siguiente hilo a ejecutarse y llama al método resume() para darle vida durante un período fijo de tiempo. Cuando el hilo ha estado en ejecución ese período de tiempo, se llama a suspend() y el siguiente hilo de ejecución en la lista de procesos será relanzado (resume()). Los schedulers no-preemtivos deciden que hilo debe correr y lo ejecutan hasta que concluye. El hilo tiene control total sobre el sistema mientras esté en ejecución. El método yield() es la forma en que un hilo fuerza al scheduler a comenzar la ejecución de otro hilo que esté esperando. Dependiendo del sistema en que esté corriendo Java, el scheduler será de un tipo u otro, preemptivo o no-preemptivo.

Prioridades

El scheduler determina el hilo que debe ejecutarse en función de la prioridad asignada a cada uno de ellos. El rango de prioridades oscila entre 1 y 10. La prioridad por defecto de un hilo de ejecución es NORM_PRIORITY, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son MIN_PRORITY, fijada a 1, y MAX_PRIORITY, que tiene un valor de 10. El método getPriority() puede utilizarse para conocer el valor actual de la prioridad de un hilo.

Hilos Demonio

Los hilos de ejecución demonio también se llaman servicios, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida.

Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo.Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (garbage collector). Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Un hilo puede fijar su indicador de demonio pasando un valor true al método setDaemon(). Si se pasa false a este método, el hilo de ejecución será devuelto por el sistema como un hilo de usuario. No obstante, esto último debe realizarse antes de que se arranque el hilo de ejecución (start()). Si se quiere saber si un hilo es un hilo demonio, se utilizará el método isDaemon().

Diferencia entre hilos y fork()

fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si no hay problemas de cantidad de memoria de la máquina y se dispone de una CPU poderosa, y siempre que se mantenga el número de

procesos hijos dentro de un límite manejable, porque se hace un uso intensivo de los recursos del sistema. Los applets Java no pueden lanzar ningún proceso en el cliente, porque eso sería una fuente de inseguridad y no está permitido. Las aplicaciones y los applets deben utilizar hilos de ejecución.

La multitarea pre-emptiva tiene sus problemas. Un hilo puede interrumpir a otro en cualquier momento, de ahí lo de pre-emptive. Fácilmente puede el lector imaginarse lo que pasaría si un hilo de ejecución está escribiendo en un array, mientras otro hilo lo interrumpe y comienza a escribir en el mismo array. Los lenguajes como C y C++ necesitan de las funciones lock() y unlock() para antes y después de leer o escribir datos. Java también funciona de este modo, pero oculta el bloqueo de datos bajo la sentencia synchronized:

```
synchronized int MiMetodo();
```

Otro área en que los hilos son muy útiles es en los interfaces de usuario. Permiten incrementar la respuesta del ordenador ante el usuario cuando se encuentra realizando complicados cálculos y no puede atender a la entrada de usuario. Estos cálculos se pueden realizar en segundo plano, o realizar varios en primer plano (música y animaciones) sin que se dé apariencia de pérdida de rendimiento.

Ejemplo de animación

Este es un ejemplo de un applet, , que crea un hilo de animación que nos presenta el globo terráqueo en rotación. Aquí se puede ver que el applet crea un hilo de ejecución de sí mismo, concurrencia. Además, animacion.start() llama al start() del hilo, no del applet, que automáticamente llamará a run():

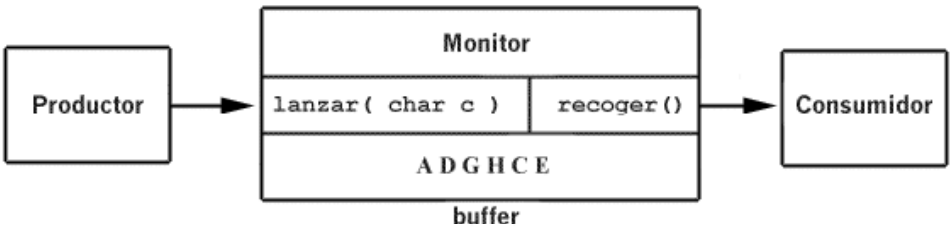
```
import java.awt.*;
import java.applet.Applet;

public class java1006 extends Applet implements Runnable {
    Image imagenes[];
    MediaTracker tracker;
    int indice = 0;
    Thread animacion;

    int maxAncho,maxAlto;
    Image offScrImage; // Componente off-screen para doble buffering
    Graphics offScrGC;

    // Nos indicará si ya se puede pintar
    boolean cargado = false;

    // Inicializamos el applet, establecemos su tamaño y
    // cargamos las imágenes
    public void init() {
        // Establecemos el supervisor de imágenes
        tracker = new MediaTracker( this );
        // Fijamos el tamaño del applet
        maxAncho = 100;
        maxAlto = 100;
        imagenes = new Image[33];

        // Establecemos el doble buffer y dimensionamos el applet
        try {
            offScrImage = createImage( maxAncho,maxAlto );
            offScrGC = offScrImage.getGraphics();
            offScrGC.setColor( Color.lightGray );
            offScrGC.fillRect( 0,0,maxAncho,maxAlto );
```

```
            resize( maxAncho,maxAlto );
        } catch( Exception e ) {
            e.printStackTrace();
            }

        // Cargamos las imágenes en un array
        for( int i=0; i < 33; i++ )
            {
            String fichero =
                new String( "Tierra"+String.valueOf(i+1)+".gif" );
            imagenes[i] = getImage( getDocumentBase(),fichero );
            // Registramos las imágenes con el tracker
            tracker.addImage( imagenes[i],i );
            }

        try {
            // Utilizamos el tracker para comprobar que todas las
            // imágenes están cargadas
            tracker.waitForAll();
        } catch( InterruptedException e ) {
            ;
            }
        cargado = true;
        }

    // Pintamos el fotograma que corresponda
    public void paint( Graphics g ) {
        if( cargado )
            g.drawImage( offScrImage,0,0,this );
        }

    // Arrancamos y establecemos la primera imagen
    public void start() {
        if( tracker.checkID( indice ) )
            offScrGC.drawImage( imagenes[indice],0,0,this );
        animacion = new Thread( this );
        animacion.start();
        }

    // Aquí hacemos el trabajo de animación
    // Muestra una imagen, para, muestra la siguiente...
    public void run() {
        // Obtiene el identificador del thread
        Thread thActual = Thread.currentThread();

        // Nos aseguramos de que se ejecuta cuando estamos en un
        // thread y además es el actual
        while( animacion != null  &&  animacion == thActual )
            {
            if( tracker.checkID( indice ) )
                {
                // Obtenemos la siguiente imagen
                offScrGC.drawImage( imagenes[indice],0,0,this );
                indice++;
                // Volvemos al principio y seguimos, para el bucle
                if( indice >= imagenes.length )
                    indice = 0;
                }

            // Ralentizamos la animación para que parezca normal
            try {
                animacion.sleep( 200 );
            } catch( InterruptedException e ) {
                ;
                }
```

```
            // Pintamos el siguiente fotograma
            repaint();
            }
        }
    }
```

En el ejemplo se pueden observar más cosas. La variable thActual es propia de cada hilo que se lance, y la variable animacion la estarán viendo todos los hilos. No hay duplicidad de procesos, sino que todos comparten las mismas variables; cada hilo de ejecución, sin embargo, tiene su pila local de variables, que no comparte con nadie y que son las que están declaradas dentro de las llaves del método run().

La excepción InterruptedExcepcion salta en el caso en que se haya tenido al hilo parado más tiempo del debido. Es imprescindible recoger esta excepción cuando se están implementando hilos de ejecución, tanto es así, que en el caso de no recogerla, el compilador generará un error.



Comunicación entre Hilos

Otra clave para el éxito y la ventaja de la utilización de múltiples hilos de ejecución en una aplicación, o aplicación multithreaded, es que pueden comunicarse entre sí. Se pueden diseñar hilos para utilizar objetos comunes, que cada hilo puede manipular independientemente de los otros hilos de ejecución.

El ejemplo clásico de comunicación de hilos de ejecución es un modelo productor/consumidor. Un hilo produce una salida, que otro hilo usa (consume), sea lo que sea esa salida. Entonces se crea un productor, que será un hilo que irá sacando caracteres por su salida; y se crea también un consumidor que irá recogiendo los caracteres que vaya sacando el productor y un monitor que controlará el proceso de sincronización entre los hilos de ejecución. Funcionará como una tubería, insertando el productor caracteres en un extremo y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.

Productor

El productor extenderá la clase Thread, y su código es el siguiente:

```
class Productor extends Thread {
    private Tuberia tuberia;
    private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    public Productor( Tuberia t ) {
        // Mantiene una copia propia del objeto compartido
        tuberia = t;
        }

    public void run() {
```

```
        char c;

        // Mete 10 letras en la tubería
        for( int i=0; i < 10; i++ )
            {
            c = alfabeto.charAt( (int)(Math.random()*26 ) );
            tuberia.lanzar( c );
            // Imprime un registro con lo añadido
            System.out.println( "Lanzado "+c+" a la tuberia." );
            // Espera un poco antes de añadir más letras
            try {
                sleep( (int)(Math.random() * 100 ) );
            } catch( InterruptedException e ) {;}
            }
        }
    }
```

Notar que se crea una instancia de la clase Tuberia, y que se utiliza el método tuberia.lanzar() para que se vaya construyendo la tubería, en principio de 10 caracteres.

Consumidor

Ahora se reproduce el código del consumidor, que también extenderá la clase Thread:

```
class Consumidor extends Thread {
    private Tuberia tuberia;

    public Consumidor( Tuberia t ) {
        // Mantiene una copia propia del objeto compartido
        tuberia = t;
        }

    public void run() {
        char c;

        // Consume 10 letras de la tubería
        for( int i=0; i < 10; i++ )
            {
            c = tuberia.recoger();
            // Imprime las letras retiradas
            System.out.println( "Recogido el caracter "+c );
            // Espera un poco antes de coger más letras
            try {
                sleep( (int)(Math.random() * 2000 ) );
            } catch( InterruptedException e ) {;}
            }
        }
    }
```

En este caso, como en el del productor, se cuenta con un método en la clase Tuberia, tuberia.recoger(), para manejar la información.

Monitor

Una vez vistos el productor de la información y el consumidor, solamente queda por ver qué es lo que hace la clase Tuberia.

Lo que realiza la clase Tuberia, es una función de supervisión de las transacciones entre los dos hilos de ejecución, el productor y el consumidor. Los monitores, en general, son piezas muy importantes de las aplicaciones multihilo, porque mantienen el flujo de comunicación entre los hilos.

```
class Tuberia {
    private char buffer[] = new char[6];
    private int siguiente = 0;
    // Flags para saber el estado del buffer
    private boolean estaLlena = false;
    private boolean estaVacia = true;

    // Método para retirar letras del buffer
    public synchronized char recoger() {
        // No se puede consumir si el buffer está vacío
        while( estaVacia == true )
            {
            try {
                wait(); // Se sale cuando estaVacia cambia a false
            } catch( InterruptedException e ) {
                ;
                }
            }
        // Decrementa la cuenta, ya que va a consumir una letra
        siguiente--;
        // Comprueba si se retiró la última letra
        if( siguiente == 0 )
            estaVacia = true;
        // El buffer no puede estar lleno, porque acabamos
        // de consumir
        estaLlena = false;
        notify();

        // Devuelve la letra al thread consumidor
        return( buffer[siguiente] );
        }

    // Método para añadir letras al buffer
    public synchronized void lanzar( char c ) {
        // Espera hasta que haya sitio para otra letra
        while( estaLlena == true )
            {
            try {
                wait(); // Se sale cuando estaLlena cambia a false
            } catch( InterruptedException e ) {
                ;
                }
            }
        // Añade una letra en el primer lugar disponible
        buffer[siguiente] = c;
        // Cambia al siguiente lugar disponible
        siguiente++;
        // Comprueba si el buffer está lleno
        if( siguiente == 6 )
            estaLlena = true;
        estaVacia = false;
        notify();
        }
}
```

En la clase Tuberia se pueden observar dos características importantes: los miembros dato (buffer[]) son privados, y los métodos de acceso (lanzar() y recoger()) son sincronizados.

Aquí se observa que la variable estaVacia es un semáforo, como los de toda la vida. La naturaleza privada de los datos evita que el productor y el consumidor accedan directamente a éstos. Si se permitiese el acceso directo de ambos hilos de ejecución a los

datos, se podrían producir problemas; por ejemplo, si el consumidor intenta retirar datos de un buffer vacío, obtendrá excepciones innecesarias, o se bloqueará el proceso.

Los métodos sincronizados de acceso impiden que los productores y consumidores corrompan un objeto compartido. Mientras el productor está añadiendo una letra a la tubería, el consumidor no la puede retirar y viceversa. Esta sincronización es vital para mantener la integridad de cualquier objeto compartido. No sería lo mismo sincronizar la clase en vez de los métodos, porque esto significaría que nadie puede acceder a las variables de la clase en paralelo, mientras que al sincronizar los métodos, sí pueden acceder a todas las variables que están fuera de los métodos que pertenecen a la clase.

Se pueden sincronizar incluso variables, para realizar alguna acción determinada sobre ellas, por ejemplo:

```
sincronized( p ) {
    // aquí se colocaría el código
    // los threads que estén intentando acceder a p se pararán
    // y generarán una InterruptedException
    }
```

El método notify() al final de cada método de acceso avisa a cualquier proceso que esté esperando por el objeto, entonces el proceso que ha estado esperando intentará acceder de nuevo al objeto. En el método wait() se hace que el hilo se quede a la espera de que le llegue un notify(), ya sea enviado por el hilo de ejecución o por el sistema.

Ahora que ya se dispone de un productor, un consumidor y un objeto compartido, se necesita una aplicación que arranque los hilos y que consiga que todos hablen con el mismo objeto que están compartiendo. Esto es lo que hace el siguiente trozo de código, del fuente :

```
class java1007 {
    public static void main( String args[] ) {
        Tuberia t = new Tuberia();
        Productor p = new Productor( t );
        Consumidor c = new Consumidor( t );

        p.start();
        c.start();
        }
    }
```

Compilando y ejecutando esta aplicación, se podrá observar en modelo que se ha diseñado en pleno funcionamiento.

Monitorización del Productor

Los programas productor/consumidor a menudo emplean monitorización remota, que permite al consumidor observar el hilo del productor interaccionando con un usuario o con otra parte del sistema. Por ejemplo, en una red, un grupo de hilos de ejecución productores podrían trabajar cada uno en una workstation. Los productores imprimirían documentos, almacenando una entrada en un registro (log). Un consumidor (o múltiples consumidores) podría procesar el registro y realizar durante la noche un informe de la actividad de impresión del día anterior.

Otro ejemplo, a pequeña escala podría ser el uso de varias ventanas en una workstation. Una ventana se puede usar para la entrada de información (el productor), y otra ventana reaccionaría a esa información (el consumidor).

Peer, es un observador general del sistema.

Applets

Appletviewer

El visualizador de applets (appletviewer) es una aplicación que permite ver en funcionamiento applets, sin necesidad de la utilización de un navegador World-Wide-Web como HotJava, Microsoft Explorer o Nescape. En adelante, se recurrirá muchas veces a él, ya que el objetivo de este Tutorial es el lenguaje Java, y es la forma más sencilla y económica de poder ver un applet en ejecución.

Applet

La definición más extendida de applet, muy bien resumida por Patrick Naughton, indica que un applet es "una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta in situ como parte de un documento web". Claro que así la definición establece el entorno (Internet, Web, etc.). En realidad, un applet es una aplicación pretendidamente corta (nada impide que ocupe más de un gigabyte, a no ser el pensamiento de que se va a transportar por la red y una mente sensata) basada en un formato gráfico sin representación indepediente: es decir, se trata de un elemento a embeber en otras aplicaciones; es un componente en su sentido estricto.

Pues bien, así es un applet. Lo que ocurre es que, dado que no existe una base adecuada para soportar aplicaciones industriales Java en las que insertar estas miniaplicaciones (aunque todo se andará), los applets se han construido mayoritariamente, y con gran acierto comercial (parece), como pequeñas aplicaciones interactivas, con movimiento, luces y sonido... en Internet.

Llamadas a Applets con appletviewer

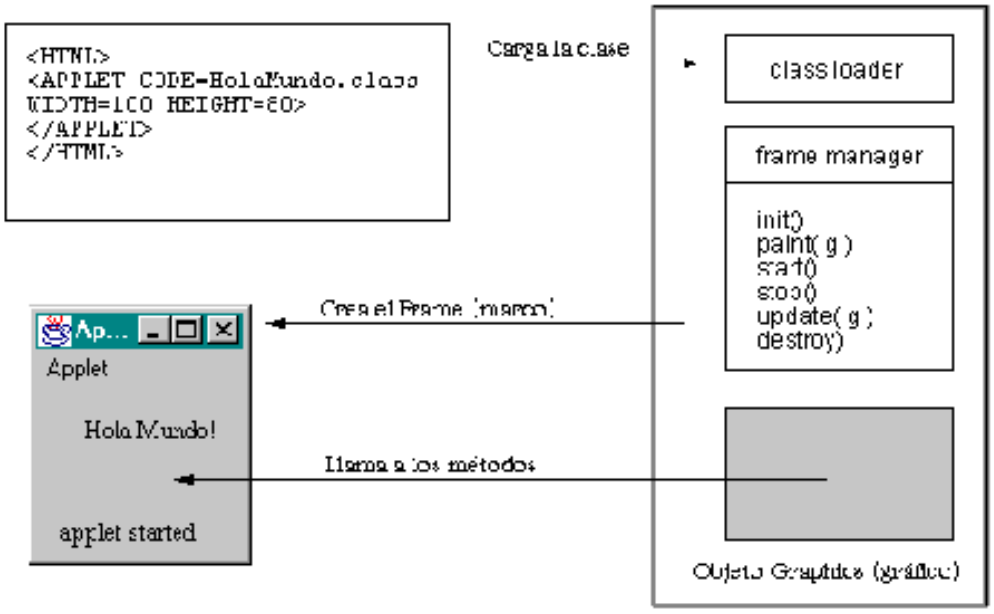
Al ser un applet una mínima aplicación Java diseñada para ejecutarse en un navegador Web, no necesita por tanto preocuparse por un método main() ni en dónde se realizan las llamadas. El applet asume que el código se está ejecutando desde dentro de un navegador. El appletviewer se asemeja al mínimo navegador. Espera como argumento el nombre del fichero html a cargar, no se le puede pasar directamente un programa Java. Este fichero html debe contener una marca que especifica el código que cargará el appletviewer:

```
<HTML>
<APPLET CODE=HolaMundo.class WIDTH=300 HEIGHT=100>
</APPLET>
</HTML>
```

El appletviewer creará un espacio de navegación, incluyendo un área gráfica, donde se ejecutará el applet, entonces llamará a la clase applet apropiada. En el ejemplo anterior, el appletviewer cargará una clase de nombre HolaMundo y le permitirá trabajar en su espacio gráfico.

Arquitectura de appletviewer

El Appletviewer representa el mínimo interfaz de navegación. En la figura se muestran los pasos que seguiría appletviewer para presentar el resultado de la ejecución del código de la clase del ejemplo.



Esta es una visión simplificada del appletviewer. La función principal de esta aplicación es proporcionar al usuario un objeto de tipo Graphics sobre el que dibujar, y varias funciones para facilitar el uso del objeto Graphics.

Cuando un applet se carga en el appletviewer, comienza su ciclo de vida, que pasaría por las siguientes fases:

- Se crea una instancia de la clase que controla el applet. En el ejemplo de la figura anterior, sería la clase HolaMundo
- El applet se incializa
- El applet comienza a ejecutarse
- El applet empieza a recibir llamadas. Primero recibe una llamada init (inicializar), seguida de un mensaje start (empezar) y paint (pintar). Estas llamadas pueden ser recibidas asíncronamente.

Métodos de appletviewer

A continuación se utiliza como excusa la función asociada al appletviewer de los siguientes métodos para intentar un acercamiento a su presentación, aunque a lo largo de secciones posteriores, se harán de nuevo múltiples referencias a ellos, porque también son los métodos propios de la clase Applet, que se utilizará en muchos ejemplos.

init()

El método init() se llama cada vez que el appletviewer carga por primera vez la clase. Si el applet llamado no lo sobrecarga, init() no hace nada. Fundamentalmente en este método se debe fijar el tamaño del applet, aunque en el caso de Netscape el tamaño que vale es el que se indique en la línea del fichero html que cargue el applet. También se

deben realizar en este método las cargas de imágenes y sonidos necesarios para la ejecución del applet. Y, por supuesto, la asignación de valores a las variables globales a la clase que se utilicen. En el caso de los applet, este método únicamente es llamado por el sistema al cargar el applet.

start()

start() es la llamada para arrancar el applet cada vez que es visitado. La clase Applet no hace nada en este método. Las clases derivadas deben sobrecargarlo para comenzar la animación, el sonido, etc. Esta función es llamada automáticamente cada vez que la zona de visualización en que está ubicado el applet se expone a la visión, a fin de optimizar en uso de los recursos del sistema y no ejecutar algo que no puede ser apreciado (aunque es potestad del programador el poder variar este comportamiento impuesto por defecto, y hacer que un applet siga activo incluso cuando se encuentre fuera del área de visión).

Esto es, imagínese la carga de un applet en un navegador minimizado; el sistema llamará al método init(), pero no a start(), que sí será llamado cuando se restaure el navegador a un tamaño que permita ver el applet. Naturalmente, start() se puede ejecutar varias veces: la primera tras init() y las siguientes (porque init() se ejecuta solamente una vez) tras haber aplicado al applet el método stop().

stop()

stop() es la llamada para detener la ejecución del applet. Se llama cuando el applet desaparece de la pantalla. La clase Applet tampoco hace nada en este método, que debería ser sobrecargado por las clases derivadas para detener la animación, el sonido, etc. Este método es llamado cuando el navegador no incluye en su campo de visión al applet; por ejemplo, cuando abandona la página en que está insertado, de forma que el programador puede paralizar los threads que no resulten necesarios respecto de un applet no visible, y luego recuperar su actividad mediante el método start().

destroy()

El método destroy() se llama cuando ya no se va a utilizar más el applet, cuando se necesita que sean liberados todos los recursos dispuestos por el applet, por ejemplo, cuando se cierra el navegador. La clase Applet no hace nada en este método. Las clases derivadas deberían sobrecargarlo para hacer una limpieza final. Los applet multithread deberían utilizar destroy() para detener los threads que quedasen activos.

El appletviewer también contiene la clase Component (componente), que usa dos métodos para ayudar al applet a escribir en el espacio gráfico que el appletviewer le proporciona para su ejecución.

paint()

Es la función llamada cada vez que el área de dibujo del applet necesita ser refrescada. La clase Applet simplemente dibuja un rectángulo gris en el área, es la clase derivada, obviamente, la que debería sobrecargar este método para representar algo inteligente en la pantalla. Cada vez que la zona del applet es cubierta por otra ventana, se desplaza el applet fuera de la visión o el applet cambia de posición debido a un

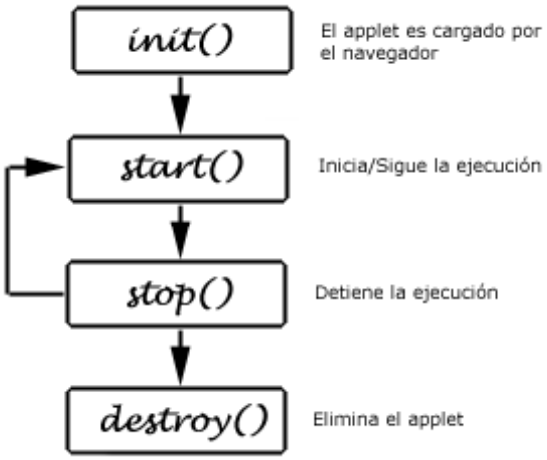
redimensionamiento del navegador, el sistema llama automáticamente a este método, pasando como argumento un objeto de tipo Graphics que delimita la zona a ser pintada; en realidad se pasa una referencia al contexto gráfico en uso, y que representa la ventana del applet en la página web.

update()

Esta es la función que realmente se llama cuando se necesita una actualización de la pantalla. La clase Applet simplemente limpia el área y llama al método paint(). Esta funcionalidad es suficiente para la mayoría de los casos; aunque, de cualquier forma, las clases derivadas pueden sustituir esta funcionalidad para sus propósitos especiales. Es decir, en las situaciones detalladas anteriormente que dañan la zona de exposición del applet, el sistema llama al método paint(), pero en realidad la llamada se realiza al método update(), cuyo comportamiento establecido en la clase Component es llamar al método paint(), tras haber rellenado la zona del applet con su color de fondo por defecto. Pudiera parecer así que se trata de un método de efecto neutro, pero si la función paint() cambiara el color del fondo, se podría percibir un flick de cambio de colores nada agradable. Por tanto, habrá que cuidarse por lo común, de eliminar este efecto de limpia primero, sobrecargando el método update(), para que llame únicamente a paint(). Otra solución sería insertar el código de pintado en una sobrecarga del método update() y escribir un método paint() que sólo llame a update(). La última solución pasaría por usar el mismo método setBackground(Color), en el método init() para así evitar el efecto visual sin tener que sobrecargar el método update(). Estas son las mismas razones que aconsejan usar el método resize() inserto en init(), para evitar el mismo desagradable efecto.

repaint()

Llamando a este método se podrá forzar la actualización del applet, la llamada a update(). Pero hay que tener cuidado, AWT posee cierta inteligencia (combinación casi siempre nefasta), de forma que si se llama a update() mediante repaint() con una frecuencia muy corta, AWT ignorará las llamadas a update() que estime oportuno, pues considera a esta función un bien escaso.



La figura anterior muestra el ciclo habitual en que se mueve la ejecución, visualización y destrucción de los applets.

Sinopsis

La llamada a appletviewer es de la forma:

```
appletviewer [-debug] urls...
```

El appletviewer toma como parámetro de ejecución, o bien el nombre del un fichero html conteniendo el tag (marca) <APPLET>, o bien un URL hacia un fichero HTML que contenga esa marca. Si el fichero html no contiene un tag <APPLET> válido, el appletviewer no hará nada. El appletviewer no muestra otras marcas html.

La única opción válida que admite la llamada a appletviewer es -debug, que arranca el applet en el depurador de Java, jdb. Para poder ver el código fuente en el depurador, se debe compilar ese fichero fuente .java con la opción -g.

Ejemplo de uso

En el ejemplo de llamada al appletviewer que se muestra a continuación, se consigue que se ejecute el applet que se creará en la sección siguiente y que se lanzará desde un fichero html del mismo nombre que nuestro fichero de código fuente Java.

```
%appletviewer HolaMundo.html
```

Funciones de menú de appletviewer

El appletviewer tiene un único menú, tal como se muestra en la imagen siguiente, y que se explica a continuación en sus opciones más importantes, ya que se usará a menudo cuando se vaya avanzando en conocimientos acerca de Java.

- Restart

La función Restart llama al método stop() y seguidamente llama de nuevo a start(), que es el método que ha lanzado inicialmente la ejecución del applet. Se puede utilizar Restart para simular el movimiento entre páginas en un documento html, de tal modo que se pueden reproducir las secuencias de parada y arranque del applet que se producirían al desplazarse a otra página y volver a la que contiene el applet.

- Reload

La función Reload llama al método stop() y luego al método destroy() en el applet actual. A continuación carga una nueva copia del applet y la arranca llamando al método start().

- Stop

La función Stop detiene la ejecución del applet. Es como si se hubiese lanzado el appletviewer con el applet cargado, pero sin ejcutarse. Llama al método stop().

- Save

La función Save permite grabar la representación actual del applet, de forma que posteriormente se puede leer el fichero en que se salva y recuperar el estado en que se

encontraban los objetos que componían el applet. Esto se verá en detalle al tratar la serialización de objetos en Java.

- Start

La función Start llama al método start(), de forma que si el applet ya estaba en ejecución, no hace nada, pero si estaba detenido con stop(), lo pone de nuevo en marcha.

- Clone

La función Clone crea una copia del applet actual en una ventana de appletviewer nueva. En realidad es un appletviewer idéntico con el mismo URL.

- Tag

La función Tag muestra en una ventana hija del appletviewer el código html cargado para su ejecución. Es similar a la función View Source que figura en la mayoría de los navegadores, Netscape y HotJava incluidos.

- Info

La función Info lee los comentarios de documentación contenidos en el fichero html y muestra la información de los parámetros (si la hay).

- Properties

El appletviewer tiene las funciones básicas de presentación de un navegador y la función Properties (propiedades de la red) permite cambiar o establecer el modo de seguridad o fijar los servidores de proxy o firewall.

- Close

La función Close llama al método destroy() de la ventana actual del appletviewer, teminando su ejecución.

- Quit

La función Quit llama al método destroy() de cada una de las copias del appletviewer que se encuentren lanzadas, concluyendo la ejecución de todas ellas y terminando entonces el appletviewer.

La marca Applet de HTML

Dado que los applets están mayormente destinados a ejecutarse en navegadores Web, había que preparar el lenguaje HTML para soportar Java, o mejor, los applets. El esquema de marcas de HTML, y la evolución del estándar marcado por Netscape hicieron fácil la adición de una nueva marca que permitiera, una vez añadido el correspondiente código gestor en los navegadores, la ejecución de programas Java en ellos.

La sintaxis de las etiquetas <APPLET> y <PARAM> es la que se muestra a continuación y que se irá explicando en detalle a través de los párrafos posteriores:

```
<APPLET CODE= WIDTH= HEIGTH= [CODEBASE=] [ALT=] [NAME=]
```

```
[ALIGN=] [VSPACE=] [HSPACE=]>
<PARAM NAME= VALUE= >
</APPLET>
```

*Atrib*utos obligatorios:

CODE	Nombre de la clase principal
WIDTH	Anchura inicial
HEIGHT	Altura inicial

Atributos opcionales:

CODEBASE	URL base del applet
ALT	Texto alternativo
NAME	Nombre de la instancia
ALIGN	Justificación del applet
VSPACE	Espaciado vertical
HSPACE	Espaciado horizontal

Los applets se incluyen en las páginas Web a través de la marca <APPLET>, que para quien conozca html resultará muy similar a la marca <IMG>. Ambas necesitan la referencia a un fichero fuente que no forma parte de la página en que se encuentran embebidos. IMG hace esto a través de SRC=parámetro y APPLET lo hace a través CODE=parámetro. El parámetro de CODE indica al navegador dónde se encuentra el fichero con el código Java compilado .class. Es una localización relativa al documento fuente.

Para proporcionar parámetros a un applet, que al fin y al cabo es un programa ejecutable, se define otra nueva etiqueta html: <PARAM>. Estas etiquetas no son estándares, por lo que son ignoradas por aquellos navegadores que no sean Java Compatibles.

Por razones que no entiendo muy bien, pero posiblemente relacionadas con los packages y classpaths, si un applet reside en un directorio diferente del que contiene a la página en que se encuentra embebido, entonces no se indica un URL a esta localización, sino que se apunta al directorio del fichero .class utilizando el parámetro CODEBASE, aunque todavía se puede usar CODE para proporcionar el nombre del fichero .class.

Al igual que IMG, APPLET tiene una serie de parámetros que lo posicionan en la página. WIDTH y HEIGHT especifican el tamaño del rectángulo que contendrá al applet, se indican en pixels. ALIGN funciona igual que con IMG (en los navegadores que lo soportan), definiendo cómo se posiciona el rectángulo del applet con respecto a los otros elementos de la página. Los valores posibles a especificar son: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM y ABSBOTTOM. Y, finalmente, lo mismo que con IMG, se puede especificar un HSPACE y un VSPACE en pixels para indicar la cantidad de espacio vacío que habrá de separación entre el applet y el texto que le rodea.

APPLET tiene una marca ALT. La utilizaría un navegador que entendiese la marca APPLET, pero que por alguna razón, no pudiese ejecutarlo. Por ejemplo, si un applet necesita escribir en el disco duro del ordenador, pero en las características de seguridad tenemos bloqueada esa posibilidad, entonces el navegador presentaría el texto asociado a ALT.

ALT no es utilizado por los navegadores que no tratan la marca APPLET, por ello se ha definido la marca </APPLET>, que finaliza la descripción del applet. Un navegador con soporte Java ignorará todo el texto que haya entre las marcas <APPLET> y </APPLET>, sin embargo, un navegador que no soporte Java ignorará las marcas y presentará el texto que haya entre ellas.

Atributos de APPLET

Los atributos que acompañan a la etiqueta <APPLET>, algunos son obligatorios y otros son opcionales. Todos los atributos, siguiendo la sintaxis de html, se especifican de la forma: atributo=valor. Los atributos obligatorios son:

CODE

Indica el fichero que contiene el applet, la clase ejecutable, que tiene la extensión .class. No se permite un URL absoluto, como ya se ha dicho, aunque sí puede ser relativo al atributo opcional CODEBASE.

OBJECT

Indica el nombre del fichero que contiene la representación serializada (grabada en disco) del Applet. El applet será reconstruido, pero no se invocará el método init(), sino que se invoca a start(). No se restauran los atributos de cuando el applet fue serializado, aunque fuesen válidos, por lo que un applet debería ser detenido invocando a su método stop(), antes de ser serializado. No puede estar presente este atributo si está presente CODE, solamente uno de ellos puede utilizarse a la vez.

WIDTH

Indica la anchura inicial que el navegador debe reservar para el applet en pixels.

HEIGHT

Indica la altura inicial en pixels. Un applet que disponga de una geometría fija no se verá redimensionado por estos atributos. Por ello, si los atributos definen una zona menor que la que el applet utiliza, únicamente se verá parte del mismo, como si se visualiza a través de una ventana, eso sí, sin ningún tipo de desplazamiento.

Los **atributos opcionales** que pueden acompañar a la marca **APPLET** comprenden los que se indican a continuación:

CODEBASE

Se emplea para utilizar el URL base del applet. En caso de no especificarse, se utilizará el mismo que tiene el documento html.

ARCHIVE

Describe uno o más archivos que contengan clases u otros recursos, que serán precargados antes de iniciar la ejecución del applet. Si se incluyen clases en el archivo que indique este atributo, éstas se cargarán utilizando una instancia del cargados de clases (AppletClassLoader), que usará el contenido del atributo CODEBASE. Si se colocar varios ficheros, se deben separar con comas (,).

ALT

Como ya se ha dicho, funciona exactamente igual que el ALT de la marca <IMG>, es decir, muestra un texto alternativo, en este caso al applet, en navegadores en modo texto o que entiendan la etiqueta APPLET pero no implementen una máquina virtual Java.

NAME

Otorga un nombre simbólico a esta instancia del applet en la página, que puede ser empleado por otros applets de la misma página para localizarlo. Así, un applet puede ser cargado varias veces en la misma página tomando un nombre simbólico distinto en cada momento.

ALIGN

Se emplea para alinear el applet permitiendo al texto fluir a su alrededor. Puede tomas los siguientes valores: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM y ABSBOTTOM.

VSPACE

Indica el espaciado vertical entre el applet y el texto, en pixels. Sólo funciona cuando se ha indicado ALIGN = LEFT o RIGHT.

HSPACE

Funciona igual que el anterior pero indicando espaciamiento horizontal, en pixels. Sólo funciona cuando se ha indicado ALIGN = LEFT o RIGHT.

Es probable encontrar en algunas distribuciones otras etiquetas para la inclusión de applets, como <APP>. Esto se debe a que estamos ante la tercera revisión de la extensión de HTML para la incrustación de applets y ha sido adoptada como la definitiva. Por ello, cualquier otro medio corresponde a implementaciones obsoletas que han quedado descartadas.

Paso de parámetros a Applets

El espacio que queda entre las marcas de apertura y cierre de la definición de un applet, se utiliza para el paso de parámetros al applet. Para ello se utiliza la marca PARAM en la página html para indicar los parámetros y el método getParameter() de la clase

java.applet.Applet para leerlos en el código interno del applet. La construcción puede repetirse cuantas veces se quiera, una tras otra.

Los atributos que acompañan a la marca PARAM son los siguientes:

NAME

Nombre del parámetro que se desea pasar al applet.

VALUE

Valor que se desea transmitir en el parámetro que se ha indicado antes.

Texto HTML

Texto HTML que será interpretado por los navegadores que no entienden la marca APPLET en sustitución del applet mismo.

Para mostar esta posibilidad se recurre de nuevo al conocido applet básico HolaMundo, para modificarlo y que pueda saludar a cualquiera. Lo que se hará será pasarle al applet el nombre de la persona a quien se desea saludar. Se genera el código para ello y se guarda en el fichero .

```
import java.awt.Graphics;
import java.applet.Applet;
   public class HolaTal extends Applet {
      String nombre;

      public void init() {
      nombre = getParameter( "Nombre" );
      }

   public void paint( Graphics g ) {
      g.drawString( "Hola "+nombre+"!",25,25 );
      }
   }
```

Al compilar el ejemplo se obtiene el fichero HolaTal.class que se incluirá en la siguiente página Web, que se genera a partir del fichero HolaTal.html, en el que se incluye el applet, y que debería tener el siguiente contenido:

```
<HTML>
<APPLET CODE=HolaTal.class WIDTH=300 HEIGHT=100>
<PARAM NAME="Nombre" VALUE="Agustin">
</APPLET>
</HTML>
```

Los parámetros no se limitan a uno solo. Se puede pasar al applet cualquier número de parámetros y siempre hay que indicar un nombre y un valor para cada uno de ellos.

El método **getParameter**() es fácil de entender. El único argumento que necesita es el nombre del parámetro cuyo valor se quiere recuperar. Todos los parámetros se pasan como Strings, en caso de necesitar pasarle al applet un valor entero, se ha de pasar como

String, recuperarlo como tal y luego convertirlo al tipo que se desee. Tanto el argumento de NAME como el de VALUE deben ir colocados entre dobles comillas (") ya que son String.

El hecho de que las marcas <APPLET> y <PARAM> sean ignoradas por los navegadores que no entienden Java, es inteligentemente aprovechado a la hora de definir un contenido alternativo a ser mostrado en este último caso. Así la etiqueta es doble:

```
<APPLET atributos>
parámetros
contenido alternativo
</APPLET>
```

El fichero anterior para mostrar el applet de ejemplo se puede modificar para que pueda ser visualizado en cualquier navegador y en unos casos presente la información alternativa y en otros, ejecute el applet:

```
<HTML>
<APPLET CODE=HolaTal.class WIDTH=300 HEIGHT=100>
<PARAM NAME="Nombre" VALUE="Agustin">
No verás lo bueno hasta que consigas un navegador
<I>Java Compatible</I>
</APPLET>
</HTML>
```

Tokens en parámetros de llamada

Ya de forma un poco más avanzada, a continuación se verá cómo también se pueden pasar varios parámetros en la llamada utilizando separadores, o lo que es lo mismo, separando mediante delimitadores los parámetros, es decir, tokenizando la cadena que contiene el valor del parámetro, por ejemplo:

```
<PARAM NAME=Nombre VALUE="Agustin|Antonio">
```

En este caso el separador es la barra vertical "|", que delimita los dos tokens, pero también se puede redefinir y utilizar cualquier otro símbolo como separador:

```
<PARAM NAME=Separador VALUE="#">
<PARAM NAME=Nombre VALUE="Agustin#Antonio">
```

Si ahora se intenta cambiar de color de fondo en que aparecen los textos en el applet, utilizando el mismo método, se podría tener:

```
<PARAM NAME=Nombre VALUE="Agustin|Antonio">
<PARAM NAME=Color VALUE="verde|rojo">
```

Es más, también se podría conseguir que parpadeasen los mensajes en diferentes colores, cambiando el color de fondo y el del texto:

```
<PARAM NAME=Nombre1 VALUE="Agustin|verde|amarillo">
<PARAM NAME=Nombre2 VALUE="Antonio|rojo|blanco">
```

Para recoger los parámetros pasados en este último caso, bastaría con hacer un pequeño bucle de lectura de los parámetros que interesan:

```
for( int i=1; ; i++ )
    p = getParameter( "Nombre"+i );
if( p == null )
    break;
. . .
}
```

incluso se podría utilizar un fichero para pasar parámetros al applet. La llamada sería del mismo tipo:

```
<PARAM NAME=Fichero VALUE="FicheroDatos">
```

y el FicheroDatos debería tener un contenido, en este caso, que sería el siguiente:

```
Agustin
fondoColor=verde
textoColor=amarillo
fuente=Courier
fuenteTam=14
Antonio
fondoColor=rojo
textocolor=blanco
```

E incluso ese FicheroDatos, se podría conseguir cargarlo independientemente de la URL en que se encontrase, de forma que utilizando el método **getContent**() se podría recuperar el contenido del fichero que contiene los parámetros de funcionamiento del applet:

```
String getContent( String url ) {
URL url = new URL( null,url );
return( (String).url.getContent() );
}
```

Para recuperar los parámetros que están incluidos en la cadena que contiene el valor es posible utilizar dos métodos:

```
StringTokenizer( string,delimitadores )
StreamTokenizer( streamentrada )
```

Así en la cadena Agustin|Antonio si se utiliza el método:

```
StringTokenizer( cadena,"|" );
```

se obtiene el token Agustin, el delimitador "|" y el token Antonio. El código del método sería el que se muestra a continuación:

```
// Capturamos el parámetro
p = getParameter( "p" );

// Creamos el objeto StringTokenizer
st = new StringTokenizer( p,"|" );

// Creamos el almacenamiento
cads = new String[ st.countTokens() ];

// Separamos los tokens de la cadena del parámetro
for( i=0; i < cads.length; i++ )
    cadenas[i] = st.nextToken();
```

En el caso de que se utilice un fichero como verdadera entrada de parámetros al applet y el fichero se encuentre en una dirección URL, se utilizaría el método StreamTokenizer() para obtener los tokens que estén contenidos en ese fichero:

```
// Creamos el objeto URL para acceder a él
url = new URL( "http://www.prueba.es/Fichero" );

// Creamos el canal de entrada
ce = url.openStream();

// Creamos el objeto StreamTokenizer
st = new StreamTokenizer( ce );

// Capturamos los tokens
st.nextToken();
```

El parámetro Archive

El parámetro ARCHIVE ha sido una de las mejores aportaciones de Sun a la nueva versión del JDK, así que a continuación se profundiza un poco más en el porqué de su incorporación y en su funcionamiento.

Una de las cosas que se achacan a Java es la rapidez. El factor principal en la percepción que tiene el usuario de la velocidad y valor de los applets es el tiempo que tardan en cargarse todas las clases que componen el applet. Algunas veces hay que estar esperando más de un minuto para ver una triste animación, ni siquiera buena. Y, desafortunadamente, esta percepción de utilidad negativa puede recaer también sobre applets que realmente sí son útiles.

Para entender el porqué de la necesidad de un nuevo método de carga para acelerarla, necesitamos comprender porqué el método actual es lento. Normalmente un applet se compone de varias clases, es decir, varios ficheros .class. Por cada uno de estos ficheros .class, el cargador de clases debe abrir una conexión individual entre el navegador y el servidor donde reside el applet. Así, si un applet se compone de 20 ficheros .class, el navegador necesitará abrir 20 sockets para transmitir cada uno de los ficheros. La sobrecarga que representa cada una de estas conexiones es relativamente significante. Por ejemplo, cada conexión necesita un número de paquetes adicionales que incrementan el tráfico en la Red.

Me imagino que ya el lector habrá pensado la solución al problema: poner todos los ficheros en uno solo, con lo cual solamente sería necesaria una conexión para descargar todo el código del applet. Bien pensado. Esto es lo mismo que han pensado en un principio los dos grandes competidores en el terreno de los navegadores, Netscape y Microsoft, y que posteriormente Sun ha recogido ya oficialmente.

Desafortunadamente, las soluciones que han implementado ambas compañías no son directamente compatibles. Microsoft, en su afán de marcar diferencia, crea su propio formato de ficheros CAB. La solución de Netscape es utilizar el archiconocido formato ZIP. Y JavaSoft ha definido un nuevo formato de ficheros, que incorpora desde el JDK 1.1, para incluir juntos todos los ficheros de imágenes, sonido y class, que ha llamado formato JAR (Java Archive). Por suerte, es fácil escribir código html de forma que maneje cualquiera de los formatos, en caso necesario. Esto es así porque se puede especificar cada uno de estos formatos de ficheros especiales en extensiones separadas de la marca <APPLET>.

No es de interés el contar la creación de ficheros CAB; quien esté interesado puede consultar la documentación de Java que proporciona Microsoft con su SDK para Java, que es bastante exhaustiva al respecto. Una vez que se dispone de este fichero, se puede añadir un parámetro CABBASE a la marca <APPLET>:

```
<APPLET NAME="Hola" CODE="HolaMundo" WIDTH=50 HEIGHT=50 >
<PARAM NAME=CODEBASE VALUE="http://www.ejemplo.es/classes">
<PARAM NAME=CABBASE VALUE="hola.cab">
</APPLET>
```

El VALUE del parámetro CABBASE es el nombre del fichero CAB que contiene los ficheros .class que componen el conjunto de applet.

Crear un archivo ZIP para utilizarlo con Netscape es muy fácil. Se deben agrupar todos los ficheros .class necesarios en un solo fichero .zip. Lo único a tener en cuenta es que solamente hay que almacenar los ficheros .class en el archivo; es decir, no hay que comprimir.

Si se está utilizando pkzip, se haría:

```
pkzip -e0 archivo.zip listaFicherosClass
```

El parámetro que se indica en la línea de comandos es el número cero, no la "O" mayúscula.

Para utilizar un fichero .zip hay que indicarlo en la marca ARCHIVE de la sección <APPLET>:

```
<APPLET NAME="Hola" CODE="HolaMundo" WIDTH=50 HEIGHT=50
CODEBASE VALUE="http://www.ejemplo.es/classes"
ARCHIVE="hola.zip">
</APPLET>
```

Pero hay más. Podemos crear ambos tipos de ficheros y hacer que tanto los usuarios de Netscape Navigator como los de Microsoft Internet Explorer puedan realizar descargas rápidas del código del applet. No hay que tener en cuenta los usuarios de otros navegadores, o de versiones antiguas de estos dos navegadores, porque ellos todavía podrán seguir cargando los ficheros a través del método lento habitual. Para compatibilizarlo todo, ponemos las piezas anteriores juntas:

```
<APPLET NAME="Hola" CODE="HolaMundo" WIDTH=50 HEIGHT=50
CODEBASE VALUE="http://www.ejemplo.es/classes"
ARCHIVE="hola.zip">
<PARAM NAME=CABBASE VALUE="hola.cab">
<PARAM NAME=CODEBASE VALUE="http://www.ejemplo.es/classes">
<PARAM NAME=CABBASE VALUE="hola.cab">
</APPLET>
```

Ahora que se puede hacer esto con ficheros .cab y .zip, es tarea del lector el trabajo de incorporar los ficheros .jar y poner los tres formatos juntos bajo el mismo paraguas de la marca <APPLET>.

Un Applet básico en Java

A continuación se trata de crear el código fuente de un applet que sea sencillo y sirva para mostrar el ciclo de desarrollo de escritura del código fuente Java, compilación, ejecución y visualización de resultados, es decir, un applet que presente un mensaje de saludo en la pantalla . Recuérdese que Java utiliza la extensión *.java* para designar los ficheros fuente. La programación de applets Java difiere significativamente de la aplicación de aplicaciones Java. Se puede prever que cuando haya necesidades de un interface gráfico, la programación de applets será bastante más sencilla que la programación de aplicaciones que realicen los mismos cometidos, aunque los applets necesiten de un visualizador de ficheros html con soporte Java, para poder ejecutarse. HolaMundo A continuación está el código fuente del applet *HolaMundo*, que es la versión applet de la mínima aplicación Java que antes se ha presentado y desarrollado como aplicación Java .

```
import java.awt.Graphics;
import java.applet.Applet;

public class HolaMundo extends Applet {
    public void paint( Graphics g ) {
```

```
          // Pinta el mensaje en la posición indicada
        g.drawString( "Hola Mundo!",25,25 );
        }
    }
```

Componentes básicos de un Applet

El lenguaje Java implementa un modelo de Programación Orientada a Objetos. Los objetos sirven de bloques centrales de construcción de los programas Java. De la misma forma que otros lenguajes de programación, Java tiene variables de estado y métodos. La descomposición de un applet en sus piezas/objetos, sería la que se muestra a continuación:

```
/*Sección de importaciones
*/

public class NombreDelNuevoApplet extends Applet {

/*
Aquí se declaran las variables de estado (public y private)
*/
    /*
    Los métodos para la interacción con los objetos se
    declaran y definen aquí
    */
    public void MetodoUno( parámetros ) {
        /*
        Aquí viene para cada método, el código Java que
        desempeña la tarea.
        Qué código se use depende del applet
        */
        }
    }
```

Para **HolaMundo**, se importan las dos clases que necesita. No hay variables de estado, y sólo se tiene que definir un método para que el applet tenga el comportamiento esperado.

Clases Incluidas

El comando **`import`** carga otras clases dentro del código fuente. El importar una clase desde un paquete de Java hace que esa clase importada esté disponible para todo el código incluido en el fichero fuente Java que la importa. Por ejemplo, en el applet HolaMundo que se está presentando aquí, se importa la clase **java.awt.Graphics**, y se podrá llamar a los métodos de esta clase desde cualquiera de los métodos que se encuentren incluidos en el fichero HolaMundo.java . Esta clase define una área gráfica y métodos para poder dibujar dentro de ella. La función, método, *paint()* declara a `g` como un objeto de tipo *Graphics*; luego, *paint()* usa el método *drawString()* de la clase **Graphics** para generar su salida.

La Clase Applet

Se puede crear una nueva clase, en este caso **HolaMundo**, extendiendo la clase básica de Java: **Applet**. De esta forma, se hereda todo lo necesario para crear un applet. Modificando determinados métodos del applet, se puede lograr que lleve a cabo las funciones que se desee.

```
import java.applet.Applet;

. . .

public class HolaMundo extends Applet {
```

Métodos de Applet

La parte del applet a modificar es el método *paint()*. En la clase **Applet**, se llama al método *paint()* cada vez que el método arranca o necesita ser refrescado, pero no hace nada. En este caso, del applet básico que se está gestando, lo que se hace es:

```
public void paint( Graphics g )

{ g.drawString( "Hola Mundo!",25,25 ); }
```

De acuerdo a las normas de sobrecarga, se ejecutará este último *paint()* y no el *paint()* vacío de la clase **Applet**. Luego, aquí se ejecuta el método *drawString()*, que le dice al applet cómo debe aparecer un texto en el área de dibujo.

Otros métodos básicos para dibujar son:

```
drawLine( int x1,int y1,int x2,int y2 )

drawRect( int x,int y,int ancho,int alto )

drawOval( int x,int y,int ancho,int alto )
```

Tanto para *drawRect()* como para *drawOval()*, las coordenadas (`x,y`) son la esquina superior izquierda del rectángulo (para *drawOval()*, el óvalo es encajado en el rectángulo que lo circunscribe). Compilación de un Applet Ahora que ya está el código del applet básico escrito y el fichero fuente Java que lo contiene guardado en disco, es necesario compilarlo y obtener un fichero *.class* ejecutable. Se utiliza el compilador Java, *javac* , para realizar la tarea. El comando de compilación será:

```
%javac HolaMundo.java
```

Eso es todo. El compilador *javac* generará un fichero *HolaMundo.class* que podrá ser llamado desde cualquier navegador con soporte Java y, por tanto, capaz de ejecutar applets Java. Llamada a Applets ¿Qué tienen de especial HotJava, Microsoft Explorer o Netscape con respecto a otros navegadores? Con ellos se puede ver código escrito en lenguaje *html* básico y acceder a todo el texto, gráfico, sonido e hipertexto que se pueda ver con cualquier otro navegador. Pero además, y esto es lo que tienen de especial, pueden ejecutar applets, que no es html estándar. Ambos navegadores entienden código html que lleve la marca

```
<APPLET>: <APPLET CODE="SuCodigo.class" WIDTH=100 HEIGHT=50>  </APPLET>
```

Esta marca html llama al applet *SuCodigo.class* y establece su ancho y alto inicial. Cuando se acceda a la página Web donde se encuentre incluida la marca, se ejecutará el byte-code contenido en *SuCodigo.class*, obteniéndose el resultado de la ejecución del applet en la ventana del navegador, con soporte Java, que se esté utilizando. Si no se dispone de ningún navegador, se puede utilizar el visor de applets que proporciona Sun con el JDK, el *appletviewer*, que además requiere muchos menos recursos de la máquina en que se esté ejecutando, que cualquier otro de los navegadores que se acaban de citar.

Applets varios

Depuración General

Compilar y ejecutar el sencillo programa HolaMundo.java a través del fichero HolaMundo.html no debería suponer ningún problema, pero alguna vez se presentarán programas más difíciles y se necesitará el truco de depuración al que todo programador recurre durante el desarrollo de programas en cualquier lenguaje.

```
System.out.printl()
```

Una de las herramientas de depuración más efectivas en cualquier lenguaje de programación es simplemente la salida de información por pantalla. El comando System.out.println imprime la cadena que se le especifique en la ventana de texto en la que se invocó al navegador. La forma de usarlo se muestra a continuación:

```
public void paint( Graphics g ) {
    g.drawString( "Hola Mundo!",25,25 );
    System.out.println( "Estamos en paint()" );
    }
```

Ciclo de Vida de un Applet

Para seguir el ciclo de vida de un applet, se supone que se está ejecutando en el navegador el applet básico HolaMundo, a través de la página HTML que lo carga y corre, y que se ha visto en ejemplos anteriores.

Lo primero que aparece son los mensajes "initializing... starting...", como resultado de la carga del applet en el navegador. Una vez cargado, lo que sucede es:

- Se crea una instancia de la clase que controla al applet
- El applet se inicializa a si mismo
- Comienza la ejecución del applet

Cuando se abandona la página, por ejemplo, para ir a la siguiente, el applet detiene la ejecución. Cuando se regresa a la página que contiene el applet, se reanuda la ejecución.

Si se utiliza la opción del navegador de Reload, es decir, volver a cargar la página, el applet es descargado y vuelto a cargar. El applet libera todos los recursos que hubiese acaparado, detiene su ejecución y ejecuta su finalizador para realizar un proceso de limpieza final de sus trazas. Después de esto, el applet se descarga de la memoria y vuelve a cargarse volviendo a comenzar su inicialización.

Finalmente, cuando se concluye la ejecución del navegador, o de la aplicación que está visualizando el applet, se detiene la ejecución del applet y se libera toda la memoria y recursos ocupados por el applet antes de salir del navegador.

Protección de Applets

Como curiosidad, más que como algo verdaderamente útil, se verá a continuación un método para proteger applets de forma muy sencilla, o por lo menos evitar que nadie pueda ocultar en sus páginas HTML quien es el autor legal de un applet.

El método es muy sencillo y se basa en la utilización de un parámetro del cual se comprobar su existencia, por ejemplo:

```
<PARAM NAME=copyright
VALUE="Applet de Prueba, A.Froufe (C)1996, Todos los derechos reservados">
```

y en el código Java del applet, se comprobaría que efectivamente el parámetro copyright existe y ese es su contenido:

```
if( !getParameter( "copyright" ).equals( "..." )
throw( new Exception( "Violacion del Copyright" ) );
```

donde "..." es el texto completo del valor del parámetro. Pero también se puede hacer de forma más elegante:

```
copyright = getParameter( "copyright" );
// System.out.println( copyright.hashCode() );

if( copyright != -913936799 )
throw( new Exception( "Violacion del Copyright" ) );
```

en donde la sentencia comentada proporciona el valor del copyright para poder introducirlo en la comparación de la presencia o no del parámetro en la llamada al applet. Habría que declarar y definir correctamente tipos y variables, pero la idea básica es la que está expuesta.

Applets de Java (Nociones)

Para escribir applets Java, hay que utilizar una serie de métodos, algunos de los cuales ya se hay sumariado al hablar de los métodos del appletviewer, que es el visualizador de applets de Sun. Incluso para el applet más sencillo se necesitarán varios métodos. Son los que se usan para arrancar (*start*) y detener (*stop*) la ejecución del applet, para pintar (*paint*) y actualizar (*update*) la pantalla y para capturar la información que se pasa al applet desde el fichero *html* a través de la marca `APPLET`.

Los applets no necesitan un método *main()* como las aplicaciones Java, sino que deben implementar (redefinir) al menos uno de los tres métodos siguientes: *init()*, *start()* o *paint()*.

init()

Esta función miembro es llamada al crearse el applet. Es llamada sólo una vez. La clase **Applet** no hace nada en *init()*. Las clases derivadas deben sobrecargar este método para cambiar el tamaño durante su inicialización, y cualquier otra inicialización de los datos que solamente deba realizarse una vez. Deberían realizarse al menos las siguientes acciones:

- Carga de imágenes y sonido
- El redimensionado del applet para que tenga su tamaño correcto

- Asignación de valores a las variables globales

Por ejemplo:

```
public void init() {
   if( width < 200 || height < 200 )
       resize( 200,200 );
   valor_global1 = 0;
   valor_global2 = 100;

   // cargaremos imágenes en memoria sin mostrarlas
   // cargaremos música de fondo en memoria sin reproducirla
   }
```

destroy()

Esta función miembro es llamada cuando el applet no se va a usar más. La clase **Applet** no hace nada en este método. Las clases derivadas deberían sobrecargarlo para hacer una limpieza final. Los applet multithread deberán usar *destroy()* para "*matar*" cualquier thread del applet que quedase activo, antes de concluir definitivamente la ejecución del applet.

start()

Llamada para activar el applet. Esta función miembro es llamada cuando se visita el applet. La clase **Applet** no hace nada en este método. Las clases derivadas deberían sobrecargarlo para comenzar una animación, sonido, etc.

```
public void start() {
   estaDetenido = false;

   // comenzar la reproducción de la música
   musicClip.play();
   }
```

También se puede utilizar *start()* para eliminar cualquier thread que se necesite.

stop()

Llamada para detener el applet. Se llama cuando el applet desaparece de la pantalla. La clase **Applet** no hace nada en este método. Las clases derivadas deberían sobrecargarlo para detener la animación, el sonido, etc.

```
public void stop() {
   estaDetenido = true;

   if( /* ¿se está reproduciendo música? */ )
       musicClip.strop();
   }
```

resize()

El método *init()* debería llamar a esta función miembro para establecer el tamaño del applet. Puede utilizar las variables ancho y alto, pero no es necesario. Cambiar el tamaño en otro sitio que no sea *init()* produce un reformateo de todo el documento y no se recomienda.

En el navegador Netscape, el tamaño del applet es el que se indica en la marca `APPLET` del html, no hace caso a lo que se indique desde el código Java del applet.

width

Variable entera, su valor es el ancho definido en el parámetro WIDTH de la marca *html* del APPLET. Por defecto es el ancho del icono.

height

Variable entera, su valor es la altura definida en el parámetro HEIGHT de la marca *html* del APPLET. Por defecto es la altura del icono. Tanto **width** como **height** están siempre disponibles para que se puede chequear el tamaño del applet.

Se puede retomar el ejemplo de *init()*:

```
public void init() {
   if( width < 200 || height < 200 )
       resize( 200,200 );
   ...
```

paint()

Este método se llama cada vez que se necesita refrescar el área de dibujo del applet. *paint()* es un método de la clase **Component**, que es heredado por varias clases intermedias y, finalmente, es heredado por la clase **Applet**. La clase **Applet** simplemente dibuja una caja con sombreado de tres dimensiones en el área. Obviamente, la clase derivada debería sobrecargar este método para representar algo inteligente en la pantalla.

Para repintar toda la pantalla cuando llega un evento *Paint*, se pide el rectángulo sobre el que se va a aplicar *paint()* y si es más pequeño que el tamaño real del applet se invoca a *repaint()*, que como va a hacer un *update()*, se actualizará toda la pantalla.

Se puede utilizar *paint()* para imprimir el mensaje de bienvenida:

```
void public paint( Graphics g ) {
   g.drawString( "Hola Java!",25,25 );
   // Dibujaremos la imágenes que necesitemos
   }
```

update()

Esta es la función que se llama realmente cuando se necesita actualizar la pantalla. La clase **Applet** simplemente limpia el área y llama al método *paint()*. Esta funcionalidad es suficiente en la mayoría de los casos. De cualquier forma, las clases derivadas pueden sustituir esta funcionalidad para sus propósitos.

Se puede, por ejemplo, utilizar *update()* para modificar selectivamente partes del área gráfica sin tener que pintar el área completa:

```
public void update( Graphics g ) {
   if( estaActualizado ) {
       g.clear(); // garantiza la pantalla limpia
       repaint(); // podemos usar el método
       // padre: super.update()
       }
   else
       // Información adicional
   g.drawString( "Otra información",25,50 );
   }
```

repaint()

A esta función se la debería llamar cuando el applet necesite ser repintado. No debería sobrecargarse, sino dejar que Java repinte completamente el contenido del applet.

Al llamar a *repaint()* sin parámetros, internamente se llama a *update()* que borrará el rectángulo sobre el que se redibujará y luego se llama a *paint()*. Como a *repaint()* se le pueden pasar parámetros, se puede modificar el rectángulo a repintar.

getParameter()

Este método carga los valores parados al applet vía la marca `APPLET` de *html*. El argumento *String* es el nombre del parámetro que se quiere obtener. Devuelve el valor que se le haya asignado al parámetro; en caso de que no se le haya asignado ninguno, devolverá *null*.

Para usar *getParameter()*, se define una cadena genérica. Una vez que se ha capturado el parámetro, se utilizan métodos de cadena o de números para convertir el valor obtenido al tipo adecuado.

```
public void init() {
   String pv;

   pv = getParameter( "velocidad" );
   if( pv == null )
       velocidad = 10;
   else
       velocidad = Integer.parseInt( pv );
   }
```

getDocumentBase()

Indica la ruta *http*, o el directorio del disco, de donde se ha recogido la página *html* que contiene el applet, es decir, el lugar donde está la hoja en todo Internet o en el disco.

print()

Para imprimir en impresora, al igual que *paint()* se puede utilizar *print()*, que pintará en la impresora el mapa de bits del dibujo. No obstante, la clase **PrintJob** es la que se ha de utilizar para estos menesteres, tal como se verá en otra sección del Tutorial.

AWT

AWT es el acrónimo del X Window Toolkit para Java, donde X puede ser cualquier cosa: Abstract, Alternative, Awkward, Another o Asqueroso; aunque parece que Sun se decanta por Abstracto, seriedad por encima de todo. Se trata de una biblioteca de clases Java para el desarrollo de Interfaces de Usuario Gráficas. La versión del AWT que Sun proporciona con el JDK se desarrolló en sólo dos meses y es la parte más débil de todo lo que representa Java como lenguaje. El entorno que ofrece es demasiado simple, no se han tenido en cuenta las ideas de entornos gráficos novedosos. Quizá la presión de tener que lanzar algo al mercado haya tenido mucho que ver en la pobreza de AWT.

Javasoft, en vista de la precariedad de que hace gala el AWT, y para asegurarse que los elementos que desarrolla para generar interfaces gráficas sean fácilmente transportables entre plataformas, se ha unido con Netscape, IBM y Lighthouse Design para crear un conjunto de clases que proporcionen una sensación visual agradable y sean más fáciles de utilizar por el programador. Esta colección de clases son las Java Foundation Classes (JFC), que están constituidas por cinco grupos de clases, al menos en este momento: AWT, Java 2D, Accesibilidad, Arrastrar y Soltar y Swing.

AWT, engloba a todos los componentes del AWT que existían en la versión 1.1.2 del JDK y en los que se han incorporado en versiones posteriores:

- Java 2D es un conjunto de clases gráficas bajo licencia de IBM/Taligent, que todavía está en construcción
- Accesibilidad, proporciona clases para facilitar el uso de ordenadores y tecnología informática a disminuidos, como lupas de pantalla, y cosas así
- Arrastrar y Soltar (Drag and Drop), son clases en las que se soporta Glasgow, que es la nueva generación de los JavaBeans
- Swing, es la parte más importante y la que más desarrollada se encuentra. Ha sido creada en conjunción con Netscape y proporciona una serie de componentes muy bien descritos y especificados de forma que su presentación visual es independiente de la plataforma en que se ejecute el applet o la aplicación que utilice estas clases. Swing simplemente extiende el AWT añadiendo un conjunto de componentes, JComponents, y sus clases de soporte. Hay un conjunto de componentes de Swing que son análogos a los de AWT, y algunos de ellos participan de la arquitectura MVC (Modelo-Vista-Controlador), aunque Swing también proporciona otros widgets nuevos como árboles, pestañas, etc.

La estructura básica del AWT se basa en Componentes y Contenedores. Estos últimos contienen Componentes posicionados a su respecto y son Componentes a su vez, de forma que los eventos pueden tratarse tanto en Contenedores como en Componentes, corriendo por cuenta del programador (todavía no hay herramientas de composición visual) el encaje de todas las piezas, así como la seguridad de tratamiento de los eventos adecuados. Con Swing se va un paso más allá, ya que todos los JComponentes son

subclases de Container, lo que hace posible que widgets Swing puedan contener otros componentes, tanto de AWT como de Swing, lo que hace prever interesantes posibilidades.

A continuación se aborda la programación con el AWT fundamentalmente para tener la base suficiente y poder seguir profundizando en las demás características del lenguaje Java, aunque también se presentarán ejemplos de utilización de JComponentes cuando Swing tenga implementado alguno que corresponda el del AWT o derivado de él.

Interfaz de Usuario

El interfaz de usuario es la parte del programa que permite a éste interactuar con el usuario. Las interfaces de usuario pueden adoptar muchas formas, que van desde la simple línea de comandos hasta las interfaces gráficas que proporcionan las aplicaciones más modernas.

El interfaz de usuario es el aspecto más importante de cualquier aplicación. Una aplicación sin un interfaz fácil, impide que los usuarios saquen el máximo rendimiento del programa. Java proporciona los elementos básicos para construir decentes interfaces de usuario a través del AWT, y opciones para mejorarlas mediante Swing, que sí permite la creación de interfaces de usuario de gran impacto y sin demasiados quebraderos de cabeza por parte del programador.

Al nivel más bajo, el sistema operativo transmite información desde el ratón y el teclado como dispositivos de entrada al programa. El AWT fue diseñado pensando en que el programador no tuviese que preocuparse de detalles como controlar el movimiento del ratón o leer el teclado, ni tampoco atender a detalles como la escritura en pantalla. El AWT constituye una librería de clases orientada a objeto para cubrir estos recursos y servicios de bajo nivel.

Debido a que el lenguaje de programación Java es independiente de la plataforma en que se ejecuten sus aplicaciones, el AWT también es independiente de la plataforma en que se ejecute. El AWT proporciona un conjunto de herramientas para la construcción de interfaces gráficas que tienen una apariencia y se comportan de forma semejante en todas las plataformas en que se ejecute. Los elementos de interfaz proporcionados por el AWT están implementados utilizando toolkits nativos de las plataformas, preservando una apariencia semejante a todas las aplicaciones que se creen para esa plataforma. Este es un punto fuerte del AWT, pero también tiene la desventaja de que un interfaz gráfico diseñado para una plataforma, puede no visualizarse correctamente en otra diferente. Estas carencias del AWT son subsanadas en parte por Swing, y en general por las JFC.

Estructura del AWT

La estructura de la versión actual del AWT se puede resumir en los puntos que se exponen a continuación:

- Los Contenedores contienen Componentes, que son los controles básicos
- No se usan posiciones fijas de los Componentes, sino que están situados a través de una disposición controlada (layouts)

- El común denominador de más bajo nivel se acerca al teclado, ratón y manejo de eventos
- Alto nivel de abstracción respecto al entorno de ventanas en que se ejecute la aplicación (no hay áreas cliente, ni llamadas a X, ni hWnds, etc.)
- La arquitectura de la aplicación es dependiente del entorno de ventanas, en vez de tener un tamaño fijo
- Es bastante dependiente de la máquina en que se ejecuta la aplicación (no puede asumir que un diálogo tendrá el mismo tamaño en cada máquina)
- Carece de un formato de recursos. No se puede separar el código de lo que es propiamente interface. No hay ningún diseñador de interfaces (todavía)

Componentes y Contenedores

Un interfaz gráfico está construida en base a elementos gráficos básicos, los Componentes. Típicos ejemplos de estos Componentes son los botones, barras de desplazamiento, etiquetas, listas, cajas de selección o campos de texto. Los Componentes permiten al usuario interactuar con la aplicación y proporcionar información desde el programa al usuario sobre el estado del programa. En el AWT, todos los Componentes de la interface de usuario son instancias de la clase Component o uno de sus subtipos.

Los Componentes no se encuentran aislados, sino agrupados dentro de Contenedores. Los Contenedores contienen y organizan la situación de los Componentes; además, los Contenedores son en sí mismos Componentes y como tales pueden ser situados dentro de otros Contenedores. También contienen el código necesario para el control de eventos, cambiar la forma del cursor o modificar el icono de la aplicación. En el AWT, todos los Contenedores son instancias de la clase Container o uno de sus subtipos.

Tipos de Componentes

En el árbol siguiente se muestra la relación que existe entre todas las clases que proporciona AWT para la creación de interfaces de usuario, presentando la jerarquía de Clases e Interfaces:

Clases:

- Adjustable
- BorderLayout
- CardLayout
- CheckboxGroup
- Color
- Component
- Button
- Canvas
- Checkbox
- Choice

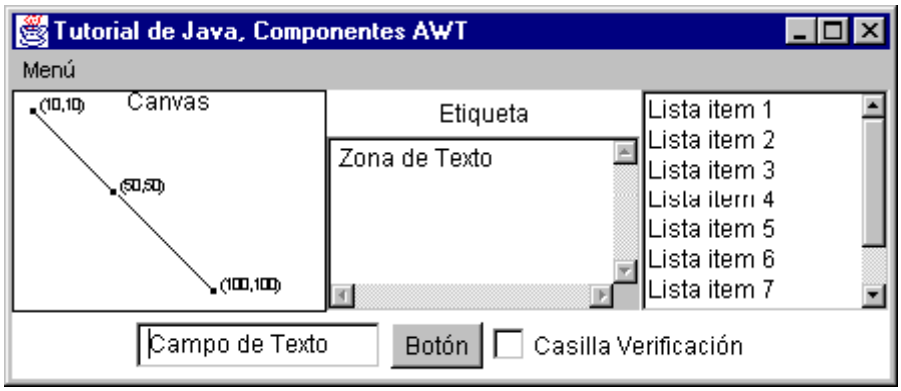
- Container
- Panel
- Applet
- ScrollPane
- Window
- Dialog
- FileDialog
- Frame
- Label
- List
- Scrollbar
- TextComponent
- TextArea
- TextField
- Cursor
- Dimension
- Event
- FlowLayout
- Font
- FontMetrics
- Graphics
- GridLayout
- GridBagConstraints
- GridBagLayout
- Image
- Insets
- MediaTracker
- MenuComponent
- MenuBar
- MenuItem
- CheckboxMenuItem
- Menu
- PopMenu
- MenuShorcut
- Point
- Polygon
- PrintJob

- Rectangle
- Toolkit

Interfaces:

- LayoutManager
- LayoutManager2
- MenuContainer
- Shape

En la figura siguiente se reproduce la ventana generada por el código de la aplicación del ejemplo **java1301** que muestra todos los Componentes que proporciona el AWT. A continuación se verán en detalle estos Componentes, pero aquí se puede ya observar la estética que presentan en su conjunto. La ventana es necesaria porque el programa incluye un menú, y los menús solamente pueden utilizarse en ventanas. El código contiene un método main() para poder ejecutarlo como una aplicación independiente.



AWT - Componentes

Component es una clase abstracta que representa todo lo que tiene una posición, un tamaño, puede ser pintado en pantalla y puede recibir eventos.

No tiene constructores públicos, ni puede ser instanciada. Sin embargo, desde el JDK 1.1 puede ser extendida para proporcionar una nueva característica incorporada a Java, conocida como componentes Lightweight.

Los Objetos derivados de la clase Component que se incluyen en el Abstract Window Toolkit son los que aparecen a continuación:

- Button
- Canvas
- Checkbox
- Choice
- Container
- Panel

- Window
- Dialog
- Frame
- Label
- List
- Scrollbar
- TextComponent
- TextArea
- TextField

Sobre estos Componentes se podrían hacer más agrupaciones y quizá la más significativa fuese la que diferencie a los Componentes según el tipo de entrada. Así habría Componentes con entrada de tipo no-textual como los botones de pulsación (**Button**), las listas (**List**), botones de marcación (**Checkbox**), botones de selección (**Choice**) y botones de comprobación (**CheckboxGroup**); Componentes de entrada y salida textual como los campos de texto (TextField), las áreas de texto (**TextArea**) y las etiquetas (**Label**); y, otros Componentes sin acomodo fijo en ningún lado, en donde se encontrarían Componentes como las barras de desplazamiento (**Scrollbar**), zonas de dibujo (**Canvas**) e incluso los Contenedores (**Panel**, **Window**, **Dialog** y **Frame**), que también pueden considerarse como Componentes.

Botones de Pulsación

Los botones de pulsación (Button), son los que se han utilizado fundamentalmente en los ejemplos de este Tutorial, aunque nunca se han considerado sus atributos específicamente.

La clase **Button** es una clase que produce un componente de tipo botón con un título. El constructor más utilizado es el que permite pasarle como parámetro una cadena, que será la que aparezca como título e identificador del botón en el interfaz de usuario. No dispone de campos o variables de instancia y pone al alcance del programador una serie de métodos entre los que destacan por su utilidad los siguientes:

addActionListener()	Añade un receptor de eventos de tipo Action producidos por el botón
getLabel()	Devuelve la etiqueta o título del botón
removeActionListener()	Elimina el receptor de eventos para que el botón deje de realizar acción alguna
setLabel()	Fija el título o etiqueta visual del botón

Además dispone de todos los métodos heredados de las clases **Component** y **Object**.

ActionListener es uno de los eventos de tipo semántico. Un evento de tipo **Action** se produce cuando el usuario pulsa sobre un objeto **Button**. Además, un objeto **Button** puede generar eventos de bajo nivel de tipo **FocusListener**, **MouseListener** o

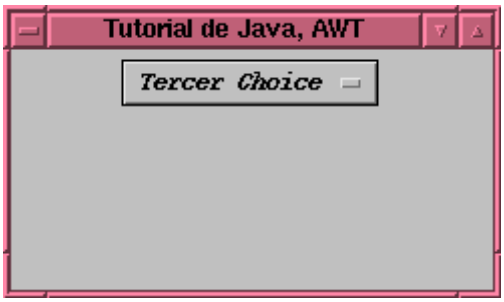
KeyListener, porque hereda los métodos de la clase **Component** que permiten instanciar y registrar receptores de eventos de este tipo sobre objetos de tipo **Button**.

Botones de Selección

Los botones de selección (Choice) permiten el rápido acceso a una lista de elementos, presentándose como título el item que se encuentre seleccionado.

La clase **Choice** extiende la clase **Component** e implementa el interfaz **ItemSelectable**, que es el interfaz que mantiene un conjunto de items en los que puede haber, o no, alguno seleccionado. Además, esta clase proporciona el método **addItemListener**(), que añade un registro de eventos item, que es muy importante a la hora de tratar los eventos que se producen sobre los objetos de tipo **Choice**.

La clase **Choice** también dispone de cerca de una veintena de métodos que permiten la manipulación de los items disponibles para la selección, cuya amplia documentación el lector puede consultar en el API del JDK.



El ejemplo que se ha implementado, java1302, para ilustrar el uso de los botones de selección, coloca un objeto de tipo **Choice** sobre un objeto **Frame** y añade tres objetos de tipo **String** al objeto **Choice**, fijando el segundo de ellos como preseleccionado a la hora de lanzar el programa. La presentación inicial de la aplicación al ejecutarla es la que reproduce la imagen.

Se instancia y registra un objeto de tipo **ItemListener** sobre el objeto **Choice** para identificar y presentar el objeto **String** que se elige cuando el usuario utiliza una selección. Cuando esto ocurre, se captura un evento en el método sobrescrito *itemStateChanged*() del objeto **ItemListener**. El código de este método utiliza una llamada a *getSelectedItem*() para el item que está marcado y presentar la cadena que le corresponde.

También se instancia y registra un objeto receptor de eventos *windowClosing*() sobre el Frame para concluir el programa cuando el usuario cierre la ventana.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class java1302 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }
```

```
// Clase del Interfaz Gráfico que instancia los objetos
class IHM {
    public IHM() {
        // Instancia un objeto Choice y coloca objetos String sobre el
        // para realizar las selecciones
        Choice miChoice = new Choice();
        miChoice.add( "Primer Choice" );
        miChoice.add( "Segundo Choice" );
        miChoice.add( "Tercer Choice" );
       // Seleccionamos la cadena correspondiente a la tercera selección
        // por defecto, al arrancar la aplicación
        miChoice.select( "Tercer Choice" );
        // Instanciamos y registramos un objeto ItemListener sobre
        // el objeto Choice
        miChoice.addItemListener( new MiItemListener( miChoice ) );
        // Colocamos el objetos Choice sobre el Frame para poder verlo
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );
        miFrame.add( miChoice );
        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );
      // Instanciamos y registramos un objeto receptor de los eventos de
        // la ventana, para recoger el evento de cierre del Frame y
        // concluir la ejecucion de la aplicacion al recibirlo
        miFrame.addWindowListener( new Conclusion() );
        }
    }
// Clase para recibir los eventos ItemListener generados por el objeto
// Choice de la aplicación
class MiItemListener implements ItemListener{
    Choice oChoice;
    MiItemListener( Choice choice ) {
        // Guardamos una referencia al objeto Choice
        oChoice = choice;
        }
    // Sobreescribimos el metodo itemStateChanged() del interfaz del
    // ItemListener
    public void itemStateChanged( ItemEvent evt ) {
        System.out.println( oChoice.getSelectedItem() );
        }
    }
// Concluye la ejecución de la aplicación cuando el usuario cierra la
// ventana, porque se genera un evento windowClosing
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
    }
}
```

Los trozos de código que merecen un repaso en el programa anterior son los que se ven a continuación. Empezando por las sentencias siguientes, que son las típicas usadas en la instanciación del objeto **Choice** y la incorporación de objetos **String** a ese objeto **Choice**.

```
Choice miChoice = new Choice();
miChoice.add( "Primer Choice" );
. . .
```

La línea de código siguiente hace que el objeto Tercer **Choice** de tipo **String**, sea el que se encuentre visible al lanzar la aplicación.

```
miChoice.select( "Tercer Choice" );
```

Y la sentencia que se reproduce ahora es la que instancia y registra un objeto de tipo ItemListener sobre el objeto Choice.

```
miChoice.addItemListener( new MiItemListener( miChoice ) );
```

A esta sentencia le sigue el código que crea el objeto **Frame**, coloca el objeto **Choice** en él, etc. Ya se han visto varias veces sentencias de este tipo, así que no merece la pena volver sobre ellas. Así que ya solamente queda como código interesante en el programa el método *itemStateChanged*() que está sobrescrito, del objeto **ItemListener**.

```
public void itemStateChanged( ItemEvent evt ) {
    System.out.println( oChoice.getSelectedItem() );
    }
```

Eventos de este tipo se producen siempre que el usuario realiza una selección (abre la lista de opciones y pulsa el botón del ratón con el cursor sobre una de ellas). Y, como se puede ver, el método *getSelectedItem*() es el encargado de obtener la representación en cadena del item que estaba seleccionado en ese momento.

Botones de Comprobación

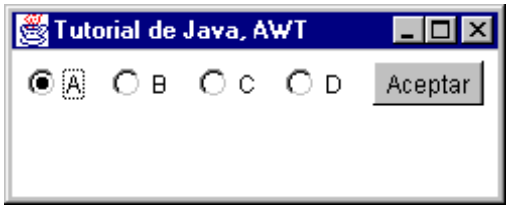
La clase **CheckBox** extiende la clase **Component** e implementa el interfaz **ItemSelectable**, que es el interfaz que contiene un conjunto de items entre los que puede haber o no alguno seleccionado.

Los botones de comprobación (**Checkbox**) se pueden agrupar para formar un interfaz de botón de radio (**CheckboxGroup**), que son agrupaciones de botones de comprobación de exclusión múltiple, es decir, en las que siempre hay un único botón activo.

La programación de objetos **Checkbox** puede ser simple o complicada, dependiendo de lo que se intente conseguir. La forma más simple para procesar objetos **Checkbox** es colocarlos en un **CheckboxGroup**, ignorar todos los eventos que se generen cuando el usuario selecciona botones individualmente y luego, procesar sólo el evento de tipo **Action** cuando el usuario fije su selección y pulse un botón de confirmación. Hay gran cantidad de programas, fundamentalmente para Windows, que están diseñados en base a este funcionamiento.

La otra forma, más compleja, para procesar información de objetos **Checkbox** es responder a los diferentes tipos de eventos que se generan cuando el usuario selecciona objetos **Checkbox** distintos. Actualmente, no es demasiado complicado responder a estos eventos, porque se pueden instanciar objetos **Listener** y registrarlos sobre los objetos **Checkbox**, y eso no es difícil. La parte compleja es la implementación de la lógica necesaria para dar sentido a las acciones del usuario, especialmente cuando ese usuario no tiene clara la selección que va a realizar y cambia continuamente de opinión.

El ejemplo que sigue, java1303, utiliza la solución simple, permitiendo que el usuario cambie de opción cuántas veces quiera y tome una decisión final pulsando sobre el botón "Aceptar". Se colocan cuatro objetos **Checkbox**, definidos sobre un objeto **CheckboxGroup**, y un objeto **Button** sobre un objeto **Frame**. La apariencia de estos elementos en pantalla es la que muestra la figura.



Se instancia y registra un objeto de tipo **ActionListener** sobre el objeto **Button**. La acción de seleccionar y deseleccionar un objeto **Checkbox** es controlada automáticamente por el sistema, al formar parte de un **CheckboxGroup**. Cada vez que se pulse el botón, se generará un evento que al ser procesado por el método sobrescrito *actionPerformed*(), determina y presenta en pantalla la identificación del botón de comprobación que está seleccionado. En la ventana se muestra tanto la identificación asignada por el sistema a los botones, como la etiqueta que es asignada directamente por el programa, demostrando que tanto una como otra se pueden utilizar para identificar el botón seleccionado.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class java1303 {
    public static void main( String args[] ) {
        // Se instancia un objeto Interfaz Hombre-maquina
        IHM ihm = new IHM();
        }
    }
// Clase del Interfaz gráfico
class IHM {
    // Constructor de la clase
    public IHM() {
        // Se crea un objeto CheckboxGroup
        CheckboxGroup miCheckboxGroup = new CheckboxGroup();
        // Ahora se crea un objeto Button y se registra un objeto
        // ActionListener sobre él
        Button miBoton = new Button( "Aceptar" );
   miBoton.addActionListener( new MiActionListener( miCheckboxGroup ) );
      // Se crea un objeto Frame para contener los objetos Checkbox y el
        // objeto Button. Se fija un FlowLayout, se incorporan a el los
        // objetos, se fija el tamaño y se hace visible
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );
        miFrame.add( new Checkbox( "A",true,miCheckboxGroup ) );
        miFrame.add( new Checkbox( "B",false,miCheckboxGroup ) );
        miFrame.add( new Checkbox( "C",false,miCheckboxGroup ) );
        miFrame.add( new Checkbox( "D",false,miCheckboxGroup ) );
        miFrame.add( miBoton );
        miFrame.setSize( 250,100 );
        miFrame.setVisible( true );
        // Instanciamos y registramos un receptor para terminar la
        // ejecución de la aplicación, cuando el usuario cierre la
        // ventana
        miFrame.addWindowListener( new Conclusion() );
        }
    }

// Esta clase indica la caja de selección que esta seleccionada
// cuando se pulsa el botón de Aceptar
class MiActionListener implements ActionListener {
    CheckboxGroup oCheckBoxGroup;
    MiActionListener( CheckboxGroup checkBGroup ) {
```

```
        oCheckBoxGroup = checkBGroup;
        }
      public void actionPerformed( ActionEvent evt ) {
      System.out.println(oCheckBoxGroup.getSelectedCheckbox().getName()+
            " " + oCheckBoxGroup.getSelectedCheckbox().getLabel() );
        }
    }
// Concluye la aplicación cuando el usuario cierra la ventana
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

Hay cuatro trozos interesantes de código en este ejemplo que merecen un poco de atención especial. El primero es la creación del objeto **CheckboxGroup**, que posteriormente se utilizará para agrupar a los cuatro objetos Checkbox en un solo grupo lógico.

```
CheckboxGroup miCheckboxGroup = new CheckboxGroup()
```

Luego se crean el botón "Aceptar" y el objeto ActionListener que se va a registrar sobre él.

```
Button miBoton = new Button( "Aceptar" );
miBoton.addActionListener( new MiActionListener( miCheckboxGroup ) );
```

También está la instanciación de los cuatro objetos Checkbox como parte del grupo CheckboxGroup y su incorporación al Frame.

```
miFrame.add( new Checkbox( "A",true,miCheckboxGroup ) );
```

Y, finalmente, es interesante el fragmento de código en el que se sobrescribe el método *actionPerformed*() en el objeto **ActionListener** que extrae la identificación del objeto **Checkbox**, que se encuentra seleccionado. Aquí, el método *getSelectedCheckbox*() es un método de la clase **CheckboxGroup**, que devuelve un objeto de tipo **Checkbox**. El método *getLabel*() es miembro de la clase **Checkbox**, y el método *getName*() es miembro de la clase **Component**, que es una superclase de **Checkbox**.

```
public void actionPerformed(ActionEvent evt)
{System.out.println(oCheckBoxGroup.getSelectedCheckbox().getName()+
    " " + oCheckBoxGroup.getSelectedCheckbox().getLabel() );     }
```

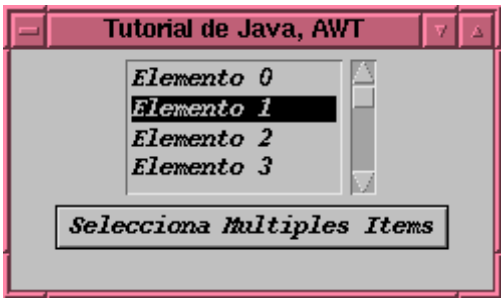
Listas

Las listas (List) aparecen en los interfaces de usuario para facilitar a los operadores la manipulación de muchos elementos. Se crean utilizando métodos similares a los de los botones Choice. La lista es visible todo el tiempo, utilizándose una barra de desplazamiento para visualizar los elementos que no caben en el área de la lista que aparece en la pantalla.

La clase **List** extiende la clase **Component** e implementa el interfaz **ItemSelectable**, que es el interfaz que contiene un conjunto de items en los que puede haber, o no, alguno seleccionado. Además, soporta el método *addActionListener*() que se utiliza para recoger los eventos **ActionEvent** que se produce cuando el usuario pica dos veces con el ratón sobre un elemento de la lista.

En el ejemplo java1304 , que ilustra el empleo de las listas, coloca un objeto **List** y un objeto **Button** sobre un objeto **Frame**. Se añaden quince objetos **String** a la lista y se

deja el segundo como seleccionado inicialmente. La apariencia de la ventana así construida, al inicializar el programa, es la que se reproduce en la imagen siguiente.



Sobre la lista se instancia y registra un objeto de tipo **ActionListener**, cuyo propósito es identificar y presentar en pantalla el objeto **String**, que el usuario ha seleccionado haciendo un doble pique sobre él. Cuando selecciona y luego pica dos veces con el ratón sobre un elemento de la lista, un evento es capturado por el método sobrescrito *actionPerformed*() del objeto **ActionListener**. El código de este método utiliza la llamada a *getSelectedItem*() de la clase **List**, para identificar y presentar la cadena que corresponde al elemento seleccionado. Sin embargo, si el usuario realiza una doble pulsación con el ratón sobre un elemento, mientras otro se encuentra seleccionado, el método *getSelectedItem*() devuelve null y no aparecerá nada en pantalla.

Al objeto **Frame** también se le incorpora un Botón para permitir realizar selección múltiple en la lista; de tal forma que cuando se pulsa, se captura el evento **ActionListener** que genera y presenta en pantalla los elementos que se encuentren seleccionados en ese momento, incluso aunque sólo haya uno de ellos.

También se instancia y registra un objeto receptor de eventos *windowClosing*() sobre el objeto Frame, para concluir el programa cuando el usuario cierre la ventana.

```
import java.awt.*;
import java.awt.event.*;
public class java1304 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM(){
        // Instancia un objeto List y coloca algunas cadenas sobre el,
        // para poder realizar selecciones
        List miLista = new List();
        for( int i=0; i < 15; i++ )
            miLista.add( "Elemento "+i );
        // Activa la seleccion multiple
        miLista.setMultipleMode( true );
        // Presenta el elemento 1 al inicio
        miLista.select( 1 );
        // Instancia y registra un objeto ActionListener sobre el objeto
        // List. Se produce un evento de tipo Action cuando el usuario
        // pulsa dos veces sobre un elemento
      miLista.addActionListener( new MiListaActionListener( miLista ) );
        // Instancia un objeto Button para servicio de la seleccion
```

```
      // multiple. Tambien instancia y registra un objeto ActionListener
         // sobre el boton
         Button miBoton = new Button( "Selecciona Multiples Items" );
       miBoton.addActionListener( new miBotonActionListener( miLista ) );
         // Coloca el objeto List y el objeto Button el el objeto Frame
         Frame miFrame = new Frame( "Tutorial de Java, AWT" );
         miFrame.setLayout( new FlowLayout() );
         miFrame.add( miLista );
         miFrame.add( miBoton );
         miFrame.setSize( 250,150 );
         miFrame.setVisible( true );
         // Instancia y registra un objeto receptor de eventos de ventana
         // para concluir la ejecucion del programa cuando el Frame se
         // cierres por accion del usuario sobre el
         miFrame.addWindowListener( new Conclusion() );
         }
    }

// Clase para recibir eventos de tipo ActionListener sobre el
// objeto List. Presenta en elemento seleccionado cuando el usuario
// pulsa dos veces sobre un item de lista cuando la seleccion es
// individual. Si el usuario pica dos veces sobre una seleccion
// multiple, se produce un evento pero el metodo getSelectedItem()
// de la clase List devuelve null y no se presenta nada en pantalla
class MiListaActionListener implements ActionListener {
    List oLista;
    MiListaActionListener( List lista ) {
        // Salva una referencia al objeto List
        oLista = lista;
        }
        // Sobreescribe el metodo actionPerformed() del interfaz
        // ActionListener
        public void actionPerformed( ActionEvent evt ) {
            if( oLista.getSelectedItem() != null ) {
                System.out.println( "Seleccion Simple de Elementos" );
                System.out.println( "  "+oLista.getSelectedItem() );
                }
            }
        }
// Clase para recoger los eventos Action que se produzcan sobre el
// objeto Button. Presenta los elementos que haya seleccionados
// cuando el usuario lo pulsa, incluso aunque solamente haya uno
// marcado. Si no hubiese ninguno, so se presentaria nada en
// la pantalla
class miBotonActionListener implements ActionListener {
    List oLista;

    miBotonActionListener( List lista ) {
        // Salva una referencia al objeto List
        oLista = lista;
        }
    // Sobreescribe el metodo actionPerformed() del interfaz
    // ActionListener
    public void actionPerformed( ActionEvent evt ) {
        String cadena[] = oLista.getSelectedItems();
        if( cadena.length != 0 ) {
            System.out.println( "Seleccion Multiple de Elementos" );
            for( int i=0; i < cadena.length; i++ )
                System.out.println( "  "+cadena[i] );
                }
            }
        }
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Concluye el programa cuando se cierra la ventana
```

```
        System.exit(0);
        }
    }
```

Si se realiza una revisión del código del ejemplo, se encuentran algunos fragmentos que merecen una reflexión. En el primero de ellos, que se reproduce en las líneas siguientes, se instancia un objeto **List** y se rellenan quince cadenas. La lista se define como una selección múltiple y se fija el segundo elemento como el inicialmente seleccionado en el momento del arranque del programa.

```
// Instancia un objeto List y coloca algunas cadenas sobre el,
// para poder realizar selecciones
List miLista = new List();
for( int i=0; i < 15; i++ )
    miLista.add( "Elemento "+i );
// Activa la seleccion multiple
miLista.setMultipleMode( true );
// Presenta el elemento 1 al inicio
miLista.select( 1 );
```

La sentencia siguiente es la que instancia y registra un objeto de tipo ActionListener sobre la lista, para que responda al doble pique del ratón sobre uno de los elementos de ese lista.

```
miLista.addActionListener( new MiListaActionListener( miLista ) );
```

Luego, ya se encuentra el archivisto código que instancia un objeto Button, que instancia y registra un objeto ActionListener sobre ese botón, y que lo incorpora al Frame.

El siguiente fragmento de código interesante se encuentra en la clase **MiListaActionListener**, que está diseñada para responder cuando el usuario pulse dos veces con el ratón sobre un item seleccionado de la lista. Si el usuario pica dos veces sobre una única selección de la lista, el método sobrescrito *actionPerformed*() identifica el elemento a través de una llamada al método *getSelectedItem*() y lo presenta en pantalla. Sin embargo, si se pulsa dos veces, cuando hay más de un elemento seleccionado, el método *getSelectedItem*() devuelve null y el elemento será ignorado.

```
public void actionPerformed( ActionEvent evt ) {
    if( aLista.getSelectedItem() != null ) {
        System.out.println( "Seleccion Simple de Elementos" );
        System.out.println( "  "+oLista.getSelectedItem() );
        }
    }
```

Las líneas de código que se reproducen a continuación, constituyen el código que responde a un evento **Action** producido sobre el objeto **Button**, y presenta en pantalla uno, o más, elementos seleccionados de la lista. En este caso, el método empleado es *getSelectedItem*(), para crear un array que mantenga todos los elementos que estén seleccionado en el momento de pulsar el botón y que luego se presenta en pantalla.

```
public void actionPerformed( ActionEvent evt ) {
    String cadena[] = oLista.getSelectedItems();
    if( cadena.length != 0 ) {
        System.out.println( "Seleccion Multiple de Elementos" );
        for( int i=0; i < cadena.length; i++ )
            System.out.println( "  "+cadena[i] );
            }
        }
    }
```

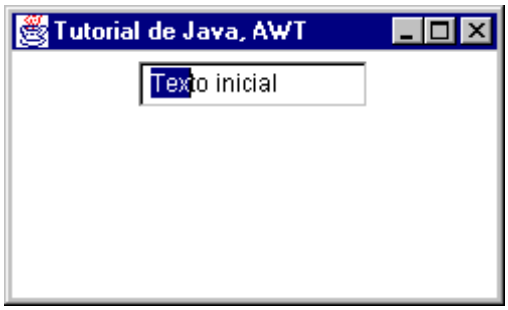
Campos de Texto

Para la entrada directa de datos se suelen utilizar los campos de texto, que aparecen en pantalla como pequeñas cajas que permiten al usuario la entrada por teclado de una línea de caracteres.

Los campos de texto (**TextField**) son los encargados de realizar esta entrada, aunque también se pueden utilizar, activando su indicador de no-editable, para presentar texto e una sola línea con una apariencia en pantalla más llamativa, debido al borde simulando 3-D que acompaña a este tipo de elementos del interfaz gráfico.

La clase **TextField** extiende a la clase **TextComponent**, que extiende a su vez, a la clase **Component**. Por ello, hay una gran cantidad de métodos que están accesibles desde los campos de tesxto. La clase **TextComponent** también es importante en las áreas de texto, en donde se permite la entrada de múltiples líneas de texto.

La clase **TextComponent** es un Componente que permite la edición de texto. Tiene un campo y no dispone de constructores públicos, por lo que no es posible instanciar objetos de esta clase. Sin embargo, sí dispone de un amplio repertorio de métodos que son heredados por sus subclases, que permiten la manipulación del texto. Entre esos métodos hay algunos muy interesantes, como son los que permiten la selección o recuperación del texto marcado, desde programa; la indicación de editabilidad de texto; la recuperación de los eventos producidos por ese Componente, etc.



En el programa, java1305 , que se implementa para ilustrar el uso de los campos de texto, se coloca un objeto **TextField** sobre un objeto **Frame**, inicializándolo con la cadena "Texto inicial", que aparecerá en el campo de texto al arrancar el programa, tal como aparece reproducido en la imagen.

Sobre el objeto **TextField** es instanciado y registrado un objeto **ActionListener**. Cuando se produce un evento porque el usuario haya pulsado la tecla Retorno mientras el objeto **TextField**, éste es recibido por el receptor de eventos, que en este ejemplo concreto, extrae y presenta en pantalla, en el **TextField**, el texto en dos formas: presentando todo el texto y también, presentando solamente el trozo de texto que esté seleccionado.

Cuando se pulsa la tecla Retorno mientras el campo de texto tiene el foco, el evento es capturado por el método sobrescrito *actionPerformed*() del objeto **ActionListener**. El código de este método utiliza el método *getSelectedItem*() de la clase **TextField** para acceder y presentar el texto que haya seleccionado el usuario. El código de este método

también invoca al método *getText*() de la clase **TextComponent**, para presentar el contenido completo del campo de texto.

```
import java.awt.*;
import java.awt.event.*;
public class java1305 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }
class IHM {
    public IHM() {
        // Instancia un objeto TextField y coloca una cadena como
        // Texto para que aparezca en el momento de su creación
        TextField miCampoTexto = new TextField( "Texto inicial" );
        // Instancia y registra un receptor de eventos de tipo Action
        // sobre el campo de texto
  miCampoTexto.addActionListener(new MiActionListener( miCampoTexto ) );
        // Coloca la eqtiqueta sobre el objeto Frame
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );
        miFrame.add( miCampoTexto );
        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );
        // Instancia y registra un objeto receptor de eventos de ventana
        // para concluir la ejecucion del programa cuando el Frame se
        // cierres por accion del usuario sobre el
        miFrame.addWindowListener( new Conclusion() );
        }
    }
// Clase para recibir los eventos de tipo Action que se produzcan
// sobre el objeto TextField sobre el cual se encuentra registrado
class MiActionListener implements ActionListener {
    TextField oCampoTexto;
    MiActionListener( TextField iCampoTexto ) {
        // Guarda una referencia al objeto TextField
        oCampoTexto = iCampoTexto;
        }
    // Se sobrescribe el método actionPerformed() del interfaz
    // ActionListener para que indique en la consola el texto que
    // se introduce
    public void actionPerformed( ActionEvent evt ) {
        System.out.println( "Texto seleccionado: " +
        oCampoTexto.getSelectedText() );
        System.out.println( "Texto completo: " +
        oCampoTexto.getText() );
        }
    }
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Concluye el programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

Las sentencias que se vuelven a reproducir a continuación, instancias un objeto TextField, inicializándolo con una cadena y luego, un objeto ActionListener es instanciado y registrado sobre el objeto TextField.

```
// Instancia un objeto TextField y coloca una cadena como
// Texto para que aparezca en el momento de su creación
TextField miCampoTexto = new TextField( "Texto inicial" );
// Instancia y registra un receptor de eventos de tipo Action
// sobre el campo de texto
miCampoTexto.addActionListener(new MiActionListener( miCampoTexto ) );
```

Hay más sentencias de código que son semejantes a las ya vistas en ejemplos anteriores, así que solamente es necesario recabar atención sobre el método *actionPerformed*(), en donde se invocan los métodos *getSelectedText*() y *getText*() para recuperar y presentar el texto.

```
public void actionPerformed( ActionEvent evt ) {
        System.out.println( "Texto seleccionado: " +
        oCampoTexto.getSelectedText() );
        System.out.println( "Texto completo: " +
        oCampoTexto.getText() );
        }
    }
```

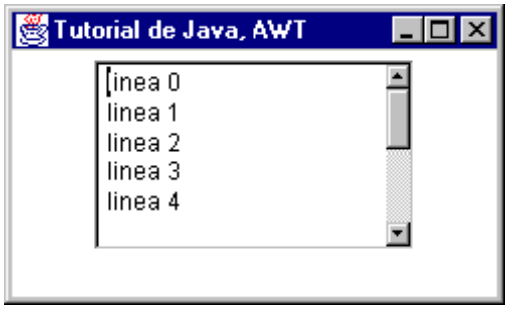
Areas de Texto

Un área de texto (**TextArea**) es una zona multilínea que permite la presentación de texto, que puede ser editable o de sólo lectura. Al igual que la clase **TextField**, esta clase extiende la clase **TextComponent** y dispone de cuatro campos, que son constantes simbólicas que pueden ser utilizadas para especificar la información de colocación de las barras de desplazamiento en algunos de los constructores de objetos **TextArea**. Estas constantes simbólicas son:

SCROLLBARS_BOTH	que crea y presenta barras de desplazamiento horizontal y vertical
SCROLLBARS_NONE	que no presenta barras de desplazamiento
SCROLLBARS_HORIZONTAL_ONLY	que crea y presenta solamente barras de desplazamiento horizontal
SCROLLBARS_VERTICAL_ONLY	que crea y presenta solamente barras de desplazamiento vertical

Esta clase **TextArea** contiene muchos métodos y, además, hay que tener en cuenta que hereda métodos definidos en las clases **TextComponent**, **Component** y **Object**, por lo que no queda más remedio que recurrir a la documentación del API que proporciona Sun para tener cumplida referencia de cada uno de ellos.

En el ejemplo,java1306 , coloca un objeto **TextArea** sobre un objeto **Frame**. Esta área de texto dispone de una barra de desplazamiento vertical y se instancia inicialmente con una cadena de diez líneas. La imagen siguiente reproduce la apariencia inicial de la ventana generada en el arranque del programa.



Sobre el objeto **TextArea** se instancia y registra un objeto **TextListener**, que recogerá eventos de tipo **TextEvent**, que se produce siempre que haya un cambio en el valor que contiene en área de texto.

Este tipo de eventos se capturan en el método sobrescrito **textValueChanged**() del objeto **TextListener**, que utiliza el método *getText*() de la clase **TextComponent** para acceder y presentar en pantalla todo el texto del objeto **TextArea**.

En este programa, el procesado del texto se realiza a muy bajo nivel, generándose un evento cada vez que cambie un solo carácter.

```
import java.awt.*;
import java.awt.event.*;
public class java1306 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }
class IHM {
    public IHM() {
        // Instancia un objeto TextArea, con una barra de desplazamiento
        // vertical y lo inicializa con diez líneas de texto
        TextArea miAreaTexto = new TextArea( "",5,20,
        TextArea.SCROLLBARS_VERTICAL_ONLY );
        for( int i=0; i < 10; i++ )
            miAreaTexto.append( "linea "+i+"\n" );
        // Instancia y registra un receptor de eventos de tipo Text
        // sobre el área de texto
        miAreaTexto.addTextListener(new MiTextListener( miAreaTexto ) );
        // Coloca el área de texto sobre el objeto Frame
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );
        miFrame.add( miAreaTexto );
        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );
        // Instancia y registra un objeto receptor de eventos de ventana
        // para concluir la ejecucion del programa cuando el Frame se
        // cierres por accion del usuario sobre el
        miFrame.addWindowListener( new Conclusion() );
        }
    }
// Clase para recibir los eventos de tipo Text que se produzcan
// sobre el objeto TextArea sobre el cual se encuentra registrado
class MiTextListener implements TextListener {
    TextArea oAreaTexto;
    MiTextListener( TextArea iAreaTexto ) {
        // Guarda una referencia al objeto TextArea
        oAreaTexto = iAreaTexto;
       }
    // Se sobrescribe el método textValueChanged() del interfaz
    // TextListener para que indique en la consola el texto que
    // ocupa el área de texto cuando se cambie
    public void textValueChanged( TextEvent evt ) {
        System.out.println( oAreaTexto.getText() );
       }
    }
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Concluye el programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

En el ejemplo anterior hay mucho código que ya se ha visto, y algunas líneas nuevas que se comentan a continuación. En primer lugar se encuentra el código que instancia el objeto de la clase **TextArea** con una barra de desplazamiento vertical y diez líneas de texto en ese objeto. Para añadir el texto al área, se utiliza el método *append*() de la clase **TextArea**, aunque hay varios métodos que también podrían haberse usado.

```
TextArea miAreaTexto = new TextArea( "",5,20,
    TextArea.SCROLLBARS_VERTICAL_ONLY );
for( int i=0; i < 10; i++ )
    miAreaTexto.append( "linea "+i+"\n" );
```

La línea siguiente instancia y registra un objeto TextListener sobre el objeto TextArea.

```
miAreaTexto.addTextListener(new MiTextListener( miAreaTexto ) );
```

El fragmento de código que se reproduce a continuación corresponde al método *textValueChanged*(), que utiliza el método *getText*() de la clase **TextComponent** para acceder y presentar todo el texto en el objeto TextArea siempre que haya un cambio en el valor del texto del objeto.

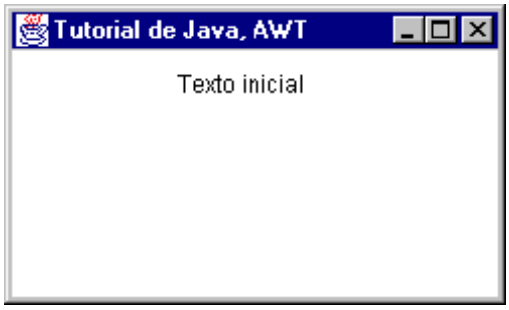
```
public void textValueChanged( TextEvent evt ) {
    System.out.println( oAreaTexto.getText() );
    }
```

Etiquetas

Una etiqueta (Label) proporciona una forma de colocar texto estático en un panel, para mostrar información fija, que no varía (normalmente), al usuario.

La clase Label extiende la clase Component y dispone de varias constantes que permiten especificar la alineación del texto sobre el objeto Label.

El ejemplo java1307, cuya imagen inicial al ejecutarlo se reproduce en la figura siguiente, es muy simple y muestra el uso normal de los objetos Label.



El programa no proporciona ningún control de eventos, porque las etiquetas normalmente no poseen ningún tipo de eventos, aunque hay que recordar que cualquier receptor de eventos de bajo nivel que se pueda registrar sobre la clase Component, también se podrá registrar sobre la clase Label, al ser ésta una subclase de Component.

```
import java.awt.*;
import java.awt.event.*;

public class java1307 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
```

```
        }
    }

class IHM {
    public IHM(){
        // Instancia un objeto Label con una cadena para inicializarlo y
       // que aparezca como contenido en el momento de su creación
        Label miEtiqueta = new Label( "Texto inicial" );

       // Coloca la eqtiqueta sobre el objeto Frame
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );
        miFrame.add( miEtiqueta );
        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );

        // Instancia y registra un objeto receptor de eventos de ventana
        // para concluir la ejecucion del programa cuando el Frame se
        // cierres por accion del usuario sobre el
        miFrame.addWindowListener( new Conclusion() );
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Concluye el programa cuando se cierra la ventana
        System.exit(0);
        }
    }
```

Canvas

Una zona de dibujo, o lienzo (Canvas), es una zona rectangular vacía de la pantalla sobre la cual una aplicación puede pintar, imitando el lienzo sobre el que un artista plasma su arte, o desde la cual una aplicación puede recuperar eventos producidos por acciones del usuario.

La clase Canvas existe para que se obtengan subclases a partir de ella. No hace nada por sí misma, solamente proporciona una forma de implementar Componentes propios. Por ejemplo, un canvas es útil a la hora de presentar imágenes o gráficos en pantalla, independientemente de que se quiera saber si se producen eventos o no en la zona de presentación.

Cuando se implementa una subclase de la clase Canvas, hay que prestar atención en implementar los métodos **minimumSize**() y **preferredSize**() para reflejar adecuadamente el tamaño de canvas; porque, en caso contrario, dependiendo del layout que utilice el contenedor del canvas, el canvas puede llegar a ser demasiado pequeño, incluso invisible.

La clase Canvas es muy simple, consiste en un solo constructor sin argumentos y dos métodos, que son:

AddNotify()--> Crea el observador del canvas

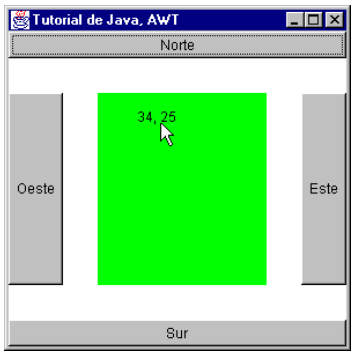
paint(Graphics)-->Repinta el canvas

En el ejemplo java1308 , no tiene mucha importancia el indicar el tamaño del canvas, porque el controlador de posicionamiento utilizado, un BorderLayout, ignora los

dos parámetros indicativos del tamaño que se han mencionado, y coloca en el Centro del layout el objeto Canvas.

El ejemplo muestra el uso de la clase Canvas y cómo se instancian objetos receptores que pueden manipular los objetos fuente sobre los que están registrados, sin necesidad de pasar referencias a esos objetos fuente a la hora de instanciar los objetos receptor. Por lo tanto, no se utilizan constructores parametrizados en la instanciación de los objetos receptores de eventos en este ejemplo.

La imagen siguiente muestra la primera apariencia en pantalla cuando se ejecuta la aplicación y se pica con el ratón sobre una posición dentro del canvas. Aparecen cuatro botones, sin funcionalidad alguna, y un objeto Canvas verde, sobre el objeto Frame; las coordenadas aparecen por haber sido pulsado el botón del ratón. Los botones están ahí simplemente para que se vea que el Frame puede contener más cosas que el Canvas.



Los cuatro botones están colocados en los bordes del Frame, utilizando un BorderLayout como controlador de posicionamiento. El Canvas aparece situado en el centro del Frame. Cuando se pulsa el botón del ratón con el cursor en el interior del canvas, aparecerán en pantalla las coordenadas en que se encuentra el ratón en el momento de la pulsación. El origen de los eventos del ratón es el objeto Canvas.

No se registran receptores de eventos sobre el Frame ni sobre los botones, por lo que si se pica sobre éstos, o sobre la zona de separación entre los componentes, no sucederá nada. Sin embargo, sí se controla el botón de cierre del Frame, de forma que si se pulsa sobre él, concluirá la ejecución del programa y el control volverá al sistema operativo.

```
import java.awt.*;
import java.awt.event.*;

// Se crea una subclase de Canvas para poder sobrescribir el método
// paint() y poder cambiar el fondo a color verde
class MiCanvas extends Canvas {
    int posicionX;
    int posicionY;

    public MiCanvas() {
        this.setBackground( Color.green );
       }

     // Se sobrescribe el método paint()
    public void paint( Graphics g ) {
        g.drawString( "" + posicionX + ", " + posicionY,
```

```
        posicionX,posicionY );
        }
    }

class java1308 extends Frame {
    public static void main( String args[] ) {
        // Se instancia un objeto del tipo de la clase
        new java1308();
        }

    public java1308() {
        // Se crea un borderlayout y se fija el espaciado entre los
        // componentes que va a albergar
        BorderLayout miLayout = new BorderLayout();
        miLayout.setVgap( 30 );
        miLayout.setHgap( 30 );

        this.setLayout( miLayout );
        this.setTitle( "Tutorial de Java, AWT" );
        this.setSize( 300,300 );

        // Se instancia un objeto de MiCanvas
        MiCanvas miObjCanvas = new MiCanvas();

         // Se añade el objeto MiCanvas creado al Centro del objeto
        // Frame a través del BorderLayout
        this.add( miObjCanvas,"Center" );

         // Se añaden los botones no-funcionales en los bordes del
        // objeto Frame a través del BorderLayout
        this.add( new Button( "Norte" ),"North" );
        this.add( new Button( "Sur" ),"South" );
        this.add( new Button( "Este" ),"East" );
        this.add( new Button( "Oeste" ),"West" );
        // Ahora se podrán ver
        this.setVisible( true );

        // Se instancia y registra un objeto receptor de eventos de la
        // ventana para poder concluir la aplicación cuando el usuario
        // cierre el Frame
        Conclusion conclusion = new Conclusion();
        this.addWindowListener( conclusion );

    // Se instancia y registra un objeto Listener para procesar los
    // eventos del ratón y poder determinar las coordenadas en que se
    // encuentra el cursor cada vez que el usuario pulse el botón sobre
    // el objeto MiCanvas.
    // El objeto receptor de eventos es instanciado anónimamente y no
    // tiene ninguna referencia de MiCanvas, ya que no se le pasa nada
    // en el constructor
        miObjCanvas.addMouseListener( new ProcRaton() );
        }
    }

   // Esta es la clase que monitoriza las pulsaciones de los botones del
ratón y presenta las coordenadas en que se encuentra el cursor cuando el
usuariorealiza la pulsación con el cursor situado en el interior del
objeto para el cual se ha registrado
class ProcRaton extends MouseAdapter {
     //Se sobrescribe el método mousePressed() para que haga lo que se ha
indicado
    public void mousePressed( MouseEvent evt ) {
     // Recoge las coordenadas x e y de la posición del cursor y las
   //  almacena en variables de instancia del objeto MiCanvas. Es
necesario
```

```
      // el casting para poder acceder a las variables de instancia
         ((MiCanvas)evt.getComponent()).posicionX = evt.getX();
         ((MiCanvas)evt.getComponent()).posicionY = evt.getY();
       // Se presentan las coordenadas en pantalla
         evt.getComponent().repaint();
         }
    }

// Concluye la ejecucion de la aplicacion cuando el usuario cierra la
// ventana, porque se genera un evento windowClosing
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

Este programa es muy similar al que sirve de ejemplo de explicación de la clase Frame, que presenta las coordenadas del cursor directamente sobre el área cliente de un objeto Frame. Hay algunas sentencias de código que merece la pena revisar un poco más detenidamente.

El primer trozo interesante de código es el que se utiliza para conseguir una subclase de la clase Canvas, para poder sobrescribir el método paint() y también hacer que el objeto sea verde desde el primer momento en que es instanciado.

```
class MiCanvas extends Canvas {
    int posicionX;
    int posicionY;

    public MiCanvas() {
        this.setBackground( Color.green );
       }

    // Se sobrescribe el método paint()
    public void paint( Graphics g ) {
        g.drawString( "" + posicionX + ", " + posicionY,
        posicionX,posicionY );
        }
    }
```

El siguiente fragmento interesante de código es el usado para crear un BorderLayout, con separaciones (gaps) horizontal y vertical, para el objeto Frame. Aunque el controlador de posicionamiento por defecto para el Frame es precisamente el BorderLayout, la indicación de separación entre Componentes no está incluida por defecto. Si se desea indicar esta separación, o gap, es necesario utilizar un controlador creado ex profeso, es decir, no se puede utilizar un objeto anónimo, sino uno con nombre para poder invocar el método que fija la separación entre Componentes.

```
BorderLayout miLayout = new BorderLayout();
miLayout.setVgap( 30 );
miLayout.setHgap( 30 );

this.setLayout( miLayout );
```

Otro grupo de sentencias interesantes son las que permiten instancias el objeto Canvas de la clase MiCanvas, que extiende a la clase Canvas y lo incorpora a un objeto Frame en su posición central. El objeto MiCanvas no se puede instanciar como objeto anónimo, porque sobre él se va a registrar posteriormente un receptor de eventos del ratón.

```
// Se instancia un objeto de MiCanvas
MiCanvas miObjCanvas = new MiCanvas();
```

```
// Se añade el objeto MiCanvas creado al Centro del objeto
// Frame a través del BorderLayout
this.add( miObjCanvas,"Center" );
```

Las sentencias anteriores están seguidas de código ya muy utilizado a la hora de incorporar botones a un objeto Frame y hacer que se visualicen en pantalla. También está el código que registra el receptor de eventos de la ventana utilizado para concluir la aplicación cuando se cierra el Frame.

Así que el siguiente trozo interesante es el fragmento de código que instancia y registra un objeto receptor de eventos que va a procesar los eventos del ratón, para determinar las coordenadas en que se encuentra el cursor en la pantalla, cuando el usuario pulsa el ratón. Este objeto Listener es instanciado anónimamente y no pasa referencia alguna del objeto MiCanvas al constructor del objeto Listener. Por lo tanto, el objeto receptor de eventos debe identificar el Componente sobre el cual ha de presentar la información de las coordenadas desde su propio código. Ya se verá que esta identificación la consigue basándose en el objeto MouseEvent que es pasar a este receptor cuando sucede un evento de este tipo.

```
miObjCanvas.addMouseListener( new ProcRaton() );
```

Ya, para concluir la revisión del ejemplo, se encuentra el código que define la clase Listener que va a presentar las coordenadas del cursor sobre el mismo objeto sobre el cual se ha registrado. Esta versión utiliza el método getComponent() sobre el mismo objeto MouseEvent que le llega para identificar el Componente que ha originado en evento. Este método devuelve una referencia a un objeto de tipo Component, luego es necesario hacer un moldeo hacia el tipo MiCanvas antes de poder acceder a las variables de instancia que se han definido para la clase MiCanvas.

```
class ProcRaton extends MouseAdapter {
    // Se sobrescribe el método mousePressed() para que haga lo que se ha
indicado
    public void mousePressed( MouseEvent evt ) {
    // Recoge las coordenadas x e y de la posición del cursor y las
  // almacena en variables de instancia del objeto MiCanvas. Es necesario
el casting para poder acceder a las variables de instancia
        ((MiCanvas)evt.getComponent()).posicionX = evt.getX();
        ((MiCanvas)evt.getComponent()).posicionY = evt.getY();
        // Se presentan las coordenadas en pantalla
        evt.getComponent().repaint();
        }
    }
```

El resto del código del programa ya está muy visto y no merece la pena el volver sobre él.

Barra de Desplazamiento

Las barras de desplazamiento (Scrollbar) se utilizan para permitir realizar ajustes de valores lineales en pantalla, porporcionan una forma de trabajar con rangos de valores o de áreas, como en el caso de un área de texto en donde se proporcionan las barras de desplazamiento de forma automática.

El ejemplo es muy sencillo y solamente presenta en pantalla tres barras de desplazamiento que podrían utilizarse como selector para fijar un color, en base a sus componentes básicos de rojo, verde y azul. La apariencia en pantalla es la que muestra la figura.

Y el código de este sencillo programa es el que se reproduce en las siguientes líneas.

```
import java.awt.*;
import java.awt.event.*;

class java1309 extends Frame {
    public static void main( String args[] ) {
        // Se instancia un objeto del tipo de la clase
        new java1309();
        }

    public java1309() {
       Scrollbar rojo,verde,azul;

       rojo = new Scrollbar( Scrollbar.VERTICAL,0,1,0,255 );
       verde = new Scrollbar( Scrollbar.VERTICAL,0,1,0,255 );
       azul = new Scrollbar( Scrollbar.VERTICAL,0,1,0,255 );

        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );

      // Se incorporan las tres barras de desplazamiento al objeto Frame
       miFrame.add( rojo );
       miFrame.add( verde );
       miFrame.add( azul );

       // Se fija el tamaño del Frame y se hace que aparezca todo
       // en pantalla
        miFrame.setSize( 250,100 );
        miFrame.setVisible( true );

       // Se instancia y registra un objeto receptor de eventos de la
       // ventana para poder concluir la aplicación cuando el usuario
       // cierre el Frame
        Conclusion conclusion = new Conclusion();
        miFrame.addWindowListener( conclusion );
       }
    }

// Concluye la ejecucion de la aplicacion cuando el usuario cierra la
// ventana, porque se genera un evento windowClosing
class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

Este tipo de interfaz proporciona al usuario un punto de referencia visual de un rango y al mismo tiempo la forma de cambiar los valores. Por ello, las barras de desplazamiento son Componentes un poco más complejos que los demás, reflejándose esta complejidad en sus constructores. Al crearlos hay que indicar su orientación, su valor

inicial, los valores mínimo y máximo que puede alcanzar y el porcentaje de rango que estará visible.

Si se utilizase una barra de desplazamiento para un rango de valores de color, tal como se hace en el ejemplo , en el cual se crea una barra de desplazamiento horizontal y en donde el ancho de esa barra será mayor, en relación al Scrollbar.

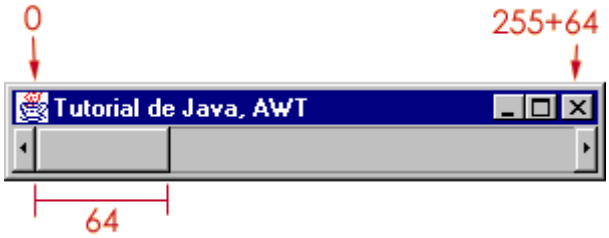
La imagen de la figura representa la captura de la ejecución inicial de la aplicación, modificada con las indicaciones explicativas de los valores.

Tanto en este ejemplo como en el anterior, que son muy sencillos, no se controlan los eventos generados por la actuación del usuario sobre la barra. Esto ya se hizo en ejemplos anteriores, . El código de este nuevo ejemplo, simplemente cambia la declaración e instanciación de las tres barras de desplazamiento del ejemplo por una barra horizontal, de la forma:

```
rango = new Scrollbar( Scrollbar.HORIZONTAL,0,64,0,255 );
```

En este caso, maxValue representa el valor máximo que va a alcanzar el lado izquierdo del indicador de la barra. Si se quieren representar 64 valores simultáneamente, es decir, de [0-63] a [192-255], maxValue debería ser 192.

El lector habrá observado que las barras de desplazamiento no proporcionan información textual a usuario sobre el valor exacto que está seleccionado, o una zona donde poder mostrar directamente los valores asociados a los desplazamientos. Si se desea proporcionar esa información, se ha de proveer explícitamente una caja de texto u otro objeto similar donde presentar esa información, tal como se muestra en el ejemplo , cuya imagen en ejecución es la que reproduce la figura siguiente.



El código del ejemplo en este caso ya recoge los eventos originados en la barra de desplazamiento. Cada vez que se produce un desplazamiento del indicador de la barra, se genera un evento de tipo Ajuste, que es recogido por el receptor registrado sobre la barra, que a su vez se encarga de presentar el valor numérico correspondiente a la posición actual en el campo de texto utilizado como indicador auxiliar.

Todo el ejemplo es una recopilación de la información proporcionada en los programas anteriores, y no merece la pena detenerse en ninguna de las sentencias que lo componen.

AWT-Contenedores

La clase Container es una clase abstracta derivada de Component, que representa a cualquier componente que pueda contener otros componentes. Se trata, en esencia, de añadir a la clase Component la funcionalidad de adición, sustracción, recuperación, control y organización de otros Componentes.

Al igual que la clase Component, no dispone de constructores públicos y, por lo tanto, no se pueden instanciar objetos de la clase Container. Sin embargo, sí se puede extender para implementar la nueva característica incorporada a Java en el JDK 1.1, de los componentes Lightweight.

El AWT proporciona varias clases de Contenedores:

- Panel
- Applet
- ScrollPane
- Window
- Dialog
- FileDialog
- Frame

Aunque los que se pueden considerar como verdaderos Contenedores con Window, Frame, Dialog y Panel, porque los demás son subtipos con algunas características determinadas y solamente útiles en circunstancias muy concretas.

Window

Es una superficie de pantalla de alto nivel (una ventana). Una instancia de la clase Window no puede estar enlazada o embebida en otro Contenedor.

El controlador de posicionamiento de Componentes por defecto, sobre un objeto Window, es el **BorderLayout.**

Una instancia de esta clase no tiene ni título ni borde, así que es un poco difícil de justificar su uso para la construcción directa de un interfaz gráfico, porque es mucho más sencillo utilizar objetos de tipo Frame o Dialog. Dispone de varios métodos para alterar el tamaño y título de la ventana, o los cursores y barrar de menús.

Frame

Es una superficie de pantalla de alto nivel (una ventana) con borde y título. Una instancia de la clase Frame puede tener una barra de menú. Una instancia de esta clase es mucho más aparente y más semejante a lo que se entiende por ventana.

Y, a no ser que el lector haya comenzado su estudio por esta página, ya se habrá encontrado en varias ocasiones con la clase Frame, que es utilizada en gran parte de los ejemplos de este Tutorial. Su uso se debe en gran parte a la facilidad de su instanciación y, lo que tampoco deja de ser interesante, su facilidad de conclusión.

La clase Frame extiende a la clase Window, y su controlador de posicionamiento de Componentes por defecto es el BorderLayout.

Los objetos de tipo Frame son capaces de generar varios tipos de eventos, de los cuales el más interesante es el evento de tipo WindowClosing, que se utiliza en este Tutorial de forma exhaustiva, y que se produce cuando el usuario pulsa sobre el botón de cerrar colocado en la esquina superior-derecha (normalmente) de la barra de título del objeto Frame.

En el ejemplo java1312 se ilustra el uso de la clase Frame y algunos de sus métodos. El programa instancia un objeto Frame con tres botones que realizan la acción que se indica en su título. La imagen reproduce la ventana que genera la aplicación y su situación tras haber pulsado el botón que cambia el cursor a forma de mano.

Es un ejemplo muy simple, aunque hay que advertir al lector que se hace uso en él de la sintaxis abreviada de las clases anidadas, que se tratarán en otra sección; para que no se asuste al ver el código del ejemplo. Este método se utiliza para instanciar y registrar receptores de eventos sobre los tres botones, más el de cerrar la ventana, colocados sobre el objeto Frame.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class java1312 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
```

```
        }
    }

class IHM {
    Frame miFrame;

    public IHM() {
        // Se instancian tres botones con textos indicando lo que
        // hacen cuando se pulse sobre ellos
        Button botonTitulo = new Button( "Imprime Titulo" );
        Button botonCursorMano = new Button( "Cursor Mano" );
        Button botonCursorFlecha = new Button( "Cursor Flecha" );

        // Instancia un objeto Frame con su titulo indicativo de que se
        // se trata, utilizando un FlowLayout
        miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );

          // Añade tres objetos Button al Frame
        miFrame.add( botonTitulo );
        miFrame.add( botonCursorMano );
        miFrame.add( botonCursorFlecha );

          // Fija el tamaño del Frame y lo hace visible
        miFrame.setSize( 250,200 );
        miFrame.setVisible( true );

        // Instancia y registra objetos ActionListener sobre los
        // tres botones utilizando la sintaxis abreviada de las
        // clases anidadas
        botonTitulo.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent evt ) {
                System.out.println( miFrame.getTitle() );
                }
            } );

        botonCursorMano.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent evt ) {
                miFrame.setCursor( new Cursor( Cursor.HAND_CURSOR ) );
                }
            } );

        botonCursorFlecha.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent evt ) {
                miFrame.setCursor( new Cursor( Cursor.DEFAULT_CURSOR ) );
                }
            } );

        // Instancia y registra un objeto WindowListener sobre el objeto
        // Frame para terminar el programa cuando el usuario haga click
        // con el raton sobre el boton de cerrar la ventana que se
        // coloca sobre el objeto Frame
        miFrame.addWindowListener( new WindowAdapter() {
            public void windowClosing( WindowEvent evt ) {
                // Concluye la aplicacion cuando el usuario cierra la
               // ventana
                System.exit( 0 );
                }
            } );
        }
    }
```

El siguiente trozo de código es el típico utilizado en la instanciación de un objeto Frame, la indicación del controlador de posicionamiento de Componentes que se va a

utilizar y, en este caso, la incorporación de los tres Componentes de tipo Button al objeto Frame.

```
// Se instancian tres botones con textos indicando lo que
// hacen cuando se pulse sobre ellos
Button botonTitulo = new Button( "Imprime Titulo" );
Button botonCursorMano = new Button( "Cursor Mano" );
Button botonCursorFlecha = new Button( "Cursor Flecha" );

// Instancia un objeto Frame con su titulo indicativo de que se
// se trata, utilizando un FlowLayout
miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.setLayout( new FlowLayout() );

// Añade tres objetos Button al Frame
miFrame.add( botonTitulo );
miFrame.add( botonCursorMano );
miFrame.add( botonCursorFlecha );
```

Y en los dos bloques de sentencias que se reproducen a continuación, se utilizan clases anidadas para instanciar y registrar objetos de tipo ActionListener. Por ejemplo, sobre el botón que permite recoger el título de la ventana, se hace tal como se indica.

```
botonTitulo.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent evt ) {
        System.out.println( miFrame.getTitle() );
        }
    } );
```

Y, para instanciar y registrar un objeto WindowListener sobre el objeto Frame, para concluir la aplicación cuando el usuario cierre la ventana, se emplean las siguientes líneas de código.

```
miFrame.addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent evt ) {
          // Concluye la aplicación cuando el usuario cierra la
        // ventana
        System.exit( 0 );
        }
    } );
```

Dialog

Es una superficie de pantalla de alto nivel (una ventana) con borde y título, que permite entradas al usuario. La clase Dialog extiende la clase Window, que extiende la clase Container, que extiende a la clase Component; y el controlador de posicionamiento por defecto es el BorderLayout.

De los constructores proporcionados por esta clase, destaca el que permite que el diálogo sea o no modal. Todos los constructores requieren un parámetro Frame y, algunos de ellos, permiten la especificación de un parámetro booleano que indica si la ventana que abre el diálogo será modal o no. Si es modal, todas las entradas del usuario serán recogidas por esta ventana, bloqueando cualquier entrada que se pudiese producir sobre otros objetos presentes en la pantalla. Posteriormente, si no se ha especificado que el diálogo sea modal, se puede hace que adquiera esta característica invocando al método setModal().

El ejemplo java1313, cuya imagen en pantalla al arrancar es la que reproduce la imagen que precede a este párrafo, presenta el mínimo código necesario para conseguir que un objeto Dialog aparezca sobre la pantalla. Cuando se arranca el programa, en la pantalla se visualizará un objeto Frame y un objeto Dialog, que debería tener la mitad de tamaño del objeto Frame y contener un título y un botón de cierre. Este botón de cierre no es operativo. El objeto Dialog no tiene la caja de control de la esquina superior izquierda al uso.

El objeto Dialog puede ser movido y redimensionado, aunque no se puede ni minimizar ni maximizar. Se pueden colocar en cualquier lugar de la pantalla, su posición no está restringida al interior del padre, el objeto Frame.

El objeto Dialog difiere significativamente en apariencia del objeto Frame, sobre todo por la presencia del borde en este último, circunstancia que llama mucho la atención.

```
import java.awt.*;
import java.awt.event.*;

public class java1313 extends Frame {
    public static void main( String args[] ) {
        // Instancia un objeto de este tipo
        new java1313();
        }

    // Constructor
    public java1313() {
        setTitle( "Tutorial de Java, AWT" );
        setSize( 250,150 );
        setVisible( true );

        Dialog miDialogo = new Dialog( this,"Dialogo" );
        miDialogo.setSize( 125,75 );
        // Hace que el dialogo aparezca en la pantalla
        miDialogo.show();
        }
    }
```

La parte más interesante del ejemplo reside en tres sentencias. La primera, instancia el objeto Dialog como hijo del objeto principal, this . La segunda sentencia establece el tamaño inicial del objeto Dialog. La tercera sentencia hace que el objeto Dialog aparezca en la pantalla.

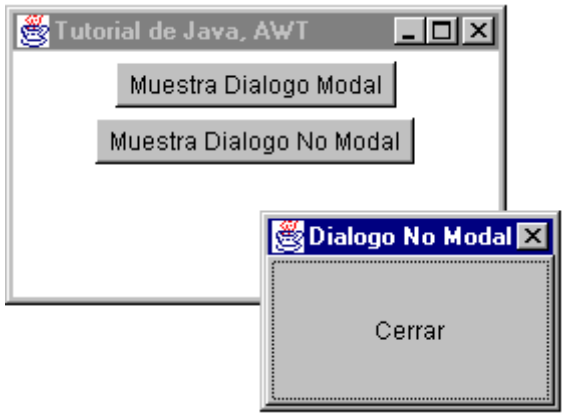
```
Dialog miDialogo = new Dialog( this,"Dialogo" );
miDialogo.setSize( 125,75 );
// Hace que el dialogo aparezca en la pantalla
miDialogo.show();
```

El ejemplo java1314, está diseñado para producir dos objetos Dialog, uno modal y otro no-modal. Un objeto Frame sirve de padre a los dos objetos Dialog.

El Dialog no-modal se crea con un botón que sirve para cerrarlo. Sobre este botón se instancia y registra un objeto ActionListener. Este objeto ActionListener es instanciado desde una clase ActionListener compartida. El código del método sobrescrito actionPerformed() de la clase ActionListener, cierra el objeto Dialog, invocando al método setVisible() con el parámetro false , aunque también se podría haber utilizado hide() o dispose().

El Dialog modal se crea de la misma forma, conteniendo un botón al que se asigna un cometido semejante.

Sobre el objeto Frame se crean dos botones adicionales, uno mostrará el diálogo modal y el otro mostrará el diálogo no-modal. Estos dos objetos Button comparten una clase ActionListener que está diseñada para mostrar un objeto Dialog de un tamaño predeterminado y en una posición parametrizada, controlada a través del parámetro offset, que se pasa al objeto ActionListener cuando se instancia.



La imagen muestra la ventana que genera la aplicación cuando se ejecuta por primera vez y se selecciona el Diálogo No-Modal, pulsando el botón correspondiente.

Para evitar que se superpongan, los objetos ActionListener de los dos botones que visualizan los objetos Dialog, se instancian con valores de offset diferentes, por lo que aparecerán en posiciones distintas de la pantalla en el momento de su visualización.

Si se compila y ejecuta el programa, aparecerán los dos botones en la pantalla, tal como muestra la imagen que aparece en párrafos anteriores. Uno de los botones puede ser utilizado para mostrar el objeto Dialog no-modal y el otro para visualizar el Dialog modal. Cuando el objeto Dialog modal no está visible, se podrá mostrar y cerrar el objeto Dialog no-modal, o pulsar en la caja de cierre del Frame para terminar la ejecución del programa. Sin embargo, cuando está visible el Dialog modal, no se podrá realizar ninguna otra acción dentro del programa; el modo de operación es el que se conoce como aplicación modal.

Un objeto receptor de eventos windowClosing() es instanciado y registrado sobre el Frame para concluir la ejecución del programa cuando se cierre el Frame; sin embargo, el Frame no puede cerrarse cuando el objeto Dialog modal está visible.

A continuación se comentan los trozos de código más interesantes del ejemplo anterior. El primero de ellos es la sentencia que instancia el objeto Frame, que a pesar de ser semejante a muchas de las ya vistas, lo que la hace importante en este programa es el ser padre de los dos objetos Dialog, por lo cual es imprescindible que sea instanciado antes de los dos objetos Dialog.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
```

El siguiente fragmento interesante es el típico código que es utilizado para instanciar los objetos Dialog, colocar un objeto Button en el objeto Dialog, e instanciar y registrar un objeto ActionListener sobre el objeto Button.

```
Dialog dialogoNoModal = new Dialog( miFrame,"Dialogo No Modal" );
Button botonCerrarNoModal = new Button( "Cerrar" );
dialogoNoModal.add( botonCerrarNoModal );
botonCerrarNoModal.addActionListener(
    new closeDialogListener( dialogoNoModal ) );
```

Este es el fragmento que crea el objeto Dialog modal. El que se usa para crear el objeto Dialog no-modal es esencialmente el mismo excepto que no tiene el parámetro booleando true en la invocación del constructor.

Este código es seguido por el que instancia los objetos Button y registra objetos ActionListener sobre estos botones. Y, a este código le sigue el que finaliza la construcción del objeto Frame, que ya se ha visto en varias ocasiones y no merece la pena insistir en él.

El trozo de código más interesante de todos es el fragmento en que la clase ActionListener es utilizada para instanciar objetos para mostrar un objeto Dialog. Y es muy interesante por dos aspectos. Uno de ellos es el constructor parametrizado que se utiliza para guardar el offset que se le pasa como parámetro al objeto ActionListener, cuando se instancia. Este valor de offset es combinado con valores en el código para controlar el tamaño y la posición del objeto Dialog cuando aparece en la pantalla. El otro aspecto interesante es que el método show() de la clase Dialog es utilizado para hacer aparecer el objeto Dialog en la pantalla.

```
class showDialogListener implements ActionListener {
    Dialog oDialog;
    int oOffset;

    showDialogListener( Dialog dialogo,int offset ) {
        oDialog = dialogo;
        oOffset = offset;
        }

    public void actionPerformed( ActionEvent evt ) {
        // Seguir este orden es critico para un dialogo modal
        oDialog.setBounds( oOffset,oOffset,150,100 );
        oDialog.show();
        }
    }
```

El orden de ejecución de las sentencias dentro del método actionPerformed() es crítico. Si el método show() se ejecuta antes del método setBounds() sobre el objeto Dialog modal, el método setBounds() utilizado para controlar el tamaño y posición del objeto

Dialog, no tendría efecto alguno. Tamaño y posición del diálogo deben establecerse antes de hacerlo visible.

El último fragmento de código interesante es el método sobrescrito actionPerformed() utilizado para cerrar los objetos Dialog. En este ejemplo se usa la llamada al método setVisible() con el parámetro false, aunque también se podría haber utilizado el método hide() o el método dispose(), con el mismo cometido.

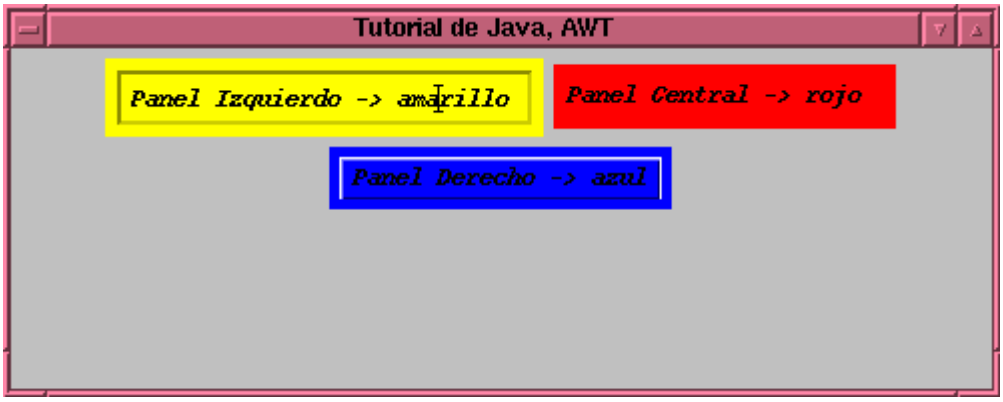
```
public void actionPerformed( ActionEvent evt ) {
    oDialog.setVisible( false );
    }
```

Panel

La clase Panel es un Contenedor genérico de Componentes. Una instancia de la clase Panel, simplemente proporciona un Contenedor al que ir añadiendo Componentes.

El controlador de posicionamiento de Componentes sobre un objeto Panel, por defecto es el FlowLayout; aunque se puede especificar uno diferente en el constructor a la hora de instanciar el objeto Panel, o aceptar el controlador de posicionamiento inicialmente, y después cambiarlo invocando al método setLayout().

Panel dispone de un método addNotify(), que se utiliza para crear un observador general (peerPerr) del Panel. Normalmente, un Panel no tiene manifestación visual alguna por sí mismo, aunque puede hacerse notar fijando su color de fondo por defecto a uno diferente del que utiliza normalmente.



El ejemplo java1315, ilustra la utilización de objetos Panel para configurar un objeto de tipo interfaz gráfica, o interfaz hombre-máquina, incorporando tres objetos Panel a un objeto Frame.

El controlador de posicionamiento de los Componentes para el objeto Frame se especifica concretamente para que sea un FlowLayout, y se alteran los colores de fondo de los objetos Panel para que sean claramente visibles sobre el Frame. Sobre cada uno de los paneles se coloca un objeto, utilizando el método add(), de tal modo que se añade un objeto de tipo campo de texto sobre el Panel de fondo amarillo, un objeto de tipo etiqueta sobre el Panel de fondo rojo y un objeto de tipo botón sobre el Panel de fondo azul.

Ninguno de los Componentes es activo, ya que no se instancian ni registran objetos receptores de eventos sobre ellos. Así, por ejemplo, el único efecto que se puede observar

al pulsar el botón del panel azul, se limita al efecto visual de la pulsación. Sin embargo, sí se instancia y registra un receptor de eventos sobre el Frame, para recoger la intención del usuario de cerrar la ventana y terminar la ejecución de la aplicación.

```
import java.awt.*;
import java.awt.event.*;

public class java1315 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        // Se construyen tres Paneles con fondos de color diferente
        // y sin contener ningun elemento activo
        Panel panelIzqdo = new Panel();
        panelIzqdo.setBackground( Color.yellow );
        panelIzqdo.add( new TextField( "Panel Izquierdo -> amarillo" ) );

        Panel panelCentral = new Panel();
        panelCentral.setBackground( Color.red );
        panelCentral.add( new Label( "Panel Central -> rojo" ) );

        Panel panelDrcho = new Panel();
        panelDrcho.setBackground( Color.blue );
        panelDrcho.add( new Button( "Panel Derecho -> azul" ) );

        // Se instancia un objeto Frame utilizando un FlowLayout y
        // se colocan los tres objetos Panel sobre el Frame
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( new FlowLayout() );

        miFrame.add( panelIzqdo );
        miFrame.add( panelCentral );
        miFrame.add( panelDrcho );
        miFrame.setSize( 500,200 );
        miFrame.setVisible( true );

        miFrame.addWindowListener( new Conclusion() );
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Concluye la aplicacion cuando el usuario cierra la ventana
        System.exit( 0 );
        }
    }
```

Las sentencias de código más interesantes del ejemplo se limitan a la típica instanciación de los tres objetos Panel, al control del color de fondo y a la incorporación de otro Componente al Panel, tal como se reproduce en las siguientes sentencias.

```
Panel panelIzqdo = new Panel();
panelIzqdo.setBackground( Color.yellow );
panelIzqdo.add( new TextField( "Panel Izquierdo -> amarillo" ) );
```

Las siguientes líneas de código instancian un objeto Frame y le añaden los tres objetos Panel construidos anteriormente.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.setLayout( new FlowLayout() );
```

```
miFrame.add( panelIzqdo );
miFrame.add( panelCentral );
miFrame.add( panelDrcho );
```

Con este trozo de código se genera el objeto de más alto nivel de la aplicación para el interfaz de usuario.

Añadir Componentes a un Contenedor

Para que un interfaz sea útil, no debe estár compuesto solamente por Contenedores, éstos deben tener Componentes en su interior. Los Componentes se añaden al Contenedor invocando al método add() del Contenedor. Este método tiene tres formas de llamada que dependen del manejador de composición o layout manager que se vaya a utilizar sobre el Contenedor.

En el código siguiente,java1316 , se incorporan dos botones al Contenedor de tipo Frame. La creación se realiza en el método init() porque éste siempre es llamado automáticamente al inicializarse el applet. De todos modos, al inciarse la ejecución se crean los botones, ya que el método init() es llamado tanto por el navegador como por el método main().

```
import java.awt.*;

public class java1316 extends java.applet.Applet {

    public void init() {
        add( new Button( "Uno" ) );
        add( new Button( "Dos" ) );
        }

    public static void main( String args[] ) {
        Frame f = new Frame( "Tutorial de Java" );
        java1316 ejemplo = new java1316();

        ejemplo.init();

        f.add( "Center",ejemplo );
        f.pack();
        f.show();
        }
    }
```

El ejemplo también muestra la forma de cómo el código puede ejecutarse tanto como aplicación, utilizando el intérprete de Java, como desde un navegador, funcionando como cualquiera de los applets que se han visto en ejemplos anteriores. En ambos casos el resultado, en lo que al ejemplo se refiere, es el mismo: aparecerán dos botones en el campo delimitado por el Contenedor Frame.

Los Componentes añadidos a un objeto Container entran en una lista cuyo orden define el orden en que se van a presentar los Componentes sobre el Contenedor, de atrás hacia delante. Si no se especifica ningún índice de orden en el momento de incorporar un Componente al Contenedor, ese Componente se añadirá al final de la lista. Hay que tener esto muy en cuenta, sobre todo a la hora de construir interfaces de usuario complejas, en las que pueda haber Componentes que solapen a otros Componentes o a parte de ellos.

AWT-Menus

No hay ningún método para diseñar un buen interfaz de usuario, todo depende del programador. Los Menús son siempre el centro de la aplicación, porque son el medio de que el usuario interactúe con esa aplicación. La diferencia entre una aplicación útil y otra que es totalmente frustrante radica en la organización de los menús, pero eso, las reglas del diseño de un buen árbol de menús, no están claras. Hay un montón de libros acerca de la ergonomía y de cómo se debe implementar la interacción con el usuario. Lo cierto es que por cada uno que defienda una idea, seguro que hay otro que defiende la contraria. Todavía no hay un acuerdo para crear un estándar, con cada Window Manager se publica una guía de estilo diferente. Así que, vamos a explicar lo básico, sin que se deba tomar como dogma de fe, para que luego cada uno haga lo que mejor le parezca.

En Java, la jerarquía de clases que intervienen en la construcción y manipulación de menús es la que se muestra en la lista siguiente:

```
java.lang.Object
    MenuShorcut
    java.awt.MenuComponent
        java.awt.MenuBar
        java.awt.MenuItem
            java.awt.Menu
            java.awt.CheckboxMenuItem
            java.awt.PopMenu
```

MenuComponent, es la superclase de todos los Componentes relacionados con menús.

MenuShortcut, representa el acelerador de teclado, o la combinación de teclas rápidas, para acceder a un MenuItem.

MenuItem, representa una opción en un menú.

Menu, es un Componente de una barra de menú.

MenuBar, encapsula el concepto de una barra de menú en un Frame.

PopupMenu, implementa un menú que puede ser presentado dinámicamente dentro de un Componente.

CheckboxMenuItem, genera una caja de selección que representa una opción en un menú.

A continuación se exploran una a una las clases que se acaban de citar.

Clase **MenuComponent**

Esta clase no contiene campos, solamente tiene un constructor y dispone de una docena de métodos que están accesibles a todas sus subclases.

Clase Menu

Esta es la clase que se utiliza para construir los menús que se manejan habitualmente, conocidos como menús de persiana (o pull-down). Dispone de varios constructores para poder, entre otras cosas, crear los menús con o sin etiqueta. No tiene campos y proporciona varios métodos que se pueden utilizar para crear y mantener los

menús en tiempo de ejecución. En el programa de ejemplo java1317 , se usarán algunos de ellos.

Clase MenuItem

Esta clase se emplea para instanciar los objetos que constituirán los elementos seleccionables del menú. No tiene campos y dispone de varios constructores, entre los que hay que citar a:

```
MenuItem( String, MenuShortcut );
```

que crea un elemento el menú con una combinación de teclas asociada para acceder directamente a él.

Esta clase proporciona una veintena de métodos, entre los que destacan los que se citan ahora:

addActionListener(ActionListener) que añade el receptor específico que va a recibir eventos desde esa opción del menú

removeActionListener(ActionListener) contrario al anterior, por lo que ya no se recibirán eventos desde esa opción del menú

setEnabled(boolean) indica si esa opción del menú puede estar o no seleccionable

isEnabled() comprobación de si la opción del menú esta habilitada

El método addActionListener() ya debería resultar familiar al lector. Cuando se selecciona una opción de un menú, bien a través del ratón o por la combinación rápida de teclas, se genera un evento de tipo ActionEvent. Para que la selección de la opción en un menú ejecute una determinada acción, se ha de instanciar y registrar un objeto ActionListener que contenga el método actionPerformed() sobreescrito para producir la acción deseada. En el ejemplo java1317 , solamente se presenta en pantalla la identificación de la opción de menú que se ha seleccionado; en un programa realmente útil, la acción seguramente que deberá realizar algo más interesante que eso.

Clase MenuShortcut

Esta clase se utiliza para instanciar un objeto que representa un acelerador de teclado, o una combinación de teclas rápidas, para un determinado MenuItem. No tiene campos y dispone de dos constructores.

Aparentemente, casi todas las teclas rápidas consisten en mantener pulsada la tecla Control a la vez que se pulsa cualquier otra tecla. Uno de los constructores de esta clase:

```
MenuShortcut( int,boolean );
```

dispone de un segundo parámetro que indica si el usuario ha de mantener también pulsada la tecla de cambio a mayúsculas (Shift). El primer parámetro es el código de la tecla, que es el mismo que se devuelve en el campo keyCode del evento KeyEvent, cuando se pulsa una tecla.

La clase KeyEvent define varias constantes simbólicas para estos códigos de teclas, como son: VK_8, VK_9, VK_A, VK_B.

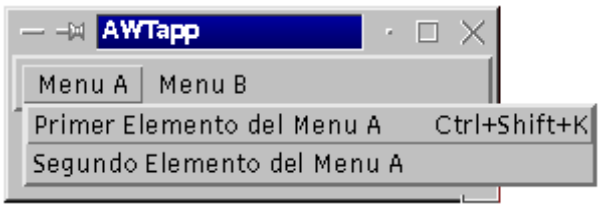
Clase MenuBar

No tiene campos, sólo tiene un constructor público, y es la clase que representa el concepto que todo usuario tiene de la barra de menú que está presente en la mayoría de las aplicaciones gráficas basadas en ventanas.

En el programa java1317, ilustra algunos de los aspectos que intervienen en los menús. Es una aplicación que coloca dos menús sobre un objeto Frame. Uno de los menús tiene dos opciones y el otro, tres. La primera opción del primer menú también tiene asignada una combinación de teclas rápidas: Ctrl+Shift+K.

Cuando se selecciona un elemento del menú, éste genera un evento de tipo ActionEvent, que presenta una línea de texto en pantalla indicando cuál ha sido el elemento del menú que se ha seleccionado, por ejemplo:

```
% java java1317
java.awt.MenuItem[menuitem0,label=Primer    Elemento    del    Menu    A,
shortcut=Ctrl+Shift+K]
java.awt.MenuItem[menuitem1,label=Segundo Elemento del Menu A]
java.awt.MenuItem[menuitem0,label=Primer    Elemento    del    Menu    A,
shortcut=Ctrl+Shift+K]
java.awt.MenuItem[menuitem3,label=Segundo Elemento del Menu B]
```



También se instancia y registra un objeto receptor de eventos windowClosing() para terminar la ejecución del programa cuando se cierra el Frame.

```
import java.awt.*;
import java.awt.event.*;

public class java1317 {
    public static void main(String args[]){
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        // Se instancia un objeto de tipo Acelerador de Teclado
        MenuShortcut miAcelerador =
            new MenuShortcut( KeyEvent.VK_K,true );

        // Se instancian varios objetos de tipo Elementos de Menu
        MenuItem primerElementoDeA = new MenuItem(
            "Primer Elemento del Menu A",miAcelerador );
        MenuItem segundoElementoDeA = new MenuItem(
            "Segundo Elemento del Menu A" );
        MenuItem primerElementoDeB = new MenuItem(
            "Primer Elemento del Menu B" );
        MenuItem segundoElementoDeB = new MenuItem(
```

```
            "Segundo Elemento del Menu B" );
        MenuItem tercerElementoDeB = new MenuItem(
            "Tercer Elemento del Menu B" );

        // Se instancia un objeto ActionListener y se registra sobre los
        // objetos MenuItem
        primerElementoDeA.addActionListener( new MiGestorDeMenu() );
        segundoElementoDeA.addActionListener( new MiGestorDeMenu() );
        primerElementoDeB.addActionListener( new MiGestorDeMenu() );
        segundoElementoDeB.addActionListener( new MiGestorDeMenu() );
        tercerElementoDeB.addActionListener( new MiGestorDeMenu() );

        // Se instancian dos objetos de tipo Menu y se les añaden los
        // objetos MenuItem
        Menu menuA = new Menu( "Menu A" );
        menuA.add( primerElementoDeA );
        menuA.add( segundoElementoDeA );

        Menu menuB = new Menu( "Menu B" );
        menuB.add( primerElementoDeB );
        menuB.add( segundoElementoDeB );
        menuB.add( tercerElementoDeB );

        // Se instancia una Barra de Menu y se le añaden los Menus
        MenuBar menuBar = new MenuBar();
        menuBar.add( menuA );
        menuBar.add( menuB );

       // Se instancia un objeto Frame y se le asocia el objeto MenuBar.
        // Observese que esta no es la tipico invocacion del metodo
        // miFrame.add(), sino que es una forma especial de invocar
       // al metodo necesaria para poder asociar un objeto Barra de Menu
        // a un objeto Frame
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        // Esto no es el metodo add(), como se podria esperar
        miFrame.setMenuBar( menuBar );
        miFrame.setSize( 250,100 );
        miFrame.setVisible( true );

       // Se instancia y registra un receptor de eventos de ventana para
        // concluir el programa cuando se cierre el Frame
        miFrame.addWindowListener( new Conclusion() );
        }
    }

// Clase para instanciar un objeto ActionListener que se registra
// sobre los elementos del menu
class MiGestorDeMenu implements ActionListener {
    public void actionPerformed( ActionEvent evt ) {
        // Presenta en pantalla el elemento que ha generado el evento
        // de tipo Action
        System.out.println( evt.getSource() );
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    }
```

A continuación se revisan los fragmentos de código más interesantes del ejemplo, aunque en el programa se puede observar que hay gran cantidad de código repetitivo, ya

que esencialmente se necesitan las mismas sentencias para crear cada opción del menú y registrar un objeto receptor de eventos sobre cada una de ellas.

El programa genera dos menús separados con el mismo código básicamente. Quizá lo más interesante no sea el código en sí mismo, sino el orden en que se realizan los diferentes pasos de la construcción del menú.

La primera sentencia que merece la pena es la que instancia un objeto MenuShortcut, que se utilizará posteriormente en la instanciación de un objeto MenuItem.

```
MenuShortcut miAcelerador =
    new MenuShortcut( KeyEvent.VK_K,true );
```

La lista de argumentos del constructor especifica la combinación de teclas que se van a utilizar y el parámetro true, indica que es necesaria la tecla del cambio a mayúsculas (Shift) pulsada. El primer parámetro es la constante simbólica que la clase KeyEvent define para la tecla K y, aparentemente, la tecla Control siempre debe estar pulsada en este tipo de combinaciones para ser utilizadas las teclas normales como aceleradores de teclado.

Cuando se utiliza este constructor, en el menú aparecerá la etiqueta asignada y a su derecha, la combinación de teclas alternativas para activarla directamente.

El siguiente fragmento de código que se puede ver es el típico de instanciación de objetos MenuItem, que posteriormente se añadirán al objeto Menu para crear el menú. Se muestran dos tipos de instrucciones, el primer estilo especifica un acelerador de teclado y el segundo no.

```
MenuItem primerElementoDeA = new MenuItem(
    "Primer Elemento del Menu A",miAcelerador );
MenuItem segundoElementoDeA = new MenuItem(
    "Segundo Elemento del Menu A" );
```

Ahora se encuentra el código que instancia y registra un objeto ActionListener sobre la opción del menú, tal como se ha visto en ejemplos anteriores. Aquí solamente se incluye para ilustrar el hecho de que la asociación de objetos ActionListener con opciones de un menú, no difiere en absoluto de la asociación de objetos ActionListener con objetos Button, o cualquier otro tipo de objeto capaz de generar eventos de tipo ActionEvent.

```
primerElementoDeA.addActionListener( new MiGestorDeMenu() );
```

Las sentencias que siguen son las ya vistas, y que aquí se emplean para instanciar cada uno de los dos objetos Menu y añadirles las opciones existentes. Es la típica llamada al método Objeto.add() que se ha utilizado en programas anteriores.

```
Menu menuA = new Menu( "Menu A" );
menuA.add( primerElementoDeA );
```

El siguiente fragmento de código instancia un objeto MenuBar y le añade los dos menús que se han definido antes.

```
MenuBar menuBar = new MenuBar();
menuBar.add( menuA );
menuBar.add( menuB );
```

En este momento ya están creados los dos objetos Menu y colocados en un objeto MenuBar. Sin embargo, no se ha dicho nada sobre el ensamblamiento del conjunto. Esto se hace en el momento de asociar el objeto MenuBar con el objeto Frame. En este caso no se puede utilizar el método add(), sino que se tiene que invocar a un método especial de la clase Frame que tiene la siguiente declaración:

public synchronized void setMenuBar(MenuBar mb)

y en este caso concreto se hace en las sentencias que se reproducen seguidamente:

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
// Esto no es el metodo add(), como se podria esperar
miFrame.setMenuBar( menuBar );
```

Y hasta aquí lo más interesante del ejemplo, porque el código que resta es similar al que ya se ha visto y descrito en otros ejemplos del Tutorial, y que el lector se habrá encontrado si ha seguido la lectura secuencialmente.

Clase CheckboxMenuItem

Esta clase se utiliza para instanciar objetos que puedan utilizarse como opciones en un menú. Al contrario que las opciones de menú que se han visto al hablar de objetos MenuItem, estas opciones tienen mucho más parentesco con las cajas de selección, tal como se podrá comprobar a lo largo del ejemplo java1318.

Esta clase no tiene campos y proporciona tres constructores públicos, en donde se puede especificar el texto de la opción y el estado en que se encuentra. Si no se indica nada, la opción estará deseleccionada, aunque hay un constructor que permite indicar en un parámetro de tipo booleano, que la opción se encuentra seleccionada, o marcada, indicando true en ese valor.

De los métodos que proporciona la clase, quizá el más interesante sea el método que tiene la siguiente declaración:

addItemListener(ItemListener)

Cuando se selecciona una opción del menú, se genera un evento de tipo ItemEvent. Para que se produzca la acción que se desea con esa selección, es necesario instanciar y registrar un objeto ItemListener que contenga el método itemStateChanged() sobrescrito con la acción que se quiere. Por ejemplo, en el programa que se presenta a continuación, la acción consistirá en presentar la identificación y estado de la opción de menú que se haya seleccionado.

Cuando se ejecuta el programa java1318 , aparece un menú sobre un objeto Frame. El menú contiene tres opciones de tipo CheckboxMenuItem. Una opción de este tipo es semejante a cualquier otra, hasta que se selecciona. Cuando se seleccione, aparecerá una marca, o cualquier otra identificación visual, para saber que esa opción está seleccionada. Estas acciones hacen que el estado de la opción cambie, y ese estado se puede conocer a través del método getState().

Cuando se selecciona una opción, se genera un evento de tipo ItemEvent, que contiene información del nuevo estado, del texto de la opción y del nombre asignado a la opción. Estos datos pueden utilizarse para identificar cuál de las opciones ha cambiado y, también, para implementar la acción requerida que, en este caso del ejemplo siguiente, consiste en presentar una línea de texto en la pantalla con la información que contiene el objeto ItemEvent, más o menos semejante a la que se reproduce a continuación:

```
% java java1318
java.awt.CheckboxMenuItem[chkmenuitem0,label=Primer Elemento,state=true]
java.awt.CheckboxMenuItem[chkmenuitem2,label=Terecer Elemento,state=true]
java.awt.CheckboxMenuItem[chkmenuitem0,label=Primer Elemento,state=false]
java.awt.CheckboxMenuItem[chkmenuitem1,label=Segundo Elemento,state=true]
java.awt.CheckboxMenuItem[chkmenuitem0,label=Primer Elemento,state=true]
```

Como siempre, se instancia y registra sobre el Frame un objeto receptor de eventos windowClosing() para la conclusión del programa cuando se cierre el Frame.

```
import java.awt.*;
import java.awt.event.*;

public class java1318 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        // Instancia objetos de tipo CheckboxMenuItem
        CheckboxMenuItem primerElementoMenu =
            new CheckboxMenuItem( "Primer Elemento" );
        CheckboxMenuItem segundoElementoMenu =
            new CheckboxMenuItem( "Segundo Elemento" );
        CheckboxMenuItem tercerElementoMenu =
            new CheckboxMenuItem( "Tercer Elemento" );

        // Instancia un objeto ItemListener y lo registra sobre los
        // objetos CheckboxMenuItem, elementos del menu de seleccion
        primerElementoMenu.addItemListener( new ControladorCheckBox() );
       segundoElementoMenu.addItemListener( new ControladorCheckBox() );
        tercerElementoMenu.addItemListener( new ControladorCheckBox() );

        // Instancia un objeto Menu y le añade los botones de la caja
        // de seleccion
        Menu menuA = new Menu( "Menu A" );
        menuA.add( primerElementoMenu );
        menuA.add( segundoElementoMenu );
        menuA.add( tercerElementoMenu );

        // Instancia un objeto MenuBar y le añade el objeto Menu
        MenuBar barraMenu = new MenuBar();
        barraMenu.add( menuA );

       // Se instancia un objeto Frame y se le asocia el objeto MenuBar.
        // Observese que esta no es la tipico invocacion del metodo
        // miFrame.add(), sino que es una forma especial de invocar
       // al metodo necesaria para poder asociar un objeto Barra de Menu
       // a un objeto Frame
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );

        // Esto no es el metodo add(), como se podria esperar
        miFrame.setMenuBar( barraMenu );
```

```
        miFrame.setSize( 250,100 );
        miFrame.setVisible( true );

        // Instancia y registra un receptor de eventos de ventana para
        // concluir la ejecucion del programa cuando se cierra el Frame
        miFrame.addWindowListener( new Conclusion() );
        }
    }

// Clase para instanciar un objeto ItemListener y registrarlo
// sobre los elementos del menu
class ControladorCheckBox implements ItemListener {
    public void itemStateChanged( ItemEvent evt ) {
        // Presenta en pantalla el elemento del menu que ha
        // generado el evento
        System.out.println( evt.getSource() );
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Termina el programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

Hay algunos trozos de código en el programa que resultan repetitivos, debido al hecho de que se utiliza básicamente el mismo código para crear cada elemento del menú y registrar un objeto Listener sobre él.

El orden en que se instancian y asocian las opciones es importante. La primera sentencia de código interesante es la que se utiliza para instanciar varios objetos CheckboxMenuItem que serán añadidos al objeto Menu para producir un menú con varias opciones.

```
CheckboxMenuItem primerElementoMenu =
    new CheckboxMenuItem( "Primer Elemento" );
```

La sentencia que se reproduce ahora es interesante solamente por lo que tiene de novedad, porque hace uso de un tipo de clase Listener que no se ha visto antes. La sentencia, por otro lado, es la típica que se requiere para instanciar y registrar objetos ItemListener sobre cada CheckboxMenuItem.

```
primerElementoMenu.addItemListener( new ControladorCheckBox() );
```

A continuación se encuentra el código que instancia un objeto Menu y le añade los objetos CheckboxMenuItem, que es semejante al utilizado en los menús normales. El siguiente código interesante corresponde a la clase que implementa el interfaz ItemListener, que vuelve a resultar interesante por lo novedoso.

```
class ControladorCheckBox implements ItemListener {
    public void itemStateChanged( ItemEvent evt ) {
        // Presenta en pantalla el elemento del menu que ha
        // generado el evento
        System.out.println( evt.getSource() );
        }   }
```

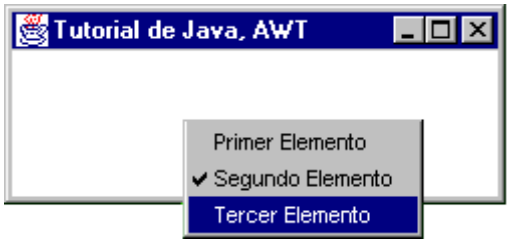
Y el resto del programa es similar al de ejemplos anteriores, así que no se vuelve más sobre él.

Clase PopupMenu

Esta clase se utiliza para instanciar objetos que funcionan como menús emergentes o pop-up. Una vez que el menú aparece en pantalla, el procesado de las opciones es el mismo que en el caso de los menús de persiana.

Esta clase no tiene campos y proporciona un par de constructores y un par de métodos, de los cuales el más interesante es el método **show**(), que permite mostrar el menú emergente en una posición relativa al Componente origen. Este Componente origen debe estar contenido dentro de la jerarquía de padres de la clase PopupMenu.

El programa siguiente,java1319 , coloca un objeto PopupMenu sobre un objeto Frame. El menú contiene tres opciones de tipo CheckboxMenuItem, y aparece cuando se pica dentro del Frame, posicionando su esquina superior-izquierda en la posición en que se encontraba el ratón en el momento de pulsar el botón



El resto del funcionamiento es semejante al de los programas de ejemplo vistos en secciones anteriores

```
import java.awt.*;
import java.awt.event.*;

public class java1319 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        // Instancia objetos CheckboxMenuItem
        CheckboxMenuItem primerElementoMenu =
            new CheckboxMenuItem( "Primer Elemento" );
        CheckboxMenuItem segundoElementoMenu =
            new CheckboxMenuItem( "Segundo Elemento" );
        CheckboxMenuItem tercerElementoMenu =
            new CheckboxMenuItem( "Tercer Elemento" );

        // Se instancia un objeto ItemListener y se registra sobre los
        // elementos de menu ya instanciados
        primerElementoMenu.addItemListener( new ControladorCheckBox() );
       segundoElementoMenu.addItemListener( new ControladorCheckBox() );
tercerElementoMenu.addItemListener( new ControladorCheckBox() );


       // Instancia un objeto Menu de tipo PopUp y le añade los objetos
        // CheckboxMenuItem
        PopupMenu miMenuPopup = new PopupMenu( "Menu Popup" );
        miMenuPopup.add( primerElementoMenu );
        miMenuPopup.add( segundoElementoMenu );
```

```
        miMenuPopup.add( tercerElementoMenu );

        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
    miFrame.addMouseListener ( new ControladorRaton( miFrame,miMenuPopup)
);
        // Aquí está la diferencia con los Menus de Barra
        miFrame.add( miMenuPopup );
        miFrame.setSize( 250,100 );
        miFrame.setVisible( true );

        // Instancia y registra un receptor de eventos de ventana para
        // terminar el programa cuando se cierra el Frame
        miFrame.addWindowListener( new Conclusion() );
        }
    }

// Clase para atrapar los eventos de pulsacion del raton y presentar
// en la pantalla el objeto menu Popup, en la posicion en que se
// encontraba el cursor
class ControladorRaton extends MouseAdapter{
    Frame aFrame;
    PopupMenu aMenuPopup;

    // Constructor parametrizado
    ControladorRaton( Frame frame,PopupMenu menuPopup ) {
        aFrame = frame;
        aMenuPopup = menuPopup;
        }

    public void mousePressed( MouseEvent evt ) {
        // Presenta el menu PopUp sobre el Frame que se especifique
        // y en las coordenadas determinadas por el click del raton,
        // cuidando de que las coordenadas no se encuentren situadas
        // sobre la barra de titulo, porque las coordenadas Y en
        // esta zona son negativas
        if( evt.getY() > 0 )
            aMenuPopup.show( aFrame,evt.getX(),evt.getY() );
        }
    }

// Clase para instanciar un objeto receptor de eventos de los
// elementos del menu que sera registrado sobre estos elementos
class ControladorCheckBox implements ItemListener {
    public void itemStateChanged( ItemEvent evt ) {
        // Presenta en pantalla el elemento que ha generado el
        // evento
        System.out.println( evt.getSource() );
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // termina el programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

El programa es muy similar al de la sección anterior en que se utilizaban objetos CheckboxMenuItem en un menú de persiana normal, así que solamente se verá el código que afecta al uso del objeto PopupMenu.

El primer fragmento de código interesante es el que instancia un objeto PopupMenu y le incorpora tres objetos CheckboxMenuItem.

```
PopupMenu miMenuPopup = new PopupMenu( "Menu Popup" );
```

```
miMenuPopup.add( primerElementoMenu );
miMenuPopup.add( segundoElementoMenu );
miMenuPopup.add( tercerElementoMenu );
```

El siguiente trozo de código es interesante porque el procedimiento en el caso de asociar un objeto PopupMenu con el objeto Frame, es diferente a cuando se utiliza un objeto MenuBar. En el caso de la barra de menú, se utiliza la llamada al método setMenuBar(), mientras que para asociar un menú emergente a un objeto Frame, se utiliza la típica llamada de la forma Objeto.add(), que es el método habitual de añadir muchos otros Componentes a un Contenedor.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.addMouseListener( new ControladorRaton(miFrame,miMenuPopup) );
// Aquí está la diferencia con los Menus de Barra
miFrame.add( miMenuPopup );
```

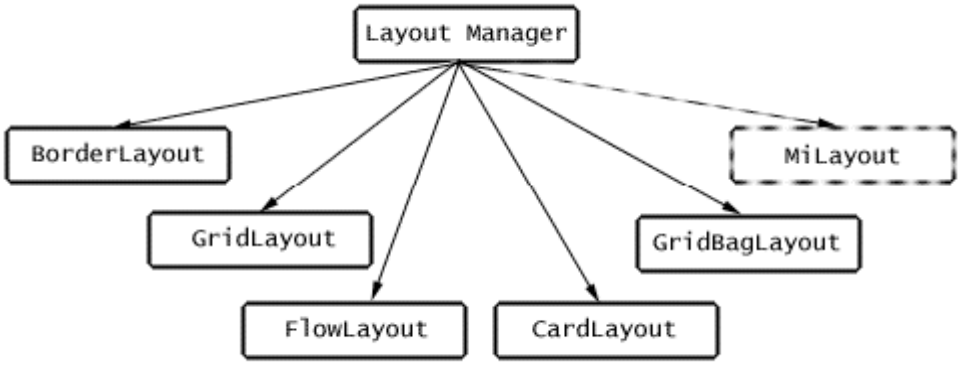
El siguiente trozo de código interesante es el método sobrescrito mousePressed(), del interfaz MouseListener, encapsulado aquí en la clase ControladorRaton. El propósito de esta clase es instanciar un objeto receptor que atrapará eventos de pulsación del ratón sobre el objeto Frame, y mostrará el objeto PopupMenu en la posición en que se encuentre el cursor del ratón en el momento de producirse el evento.

```
    public void mousePressed( MouseEvent evt ) {
    // Presenta el menu PopUp sobre el Frame que se especifique
    // y en las coordenadas determinadas por el click del raton,
    // cuidando de que las coordenadas no se encuentren situadas
    // sobre la barra de titulo, porque las coordenadas Y en
    // esta zona son negativas
    if( evt.getY() > 0 )
        aMenuPopup.show( aFrame,evt.getX(),evt.getY() );
    }
```

Como se puede observar, es lo típico, excepto el uso del método show() de la clase PopupMenu. También se debe notar la referencia al objeto PopupMenu y otra al objeto Frame, que se pasan cuando se instancia el objeto. Estas dos referencias son necesarias para la invocación del método show(), tal como se muestra en el código anterior. La referencia al objeto Frame se utiliza para establecer la posición en donde aparecerá el menú, y la referencia al objeto PopupMenu especifica el menú que se debe mostrar.

AWT - Layouts (I)

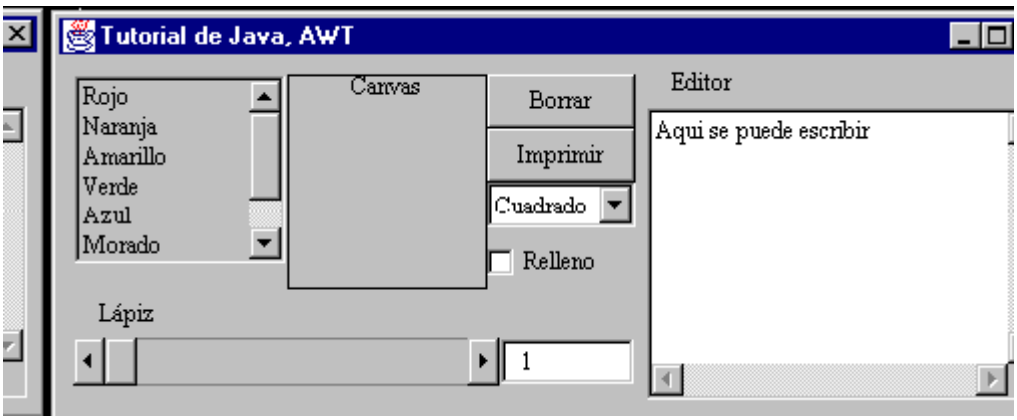
Los layout managers o manejadores de composición, en traducción literal, ayudan a adaptar los diversos Componentes que se desean incorporar a un Panel, es decir, especifican la apariencia que tendrán los Componentes a la hora de colocarlos sobre un Contenedor, controlando tamaño y posición (layout) automáticamente. Java dispone de varios, en la actual versión, tal como se muestra en la imagen:



¿Por qué Java proporciona estos esquemas predefinidos de disposición de componentes? La razón es simple: imaginemos que se desean agrupar objetos de distinto tamaño en celdas de una rejilla virtual: si confiados en nuestro conocimiento de un sistema gráfico determinado, se codificase a mano tal disposición, se debería preveer el redimensionamiento del applet, su repintado cuando sea cubierto por otra ventana, etc., además de todas las cuestiones relacionadas con un posible cambio de plataforma (uno nunca sabe a donde van a ir a parar los propios hijos, o los applets).

Sigamos imaginando, ahora, que un hábil equipo de desarrollo ha previsto las disposiciones gráficas más usadas y ha creado un gestor para cada una de tales configuraciones, que se ocupará, de forma transparente para nosotros, de todas esas cuitas de formatos. Bien, pues estos gestores son instancias de las distintas clases derivadas de LayoutManager y que se utilizan en el applet que genera la figura siguiente, donde se muestran los diferentes tipos de layouts que proporciona el AWT.

El ejemplo java1320.java, ilustra el uso de paneles, listas, barras de desplazamiento, botones, selectores, campos de texto, áreas de texto y varios tipos de layouts.



En el tratamiento de los Layouts se utiliza un método de validación, de forma que los Componentes son marcados como no válidos cuando un cambio de estado afecta a la geometría o cuando el Contenedor tiene un hijo incorporado o eliminado. La validación se

realiza automáticamente cuando se llama a pack() o show(). Los Componentes visibles marcados como no válidos no se validan automáticamente.

FlowLayout

Es el más simple y el que se utiliza por defecto en todos los Paneles si no se fuerza el uso de alguno de los otros. Los Componentes añadidos a un Panel con FlowLayout se encadenan en forma de lista. La cadena es horizontal, de izquierda a derecha, y se puede seleccionar el espaciado entre cada Componente.

Si el Contenedor se cambia de tamaño en tiempo de ejecución, las posiciones de los Componentes se ajustarán automáticamente, para colocar el máximo número posible de Componentes en la primera línea.

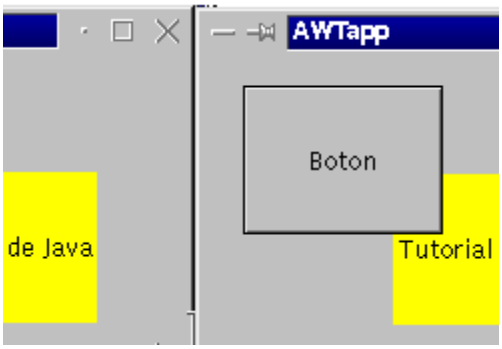
Los Componentes se alinean según se indique en el constructor. Si no se indica nada, se considera que los Componentes que pueden estar en una misma línea estarán centrados, pero también se puede indicar que se alineen a izquierda o derecha en el Contenedor.

El ejemplo que se presenta a continuación, java1321.java, es muy sencillito y lo que hace es colocar cinco objetos Button, sin funcionalidad alguna, sobre un objeto Frame, utilizando como controlador de posicionamiento un FlowManager. Los botones no son funcionales porque no se registra ningún objeto receptor de eventos sobre ellos.

```
import java.awt.*;
import java.awt.event.*;

public class java1321 {
    public static void main( String args[] ) {
    // Instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
    }

// La siguiente clase se utiliza para instanciar un objeto de tipo
// Interfaz Grafica de Usuario
class IHM {
    public IHM() {
        // Se crea un objeto Button con el texto que se pasa como
        // parametro y el tamaño y posicion indicadas dentro de
        // su contenedor (en pixels)
        Button miBoton = new Button( "Boton" );
        // Al rectamgulo se le pasan los parametros: x,y,ancho,alto
        miBoton.setBounds( new Rectangle( 25,20,100,75 ) );

        // Se crea un objeto Label con el texto que se indique como
        // parametro en la llamada y el tamaño especificado y en la
        // posicion que se indique dentro de su contenedor (en pixels)
        // Se pone en amarillo para que destaque
        Label miEtiqueta = new Label( "Tutorial de Java" );
        miEtiqueta.setBounds( new Rectangle( 100,75,100,75 ) );
        miEtiqueta.setBackground( Color.yellow );

        // Se crea un objeto Frame con el titulo que se indica en la
        // lamada y sin ningun layout
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
```

```
        miFrame.setLayout( null );

        // Añade los dos componentes al Frame, fijando su tamaño en
        // pixels y lo hace visible
        miFrame.add( miBoton );
        miFrame.add( miEtiqueta );
        miFrame.setSize( 250,175 );
        miFrame.setVisible( true );
        }
    }
```

Al compilar y ejecutar el programa, en pantalla aparecerá inicialmente una ventana como la que se muestra en la siguiente imagen



.

Como se puede observar en el código del ejemplo, el objeto FlowLayout se construye con alineación izquierda, una separación horizontal entre Componentes de 10 pixels y una separación vertical de 15 pixels. Si se cambia de tamaño manualmente al Frame, ya con el programa en ejecución, las posiciones de los Componentes se ajustan automáticamente, colocando el número máximo posible de ellos en la primera línea.

Ahora se presenta otro ejemplo, java1322.java, que aunque tampoco haga nada espectacular, por lo menos permite ver cómo se modifica un layout dinámicamente en tiempo de ejecución. En el programa se añaden cinco botones a un Frame utilizando un objeto FlowLayout como manejador de posicionamiento de estos botones, fijando una separación de 3 pixels entre los Componentes, tanto en dirección horizontal como vertical.

Se instancia y registra un objeto receptor de eventos de tipo acción para recoger los eventos de los cinco botones. La acción del controlador de eventos es incrementar el espacio entre los Componentes en 5 pixels al pulsar cualquiera de los botones. Esto se consigue incrementando los atributos Vgap y Hgap del objeto FlowLayout, fijando como controlador de posicionamiento el layout modificado y validando el Frame. Este último paso es imprescindible para que los cambios tengan efecto y se hagan visibles.

También se instancia y registra un objeto receptor de eventos windowClosing() para terminar el programa cuando se cierre el Frame. El código del ejemplo es el siguiente.

```
import java.awt.*;
import java.awt.event.*;

public class java1322 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
```

```
        }
    }

class IHM {
    public IHM() {
       Frame miFrame = new Frame( "Tutorial de Java, AWT" );
       // Instancia un objeto FlowLayout object alieado al Centro
       // y con una separacion de 3 pixels en horizonal y vertical
       FlowLayout miFlowLayout = new FlowLayout( FlowLayout.CENTER,3,3 );

        // Se fija este FlowLayout para que sea el controlador de
        // posicionamiento de componentes para el objeto Frame
        miFrame.setLayout( miFlowLayout );

        // Se instancian cinco objetos Button, para indicar los
        // posicionamientos del FlowLayout
        Button boton1 = new Button( "Primero" );
        Button boton2 = new Button( "Segundo" );
        Button boton3 = new Button( "Tercero" );
        Button boton4 = new Button( "Cuarto" );
        Button boton5 = new Button( "Quinto" );

        // Se añaden los cinco botones al Frame en las mismas posiciones
        // que vienen dadas por las etiquetas que se les han asignado en
        // el constructor
        miFrame.add( boton1 );
        miFrame.add( boton2 );
        miFrame.add( boton3 );
        miFrame.add( boton4 );
        miFrame.add( boton5 );

        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );

        // Instancia un objeto receptor de eventos de tipo action y
        // lo registra para los cinco botones que se han añadido al
        // objeto Frame
        MiReceptorAction miReceptorAction =
            new MiReceptorAction( miFlowLayout,miFrame );
        boton1.addActionListener( miReceptorAction );
        boton2.addActionListener( miReceptorAction );
        boton3.addActionListener( miReceptorAction );
        boton4.addActionListener( miReceptorAction );
        boton5.addActionListener( miReceptorAction );

        // Se instancia y registra un receptor de eventos de ventana
        // para terminar la ejecucion del programa cuando se cierre
        // el Frame
        miFrame.addWindowListener( new Conclusion() );
        }
    }

class MiReceptorAction implements ActionListener {
    FlowLayout miObjLayout;
    Frame miObjFrame;

    MiReceptorAction( FlowLayout layout,Frame frame ) {
        miObjLayout = layout;
        miObjFrame = frame;
        }

    // Cuando sucede un evento Action, se incrementa el espacio que
    // que hay entre los componentes en el objeto FlowLayout.
    // Luego se fija el controlador de posicionamiento al nuevo
    // que se construye, y luego se valida el Frame para asegurar
```

```
    // que se actualiza en la pantalla
    public void actionPerformed( ActionEvent evt ){
        miObjLayout.setHgap( miObjLayout.getHgap()+5 );
        miObjLayout.setVgap( miObjLayout.getVgap()+5 );
        miObjFrame.setLayout( miObjLayout );
        miObjFrame.validate();
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Termina el programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

El programa contiene algunas sentencias que merecen un comentario. Por ejemplo, el código que se reproduce a continuación instancia un objeto FlowLayout con alineación central y con 3 pixels de separación entre Componentes, tanto vertical como horizontalmente. Este objeto se pasa al método setLayout() para que sea el controlador de posicionamiento para el objeto Frame.

```
// Instancia un objeto FlowLayout object alieado al Centro
// y con una separacion de 3 pixels en horizonal y vertical
FlowLayout miFlowLayout = new FlowLayout( FlowLayout.CENTER,3,3 );
// Se fija este FlowLayout para que sea el controlador de
// posicionamiento de componentes para el objeto Frame
miFrame.setLayout( miFlowLayout );
```

La siguientes sentencias se utilizan para instanciar un objeto receptor de eventos Action, un ActionListener, y lo registran sobre los cinco botones:

```
MiReceptorAction miReceptorAction =
    new MiReceptorAction( miFlowLayout,miFrame );
boton1.addActionListener( miReceptorAction );
```

Y el código que sigue, ya es el programa controlador de eventos, que modifica el layout dinámicamente en tiempo de ejecución. Este código responde, cuando se pulsa cualquiera de los botones, utilizando los métodos get() y set() sobre los atributos de separación vertical, Vgap, y horizontal, Hgap, del FlowLayout; modificando el espaciado entre Componentes. Luego, se utiliza el método setLayout() para hacer que el objeto Layout así modificado sea el controlador de posicionamiento del objeto Frame, y, por fin, se hace que los cambios sean efectivos y se visualicen en pantalla validando el objeto Frame, a través de una llamada al método de validación, validate().

```
public void actionPerformed( ActionEvent evt ){
    miObjLayout.setHgap( miObjLayout.getHgap()+5 );
    miObjLayout.setVgap( miObjLayout.getVgap()+5 );
    miObjFrame.setLayout( miObjLayout );
    miObjFrame.validate();
    }
```

BorderLayout

La composición BorderLayout (de borde) proporciona un esquema más complejo de colocación de los Componentes en un panel. La composición utiliza cinco zonas para

colocar los Componentes sobre ellas: Norte, Sur, Este, Oeste y Centro. Es el layout o composición que se utilizan por defecto Frame y Dialog.

El Norte ocupa la parte superior del panel, el Este ocupa el lado derecho, Sur la zona inferior y Oeste el lado izquierdo. Centro representa el resto que queda, una vez que se hayan rellenado las otras cuatro partes. Así, este controlador de posicionamiento resuelve los problemas de cambio de plataforma de ejecución de la aplicación, pero limita el número de Componentes que pueden ser colocados en Contenedor a cinco; aunque, si se va a construir un interfaz gráfico complejo, algunos de estos cinco Componentes pueden Contenedores, con lo cual el número de Componentes puede verse ampliado.

En los cuatro lados, los Componentes se colocan y redimensionan de acuerdo a sus tamaños preferidos y a los valores de separación que se hayan fijado al Contenedor. El tamaño prefijado y el tamaño mínimo son dos informaciones muy importantes en este caso, ya que un botón puede ser redimensionado a proporciones cualesquiera; sin embargo, el diseñador puede fijar un tamaño preferido para la mejor apariencia del botón. El controlador de posicionamiento puede utilizar este tamaño cuando no haya indicaciones de separación en el Contenedor, o puede ignorarlo, dependiendo del esquema que utilice. Ahora bien, si se coloca una etiqueta en el botón, se puede indicar un tamaño mínimo de ese botón para que siempre sea visible, al menos, el rótulo del botón. En este caso, el controlador de posicionamiento muestra un total respeto a este valor y garantiza que por lo menos ese espacio estará disponible para el botón.

El ejemplo que se verá a continuación, java1323.java, es muy simple. Lo que hace es crear una ventana a través de un objeto Frame y colocar cinco objetos Button, sin funcionalidad alguna, utilizando un BorderLayout como manejador de composición. A uno de los botones se le ha colocado un texto más largo, para que el controlador de posicionamiento reserve espacio de acuerdo al tamaño mínimo que se indique para ese botón.

```
import java.awt.*;
import java.awt.event.*;

public class java1323 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );

        miFrame.add( new Button( "Sur" ),"South" );
        miFrame.add( new Button( "Oeste" ),"West" );
        miFrame.add( new Button( "Este" ),"North" );
        miFrame.add( new Button( "Boton del Este" ),"East" );
        miFrame.add( new Button( "Centro" ),"Center" );
        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );
        }
    }
```

Si se compila y ejecuta este programa, aparece en pantalla una imagen como la que se reproduce a continuación:

Se observará que se puede cambiar de tamaño el objeto Frame, y que dentro de los límites, los Componentes se van cambiado a la vez, para acomodarse al tamaño que va adoptando la ventana. Aunque todavía hay problemas que no se han eliminado, y se podrá redimensionar la ventana para conseguir que los botones desaparezcan, o se trunquen los rótulos. Sin embargo, es una forma muy flexible de posicionar Componentes en una ventana y no tener que preocuparse de su redimensionamiento y colocación dentro de unos límites normales y sin buscarle las cosquillas al sistema.

El siguiente ejemplo, java1324.java, aunque tampoco hace nada particularmente interesante, sí tiene más sustancia que el anterior e ilustra algunos aspectos adicionales del BorderLayout. En el programa se añaden cinco botones a un Frame utilizando un objeto BorderLayout como manejador de posicionamiento de estos botones, fijando una separación de 3 pixels entre los Componentes, tanto en dirección horizontal como vertical.

Se instancia y registra un objeto receptor de eventos de tipo acción para recoger los eventos de los cinco botones. La acción del controlador de eventos es incrementar el espacio entre los Componentes en 5 pixels al pulsar cualquiera de los botones. Esto se consigue incrementando los atributos Vgap y Hgap del objeto BorderLayout, fijando como controlador de posicionamiento el layout modificado y validando el Frame. Este último paso es imprescindible para que los cambios tengan efecto y se hagan visibles.

También se instancia y registra un objeto receptor de eventos windowClosing() para terminar el programa cuando se cierre el Frame.

```
import java.awt.*;
import java.awt.event.*;

public class java1324 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );
    // Se instancia un objeto BorderLayout con una holgura en vertical y
    // horizontal de 3 pixels
    BorderLayout miBorderLayout = new BorderLayout( 3,3 );
    // Se fija este BorderLayout para que sea el controlador de
    // posicionamiento de componentes para el objeto Frame
    miFrame.setLayout( miBorderLayout );

    // Se instancian cinco objetos Button, para indicar los
    // posicionamientos del BorderLayout
        Button boton1 = new Button( "Sur" );
        Button boton2 = new Button( "Oeste" );
```

```
        Button boton3 = new Button( "Norte" );
        Button boton4 = new Button( "Este" );
        Button boton5 = new Button( "Centro" );

        // Se añaden los cinco botones al Frame en las mismas posiciones
        // que vienen dadas por las etiquetas que se les han asignado en
        // el constructor
        miFrame.add( boton1,"South" );
        miFrame.add( boton2,"West" );
        miFrame.add( boton3,"North" );
        miFrame.add( boton4,"East" );
        miFrame.add( boton5,"Center" );

        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );

        // Instancia un objeto receptor de eventos de tipo action y
        // lo registra para los cinco botones que se han añadido al
        // objeto Frame
        MiReceptorAction miReceptorAction =
            new MiReceptorAction( miBorderLayout,miFrame );
        boton1.addActionListener( miReceptorAction );
        boton2.addActionListener( miReceptorAction );
        boton3.addActionListener( miReceptorAction );
        boton4.addActionListener( miReceptorAction );
        boton5.addActionListener( miReceptorAction );

        // Se instancia y registra un receptor de eventos de ventana
        // para terminar la ejecucion del programa cuando se cierre
        // el Frame
        miFrame.addWindowListener( new Conclusion() );
        }
    }

class MiReceptorAction implements ActionListener{
    BorderLayout miObjBorderLayout;
    Frame miObjFrame;

    MiReceptorAction( BorderLayout layout,Frame frame ) {
        miObjBorderLayout = layout;
        miObjFrame = frame;
        }

    // Cuando sucede un evento Action, se incrementa el espacio que
    // que hay entre los componentes en el objeto BorderLayout.
    // Luego se fija el controlador de posicionamiento al nuevo
    // que se construye, y luego se valida el Frame para asegurar
    // que se actualiza en la pantalla
    public void actionPerformed( ActionEvent evt ) {
        miObjBorderLayout.setHgap( miObjBorderLayout.getHgap()+5 );
        miObjBorderLayout.setVgap( miObjBorderLayout.getVgap()+5 );
        miObjFrame.setLayout( miObjBorderLayout );
        miObjFrame.validate();
        }
    }

class Conclusion extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        // Termina el programa cuando se cierra la ventana
        System.exit( 0 );
        }
    }
```

Seguidamente se comentan algunas de las sentencias más interesantes que constituyen el código del programa. Por ejemplo, las líneas de código siguientes:

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
// Se instancia un objeto BorderLayout con una holgura en vertical y
// horizontal de 3 pixels
BorderLayout miBorderLayout = new BorderLayout( 3,3 );
// Se fija este BorderLayout para que sea el controlador de
// posicionamiento de componentes para el objeto Frame
miFrame.setLayout( miBorderLayout );
```

instancian un objeto Frame, también instancian un objeto BorderLayout con una separación de tres pixels entres Componentes y establecen ese objeto BorderLayout como controlador de posicionamiento de Componentes del Frame.

Las sentencias

```
Button boton1 = new Button( "Sur" );
. . .
miFrame.add( boton1,"South" );
```

Son las típicas utilizadas para la instanciación de los objetos Button y para añadir estos objetos al objeto Frame.

Las sentencias siguientes son las que instancian un objeto receptor de eventos de tipo Action, y lo registran sobre los cinco botones.

```
MiReceptorAction miReceptorAction =
    new MiReceptorAction( miBorderLayout,miFrame );
boton1.addActionListener( miReceptorAction );
```

El código que sigue ahora, ya en el controlador de eventos actionPerformed() utiliza los métodos getHgap(), getVgap(), setHgap() y setVgap() para modificar los atributos de separación entre Componentes en el BorderLayout. Este BorderLayout modificado es utilizado, en conjunción con setLayout(), para convertirlo en el controlador de posicionamiento del objeto Frame. Y, por último, se utiliza el método validate() para forzar al objeto Frame a reajustar el tamaño y posición de los Componentes y presentar en pantalla la versión modificada de sí mismo.

```
public void actionPerformed( ActionEvent evt ) {
    miObjBorderLayout.setHgap( miObjBorderLayout.getHgap()+5 );
    miObjBorderLayout.setVgap( miObjBorderLayout.getVgap()+5 );
    miObjFrame.setLayout( miObjBorderLayout );
    miObjFrame.validate();
    }
```

AWT - Layouts (II)

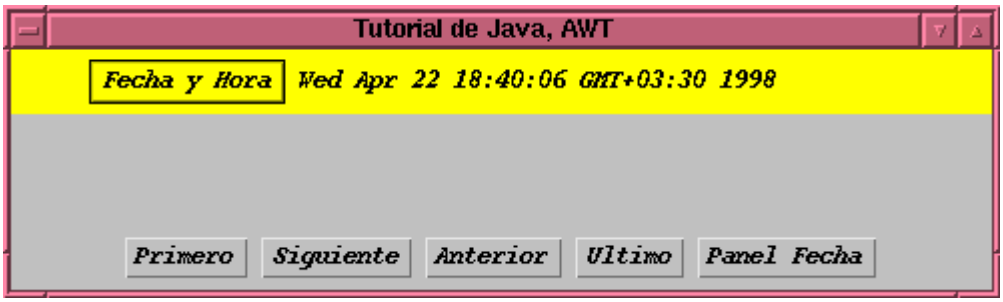
CardLayout

Este es el tipo de composición que se utiliza cuando se necesita una zona de la ventana que permita colocar distintos Componentes en esa misma zona. Este layout suele ir asociado con botones de selección (Choice), de tal modo que cada selección determina el panel (grupo de componentes) que se presentarán.

En el ejemplo java1327.java, con el que se ilustra el uso de este tipo de controlador de posicionamiento, se creará u objeto interfaz de usuario basado en un objeto Frame, que contiene a los dos objetos Panel que lo conforman. Uno de los objetos panel servirá como pantalla sobre la que presentar las diversas fichas (cards), utilizando el CardLayout. El otro

Panel contiene varios objetos Button que se pueden utilizar para cambiar las fichas que se presentan en el Panel contrario.

El programa es un poco más complejo que los ejemplos anteriores de los controladores de posición, debido a que el CardLayout también es más complicado, pero por el contrario, esta complejidad le proporciona un considerable poder y funcionalidad a la hora de su empleo.



Todos los botones de las fichas son pasivos, no tienen registrados controladores de eventos, excepto uno de ellos, que contiene un botón sobre el cual, al pulsarlo, se presenta la fecha y la hora del sistema en esa misma ficha. Esto se consigue a través de objetos de tipo ActionListener que funcionan a dos niveles. En un primer nivel, los objetos ActionListener se utilizan para seleccionar la ficha que se visualizará. Y en un segundo nivel, se registra un objeto ActionListener para uno de los botones de una de las fichas, que hará que aparezca en pantalla la fecha y la hora, siempre que la ficha que contiene el botón esté visible y se pulse ese botón. La visualización de esta ficha en la ventana, durante la ejecución del ejemplo, es la misma imagen que se reproduce en la siguiente figura.

En el programa, se crea un objeto Frame a un nivel superior del interfaz, que contiene dos objetos Panel. Uno de los objetos Panel es el panel de visualización que utiliza el programa para presentar cada una de las siguientes fichas, que se añaden al Panel utilizando el CardLayout como controlador de posicionamiento:

Una ficha con el botón "La Primera ficha es un Boton"

Una ficha con la etiqueta "La Segunda ficha es un Label"

Una ficha con la etiqueta "Tercera ficha, tambien un Label"

Una ficha con la etiqueta "Cuarta ficha, de nuevo un Label"

Una ficha, que es el "panel fecha", con el botón "Fecha y Hora" y la etiqueta donde presenta la fecha

Una ficha con un campo de texto, inicializado con la cadena "La Ultima ficha es un campo de texto"

Todos los Componentes de cada una de las fichas son pasivos, no tienen ningún controlador de eventos registrado sobre ellos; excepto la ficha identificada como panelFecha, que consiste en un objeto panel al que se han incorporado un objeto Button y un objeto Label. Esta ficha, o tarjeta, no es pasiva, porque hay un objeto ActionListener instanciado y registrado sobre el botón que presenta la fecha y la hora en el objeto Label de esta misma ficha.

El otro Panel, que contiene el objeto Frame principal, es el panel de control, que contiene a su vez a cinco botones, cuatro de desplazamiento y otro que va a permitir la

visualización directa de la ficha en que se presenta la fecha y la hora. Todos los botones son activos, ya que disponen de controladores de eventos registrados sobre ellos, y la acción que realizan es la que indica su etiqueta; por ejemplo, si se pulsa el botón "siguiente", aparecerá en la ventana la siguiente ficha a la que se esté visualizando en ese momento.

Además, se instancia y registra un objeto receptor de eventos windowClosing() sobre el Frame para concluir la ejecución del programa cuando el usuario pulsa el botón de cierre del Frame.

Ahora se discuten las partes más interesantes del código del programa que, al ser más complejo que los vistos en secciones anteriores, también tiene más trozos de código que merecen un comentario.

El primer trozo de código interesante que se encuentra en la visión del programa es el usado para crear una cualquiera de las fichas, que contiene un objeto Button y un objeto Label, y que va a componer posteriormente el Panel que permitirá visualizar la fecha y la hora. Sobre el objeto Button se instancia y registra un objeto receptor de eventos de tipo ActionListener. El método sobrescrito actionPerformed() en el objeto ActionListener, hace que la fecha y la hora aparezcan en el objeto Label que está sobre la ficha, cuando se pulsa sobre el botón.

```
Label labelFecha = new Label("
");
Button botonFecha = new Button( "Fecha y Hora" );
Panel panelFecha = new Panel();
panelFecha.add( botonFecha );
panelFecha.add( labelFecha );

// Se instancia y registra un objeto ActionListener sobre el
// boton que va a presentar la fecha y la hora
botonFecha.addActionListener(
new ActionListernerFecha( labelFecha ) );
```

Las siguientes sentencias son las que permiten la creación de las fichas del panel de visualización. Primero se instancia un objeto de tipo CardLayout y se asigna a una variable de referencia, ya que no puede ser un objeto anónimo, porque se va a necesitar a la hora de procesar los eventos.

```
CardLayout miCardLayout = new CardLayout();
```

A continuación, se instancia el objeto Panel correspondiente al panel de visualización de las fichas, se especifica el controlador de posicionamiento de los Componentes y se cambia su fondo para que sea de color amarillo y se pueda distinguir fácilmente del otro objeto Panel que se usa en el interfaz.

Panel panelPresentacion = new Panel();

```
// Se fija el Cardlayout para el objeto panel
panelPresentacion.setLayout( miCardLayout );
panelPresentacion.setBackground( Color.yellow );
```

Una vez que ya existe el objeto Panel para la visualización, el siguiente paso es añadir fichas al Panel utilizando el CardLayout. En las llamadas al método add() de la clase Container de las sentencias que se reproducen, el primer parámentro es el objeto que se añade, y el segundo parámentro es el nombre del objeto. Todos los objetos que se añaden son anónimos porque son pasivos, excepto el objeto panelFecha, que se ha construido anteriormente y que no puede ser anónimo al estar compuesto por un objeto Panel, un

objeto Button y un objeto Label, y se necesita una variable de referencia del Panel para poder instanciarlo.

El nombre del objeto especificado como cadena en el segundo parámetro del método add(), se utilizará como parámentro del método show(), para indicar cual de las fichas será la que se visualice.

```
PanelP      resentacion.add(
    new Button( "La Primera ficha es un Boton" ),"primero" );
panelPresentacion.add(
    new Label( "La Segunda ficha es un Label" ),"segundo" );
panelPresentacion.add(
    new Label( "Tercera ficha, tambien un Label" ),"tercero" );
panelPresentacion.add(
    new Label( "Cuarta ficha, de nuevo un label" ),"cuarto" );
panelPresentacion.add( panelFecha,"panel fecha" );
panelPresentacion.add(
    new TextField( "La Ultima ficha es un campo de texto" ),"sexto" );
```

La siguiente tarea es ya la construcción del panel de control. Las dos sentencias que siguen son ya conocidas, y sirven para instanciar los objetos Button del panel de control y registrar objetos ActionListener para que recojan los eventos que se produzcan sobre ellos.

```
Button botonSiguiente = new Button( "Siguiente" );
. . .
botonPrimero.addActionListener(
    new ListenerPrimero( miCardLayout,panelPresentacion ) );
```

Las dos líneas de código mostradas a continuación, también resultarán ya conocidas, y aquí se utilizan para instanciar el objeto del panel de control y colocar los cinco botones sobre él.

```
Panel panelControl = new Panel();
panelControl.add( botonPrimero );
```

Ahora se colocan los dos paneles sobre el Frame que constituye la parte principal del interfaz de usuario utilizando un BorderLayout y colocándolos en posiciones superior e inferior.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
// Le incorporamos el panel de visualizacion y el panel de control
// para crear un objeto mas complejo
miFrame.add( panelPresentacion,"North" );
miFrame.add( panelControl,"South" );
```

En este momento ya está creada la clase correspondiente al interfaz de usuario y lo que resta es definir las clases que van a recibir los eventos, para lo cual se instancian y registran objetos receptores que contienen el método actionPerformed() sobrescrito. La primera clase ActionListener es la que se usa para dar servicio a los eventos provocados por la pulsación sobre el obobjeto Button colocado sobre la ficha que presenta la fecha y la hora. Esta clase contiene una variable de instancia, un constructor y el método actionPerformed() sobrescrito. La parte más interesante es el código del método sobrescrito actionPerformed(), tal como muestra el siguiente trozo de código:

```
public void actionPerformed( ActionEvent evt ) {
    miObjLabel.setText( new Date().toString() );
    }
```

Como se puede observar, este método instancia un nuevo objeto de la clase Date, que cada vez que es instanciado contiene información que puede utilizarse para obtener la fecha y hora en que fue instanciado. Esta información se extrae utilizando el método

toString() y luego, se emplea el método setText() de la clase Label para depositar la fecha y hora en el objeto Label.

Esta clase es seguida de cinco clases ActionListener muy semejantes entre ellas, que se utilizan para cambiar la ficha que se presenta en pantalla. Cuando cualquiera de los botones del panel de control es pulsado. Las definiciones de las clases difieren únicamente en el método que se llama dentro del método sobrescrito actionPerformed().

El resto del código no es de especial interés, y ya es suficiente (esperemos), con los comentarios situados en el código fuente del ejemplo, para que el lector tome posición correcta ante el entendimiento de ese código.

AWT Posicionamiento Absoluto

Los Componentes se pueden colocar en Contenedores utilizando cualquiera de los controladores de posicionamiento, o utilizando posicionamiento absoluto para realizar esta función. La primera forma de colocar los Componentes se considera más segura porque automáticamente serán compensadas las diferencias que se puedan encontrar entre resoluciones de pantalla de plataformas distintas.

La clase Component proporciona métodos para especificar la posición y tamaño de un Componente en coordenadas absolutas indicadas en pixels:

setBounds(int,int,int,int);
setBounds(Rectangle);

La posición y tamaño si se especifica en coordenadas absolutas, puede hacer más difícil la consecución de una apariencia uniforme, en diferentes plataformas, del interfaz, según algunos autores; pero, a pesar de ello, es interesante saber cómo se hace.

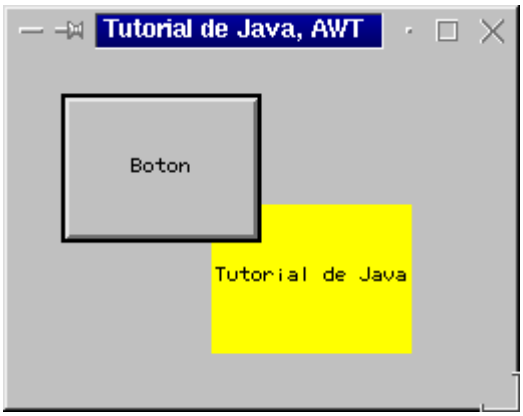
La siguiente aplicación, java1328.java, coloca un objeto Button y un objeto Label sobre un objeto Frame, utilizando coordenadas absolutas en pixels. El programa está diseñado para ser lo más simple posible y no contiene ningún controlador de eventos, por lo que el botón de cerrar la ventana colocado sobre el Frame no es funcional, y hay que concluir a las bravas la aplicación.

El objeto Button y el objeto Label, de color amarillo, se colocan sobre un objeto Frame, de tal forma que la posición y tamaño indicados para los Componentes hacen que los dos se superpongan. Este es uno de los potenciales problemas que se producen con el uso de coordenadas absolutas; aunque, por supuesto, a menos sobre una determinada plataforma, con un cuidadoso diseño. Si se cambia el tamaño del Frame, el botón y la etiqueta permanecen en el mismo tamaño y posición. El Frame puede ser cambiado de tamaño hasta el punto de no poder ver los dos Componentes, y esto no es agradable para ningún usuario.

```
import java.awt.*;
import java.awt.event.*;

public class java1328 {
    public static void main( String args[] ) {
        // Instancia un objeto de tipo Interfaz Hombre-Maquina
        IHM ihm = new IHM();
        }
    }
```

```
// La siguiente clase se utiliza para instanciar un objeto de tipo
// Interfaz Grafica de Usuario
class IHM {
    public IHM() {
        // Se crea un objeto Button con el texto que se pasa como
        // parametro y el tamaño y posicion indicadas dentro de
        // su contenedor (en pixels)
        Button miBoton = new Button( "Boton" );
        // Al rectagulo se le pasan los parametros: x,y,ancho,alto
        miBoton.setBounds( new Rectangle( 25,20,100,75 ) );

        // Se crea un objeto Label con el texto que se indique como
        // parametro en la llamada y el tamaño especificado y en la
        // posicion que se indique dentro de su contenedor (en pixels)
        // Se pone en amarillo para que destaque
        Label miEtiqueta = new Label( "Tutorial de Java" );
        miEtiqueta.setBounds( new Rectangle( 100,75,100,75 ) );
        miEtiqueta.setBackground( Color.yellow );

        // Se crea un objeto Frame con el titulo que se indica en la
        // lamada y sin ningun layout
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        miFrame.setLayout( null );

        // Añade los dos componentes al Frame, fijando su tamaño en
        // pixels y lo hace visible
        miFrame.add( miBoton );
        miFrame.add( miEtiqueta );
        miFrame.setSize( 250,175 );
        miFrame.setVisible( true );
        }
    }
```

El resultado en pantalla de la ejecución del programa se muestra en la imagen que se reproduce a continuación.



Acto seguido se entra un poco más a fondo en el programa anterior, para aclarar las partes más interesantes del código. Por ejemplo, las siguientes dos sentencias:

```
Button miBoton = new Button( "Boton" );
// Al rectágulo se le pasan los parámetros: x,y,ancho,alto
miBoton.setBounds( new Rectangle( 25,20,100,75 ) );
```

lo que hacen en realidad es crear un objeto Button con el rótulo "Boton", indicar que su posición en el eje X es 25 y en el eje Y es 50, medidas en pixels desde la esquina superior izquierda del Contenedor del objeto, teniendo en cuenta que el eje Y crece hacia abajo. El punto de referencia del botón es su esquina superior-izquierda; finalmente, especifica una anchura de 100 pixels y una anchura de 75 pixels para el botón.

En este punto del programa, una vez ejecutadas estas dos sentencias, todavía no se ha identificado el objeto que va a contener el Botón, por lo tanto, setBountds() proporciona información de las coordenadas absolutas donde se colocará el botón en cualquiera que sea el Contenedor que lo contenga. El método setBounds() es un método de la clase Component y por lo tanto, heredado en todas las subclases de Component, incluida la clase Button. El método indica que el Componente tendrá un tamaño y estará posicionado de acuerdo a una caja o rectángulo, cuyos lados son paralelos a los ejes X e Y.

Hay dos versiones sobrecargadas de setBounds(). Una de ellas permite que tamaño y posición se indiquen a través de cuatro enteros: ordenadas, abscisas, ancho y alto. La otra versión requiere que se utilice un objeto Rectangle, que se pasa como parámetro al método. Esta última versión es más flexible, porque la clase Rectangle tiene siete constructores diferentes que pueden ser utilizados para crear la caja que indicará la posición y tamaño del componente.

Las siguientes líneas de código crean un objeto Label con fondo amarillo y, de nuevo, se utiliza el método setBounds() para posicionarlo:

```
Label miEtiqueta = new Label( "Tutorial de Java" );
miEtiqueta.setBounds( new Rectangle( 100,75,100,75 ) );
miEtiqueta.setBackground( Color.yellow );
```

El método setBackground() también es un método de la clase Component heredado por todas sus subclases. Este método requiere un parámetro de tipo Color, que se puede crear de varias formas, pero la más simple es referirse a una de las constantes que están definidas en la clase Color, utilizando sintaxis del tipo:

```
public final static Color yellow;
```

Hay que recordar que a las variables declaradas como estáticas en una clase se puede acceder utilizando el nombre de la clase y el nombre de la variable. Haciendo las variables de tipo final, lo que se consiguen son en realidad, constantes.

Hay unos treinta colores diferentes definidos de este modo, si se necesita un color distinto, no incluido en esta lista, se puede utilizar uno de los constructores sobrecargados que permiten especificar un color en función de la cantidad de rojo, verde y azul que intervienen en su composición.

Las siguientes líneas de código:

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.setLayout( null );
```

crean un objeto Frame con un título. Este objeto Frame, o marco, es el tipo de objeto que los usuarios de Windows consideran como una típica Ventana. Se puede cambiar de tamaño y tiene cuatro botones: un botón de control o de menú de ventana en la parte superior izquierda y, en la zona superior-derecha, dispone de tres botones para minimizar, maximizar y cerrar la ventana.

En este ejemplo no se acepta el layout de tipo BorderLayout, que por defecto proporciona el sistema, sino que se especifican posición y tamaño de los Componentes en coordenadas absolutas. Por ello, será necesario sobreescribir la especificación del layout con null.

Este parámetro se le pasa al método setLayout() de la clase Container de la que Frame es una subclase, y que espera como parámetro un objeto de la clase LayoutManager, o null, para indicar que no se va a utilizar ningún manejador de composición, como en este caso.

Las dos líneas que aparecen a continuación:

```
miFrame.add( miBoton );
miFrame.add( miEtiqueta );
```

hacen que los dos componentes que previamente se habían definido entren a formar parte de la composición visual utilizando para ello el método add() de la clase Container. Este método tiene varias versiones sobrecargadas, y aquí se utiliza la que requiere un objeto de tipo Component como parámetro; y como Button y Label son subclases de la clase Component, están perfectamente cualificados para poder ser parámetros del método add().

Una vez que se ha llegado a este punto, ya solamente falta fijar el tamaño que va a tener el marco en la pantalla y hacerlo visible, cosa que hacen las dos líneas de código que se reproducen a continuación:

```
miFrame.setSize( 250,175 );
miFrame.setVisible( true );
```

Layouts III

GridLayout

La composición GridLayout proporciona gran flexibilidad para situar Componentes. El controlador de posicionamiento se crea con un determinado número de filas y columnas y los Componentes van dentro de las celdas de la tabla así definida.

Si el Contenedor es alterado en su tamaño en tiempo de ejecución, el sistema intentará mantener el mismo número de filas y columnas dentro de los márgenes de separación que se hayan indicado. En este caso, estos márgenes tienen prioridad sobre el tamaño mínimo que se haya indicado para los Componentes, por lo que puede llegar a conseguirse que sean de un tamaño tan pequeño que sus etiquetas sean ilegibles.

En el ejemplo java1325.java, se muestra el uso de este controlador de posicionamiento de Componentes. Además, el programa tiene un cierto incremento de complejidad respecto a los que se han visto hasta ahora para mostrar layouts, porque en este caso se crea un objeto para el interfaz de usuario que está compuesto a su vez de más objetos de nivel inferior. Y también, se ilustra el proceso de modificar dinámicamente, en tiempo de ejecución, un layout.

El interfaz de usuario está constituido por un objeto Frame sobre el que se colocan objetos Panel utilizando el controlador de posicionamiento por defecto para el Frame, BorderLayout. Uno de los objetos Panel contiene seis objetos Button posicionados utilizando un GridLayout, y no tienen ninguna funcionalidad asignada, no se registra sobre ellos ningún receptor de eventos. Inicialmente, los botones se colocan en una tabla de dos filas y tres columnas.

El otro objeto Panel contiene dos botones con los rótulos 3x2 y 2x3, que se colocan utilizando un FlowLayout. Además, estos botones sí son funcionales, ya que se registran sobre ellos objetos ActionListener. Cuando se pulsa sobre el botón 3x2, los botones del otro Panel se reposicionan en tres filas y dos columnas. Y, de forma semejante, cuando se pulsa sobre el botón 2x3, los objetos Button del otro panel se recolocan en dos filas y tres columnas.

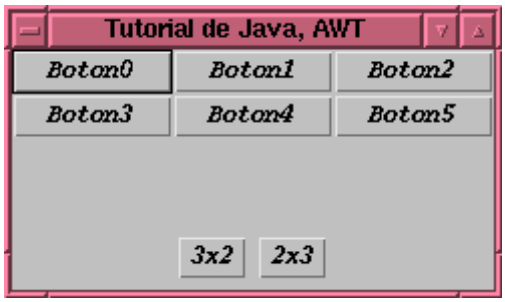
Se instancia y registra, también, un receptor de eventos de cierre de ventana sobre el objeto Frame, para terminar el programa cuando se pulse sobre el botón de cierre del Frame.

```
import java.awt.*;
import java.awt.event.*;

public class java1325 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        // Se instancian los dos botones que van a proporcionar la
        // funcionalidad a la aplicacion
        Button boton7 = new Button( "3x2" );
        Button boton8 = new Button( "2x3" );

        // Se instancia un objeto layout de tipo GridLayout para ser
        // utilizado con el Panel
        GridLayout miGridLayout = new GridLayout( 2,3 );

        // Instancia el primero de los dos objetos Panel que sera
        // integrado en el objeto Frame
        Panel panel1 = new Panel();
        // Fijamos el layout que habiamos definido para el panel
        panel1.setLayout( miGridLayout );
        // Se colocan los seis botones sobre el panel con una
        // etiqueta indicando su numero
        for( int i=0; i < 6; i++)
            panel1.add( new Button( "Boton"+i ) );

        // Se instancia el segundo objeto Panel utilizando el FlowLayout
        // por defecto y se colocan los dos botones funcionales sobre el.
        // A estos botones se les añadira su funcionalidad a traves de
        // objetos receptores de los eventos ActionListener registrados
        // sobre ellos
        Panel panel2 = new Panel();
        panel2.add( boton7 );
        panel2.add( boton8 );

        // Se instancia el objeto Frame, que sera el padre de todo
        // el interfaz de usuario que se esta creando
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );

        // IMPORTANTE: Se añaden los dos objetos Panel que se han
        // preparado al objeto Frame para crear el interfaz definitivo
        miFrame.add( panel1,"North" );
        miFrame.add( panel2,"South" );

        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );

        // Se instancian los objetos Receptores de eventos Action y se
        // registran con los botones 7 y 8, que corresponden al
        // segundo Panel y que van a modificar el posicionamiento de
        // los otrs seis botones en el Panel contiguo
        boton7.addActionListener(
            new A3x2ActionListener( miGridLayout,miFrame ) );
        boton8.addActionListener(
            new A2x3ActionListener( miGridLayout,miFrame ) );
```

```
        // Se termina de dar funcionalidad al interfaz, instanciando y
        // registrando un objeto receptor de eventos de la ventana para
        //  concluir  la  ejecucion  de  la  aplicacion  cuando  el  usuario
cierre
        // el Frame
        miFrame.addWindowListener( new Conclusion() );
        }
    }

// Las siguientes dos clases son las clases de los ActionListener,
// los receptores de eventos de tipo Action. Un objeto de cada una
// de ellas se instancia y registra sobore cada uno de los dos
// botones funcionales de la aplicacion. El proposito de estos
// controladores de eventos es modificar el layout del panel
// contiguo, de forma que los botones que estan colocados sobre
// el se cambien de posicion
class A3x2ActionListener implements ActionListener {
    GridLayout miObjGridLayout;
    Frame miObjFrame;

    A3x2ActionListener( GridLayout layout,Frame frame ) {
        miObjGridLayout = layout;
        miObjFrame = frame;
        }

    // Cuando se produce un evento Action, se fijan las filas a 3 y
    // la columnas a 2 sobre el objeto GridLayout. Luego se fija el
    // controlador de posicionamiento para que sea el nuevo que
    // acabamos de modificar, y posteriormente se valida el Frame
    // para asegurarse de que el alyout es valido y tendra efecto
    // sobre la visualizacion en pantalla
    public void actionPerformed( ActionEvent evt ) {
        miObjGridLayout.setRows( 3 );
        miObjGridLayout.setColumns( 2 );
        miObjFrame.setLayout( miObjGridLayout );
        miObjFrame.validate();
        }
    }

class A2x3ActionListener implements ActionListener {
    GridLayout miObjGridLayout;
    Frame miObjFrame;

    A2x3ActionListener( GridLayout layout,Frame frame ) {
        miObjGridLayout = layout;
        miObjFrame = frame;
        }

    // Cuando se produce un evento Action, se fijan las filas a 2 y
    // la columnas a 3 sobre el objeto GridLayout. Luego se fija el
    // controlador de posicionamiento para que sea el nuevo que
    // acabamos de modificar, y posteriormente se valida el Frame
    // para asegurarse de que el alyout es valido y tendra efecto
    // sobre la visualizacion en pantalla
    public void actionPerformed( ActionEvent evt ) {
        miObjGridLayout.setRows( 2 );
        miObjGridLayout.setColumns( 3 );
        miObjFrame.setLayout( miObjGridLayout );
        miObjFrame.validate();
        }
    }

class Conclusion extends WindowAdapter {
```

```
    public void windowClosing( WindowEvent evt ) {
        // Termina la ejecucion del programa cuando se cierra la
        // ventana principal de la aplicacion
        System.exit( 0 );
        }
}
```

Si se compila y ejecuta la aplicación, inicialmente aparecerá en pantalla una ventana semejante a la que se reproduce en la imagen siguiente:



A continuación se comentan algunas de las sentencias de código más interesantes del programa. Para comenzar, se instancian dos objetos Button que posteriormente serán los botones funcionales del programa. Estos objetos no pueden ser instanciados anónimamente, precisamente porque sobre ellos se va a registrar un objeto de tipo ActionListener, y es necesario que pueda tener una variable de referencia sobre cada objeto para permitir ese registro.

```
Button boton7 = new Button( "3x2" );
Button boton8 = new Button( "2x3" );
```

La siguiente sentencia instancia un objeto GridLayout que será utilizado como controlador de posicionamiento para uno de los objetos Panel. Tampoco este objeto puede ser anónimo, porque se necesitará acceso a él a la hora de modificar la colocación de los Componentes sobre ese Panel.

```
GridLayout miGridLayout = new GridLayout( 2,3 );
```

En el siguiente trozo de código se instancia uno de los objetos Panel que se integrarán en el objeto Frame principal. Se usará el objeto GridLayout, instanciado antes, como controlador de posicionamiento en el panel. Y, a continuación, se usa un bucle para colocar los seis objetos Button en el Panel. En este caso, sí es posible hacer que los objetos Button sean anónimos, ya que no se va a registrar sobre ellos ningún tipo de receptor de eventos.

```
Panel panel1 = new Panel();
// Fijamos el layout que habiamos definido para el panel
panel1.setLayout( miGridLayout );
// Se colocan los seis botones sobre el panel con una
// etiqueta indicando su numero
for( int i=0; i < 6; i++)
    panel1.add( new Button( "Boton"+i ) );
```

Ahora se instancia el segundo objeto Panel, y se colocan en él los dos botones que se habían instanciado antes, que son los que van a tener funcionalidad. Esto se hace en las siguientes sentencias:

```
Panel panel2 = new Panel();
panel2.add( boton7 );
panel2.add( boton8 );
```

El próximo paso es instanciar un objeto Frame que servirá como interfaz. Una vez que se instancie, se colocarán en él los dos objetos Panel utilizando su controlador de posicionamiento por defecto, el BorderLayout.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
// IMPORTANTE: Se añaden los dos objetos Panel que se han
// preparado al objeto Frame para crear el interfaz definitivo
miFrame.add( panel1,"North" );
miFrame.add( panel2,"South" );
```

En este instante, la creación física del interfaz está concluida, así que el siguiente paso es instanciar y registrar un objeto anónimo receptor de eventos de tipo Action, para procesar los eventos de los dos objetos Button que van a permitir cambiar la apariencia de los botones.

```
boton7.addActionListener( new A3x2ActionListener( miGridLayout,miFrame ) );
boton8.addActionListener( new A2x3ActionListener( miGridLayout,miFrame ) );
```

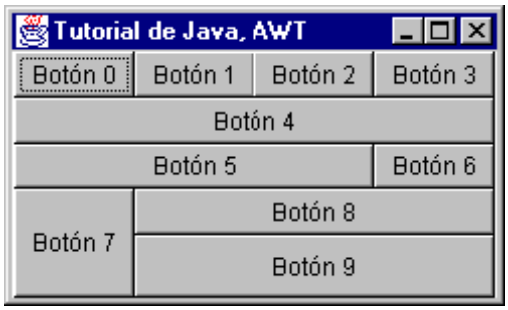
El código del controlador de eventos, tal como se puede ver en las sentencias que se reproducen a continuación, hace uso de los métodos de la clase del Layout para ejecutar las acciones pertinentes, a la recepción de un evento. En este caso, un evento provocado por uno de los objetos Button hace que los Componentes se coloquen en una malla de 3 filas por 2 columnas; y un evento sobre el otro objeto Button hace que se recoloquen en dos filas y tres columnas. El código del otro controlador es semejante.

```
public void actionPerformed( ActionEvent evt ) {
    miObjGridLayout.setRows( 3 );
    miObjGridLayout.setColumns( 2 );
    miObjFrame.setLayout( miObjGridLayout );
    miObjFrame.validate();
    }
```

GridBagLayout

Es igual que la composición de GridLayout, con la diferencia que los Componentes no necesitan tener el mismo tamaño. Es quizá el controlador de posicionamiento más sofisticado de los que actualmente soporta AWT.

A la hora de ponerse a trabajar con este controlador de posicionamiento, hay que tomar el rol de un auténtico aventurero. Parece que la filosofía de la gente de JavaSoft es que todo debe hacerse en el código. La verdad es que hasta que no haya en Java algo semejante a los recursos de X, el trabajo del programador, si quiere prescindir de herramientas de diseño, es un tanto prehistórico en su forma de hacer las cosas.



Si el lector acepta una recomendación, el consejo es que evite como la peste el uso del GridBagLayout, porque tanta sofisticación lo único que acarrea son dolores

de cabeza; y, siempre se puede recurrir a la técnica de combinar varios paneles utilizando otros controladores de posicionamiento, dentro del mismo programa. Loa applets no apreciarán esta diferencia, al menos no tanto como para justificar los problemas que conlleva el uso del GridBagLayout.

A pesar de los pesares, se ha implementado como muetra el ejemplo java1326.java, un pograma que presenta diez botones en pantalla, con la apariencia que muestra la figura anterior.

```
import java.awt.*;

public class java1326 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
        }
    }

class IHM {
    public IHM() {
        Frame miFrame = new Frame( "Tutorial de Java, AWT" );
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();

        miFrame.setLayout( gridbag );

        // Para este grupo fijamos la anchura y vamos variando alguna
        // de las caracteristicas en los siguientes, de tal forma que
        // los botones que aparecen en cada una de las lineas tengan
        // apariencia diferente en pantalla
        gbc.fill = GridBagConstraints.BOTH;
        gbc.weightx = 1.0;
        Button boton0 = new Button( "Botón 0" );
        gridbag.setConstraints( boton0,gbc );
        miFrame.add( boton0 );
        Button boton1 = new Button( "Botón 1" );
        gridbag.setConstraints( boton1,gbc );
        miFrame.add( boton1 );
        Button boton2 = new Button( "Botón 2" );
        gridbag.setConstraints( boton2,gbc );
        miFrame.add( boton2 );

        gbc.gridwidth = GridBagConstraints.REMAINDER;
        Button boton3 = new Button( "Botón 3" );
        gridbag.setConstraints( boton3,gbc );
        miFrame.add( boton3 );

        gbc.weightx = 0.0;
        Button boton4 = new Button( "Botón 4" );
        gridbag.setConstraints( boton4,gbc );
        miFrame.add( boton4 );

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        Button boton5 = new Button( "Botón 5" );
        gridbag.setConstraints( boton5,gbc );
        miFrame.add( boton5 );

        gbc.gridwidth = GridBagConstraints.REMAINDER;
        Button boton6 = new Button( "Botón 6" );
        gridbag.setConstraints( boton6,gbc );
        miFrame.add( boton6 );

        gbc.gridwidth = 1;
        gbc.gridheight = 2;
```

```
        gbc.weighty = 1.0;
        Button boton7 = new Button( "Botón 7" );
        gridbag.setConstraints( boton7,gbc );
        miFrame.add( boton7 );

        gbc.weighty = 0.0;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.gridheight = 1;
        Button boton8 = new Button( "Botón 8" );
        gridbag.setConstraints( boton8,gbc );
        miFrame.add( boton8 );
        Button boton9 = new Button( "Botón 9" );
        gridbag.setConstraints( boton9,gbc );
        miFrame.add( boton9 );
        miFrame.setSize( 250,150 );
        miFrame.setVisible( true );
        }
}
```

Para aprovechar de verdad todas las posibilidades que ofrece este layout, hay que pintar antes en papel como van a estar posicionados los Componentes; utilizar gridx , gridy, gridwidth y gridheight en vez de GridBagConstraints.RELATIVE, porque en el proceso de validación del layout pueden quedar todos los Componentes en posición indeseable. Además, se deberían crear métodos de conveniencia para hacer más fácil el posicionamiento de los Componentes. En resumen, que las complicaciones que genera el uso del GridBagLayout no compensan.

Pero hay una luz en el horizonte, porque Swing incorpora un controlador de posicionamiento nuevo que utiliza la técnica que se ha popularizado con Smalltalk "Springs and Struts", lo cual redudirá significativamente la necesidad de tener que recurrir al GridBagLayout.

No obstante, y por no dejar cojo en algún sentido al Tutorial, y también por si el lector, a pesar de las advertencias, quiere intentar la aventura de implementar su aplicación basándose en este controlador de posicionamiento, a continuación se tratan una serie de cuestiones para intentar explicar los parámetros y características que se pueden configurar dentro del GridBagLayout, y ya que sea el propio lector el que intente probar suerte con él.

Este controlador de posicionamiento coloca los Componentes en filas y columnas, pero permite especificar una serie de parámetros adicionales para ajustar mejor la posición y tamaño de los Componentes dentro de las celdas. Y, al contrario que ocurría con el GridLayout, las filas y columnas no tienen porque tener un tamaño uniforme.

El controlador utiliza en número de Componentes en la fila más larga, el número de filas totales y el tamaño de los Componentes, para determinar el número y tamaño de las celdas que va a colocar en la tabla. La forma de visualizarse este conjunto de celdas, puede determinarse a través de una serie de características recogidas en un objeto de tipo GridBagConstraints. Estas características, o propiedades, son las que se van a describir a continuación. El objeto GridBagConstraints se inicializa a unos valores de defecto, cada uno de los cuales puede ser ajustado para alterar la forma en que se presenta el Componentes dentro del layout.

El ejemplo java1332.java, es el que se va a utilizar para comprobar el efecto que causan las modificaciones en los parámetros del objeto GridBagConstraints. Más concretamente, se van a modificar las características del objeto a la hora de manipula el posicionamiento de los dos botones de la parte inferior del panel.

```
import java.awt.*;
```

```
import java.awt.event.*;

public class java1332 extends Frame {
    Panel panel;

    public java1332() {
        // Estos son los botones que se van a marear
        Button botAceptar,botCancelar;

        // Este es el panel que contiene a todos los componentes
        panel = new Panel();
        panel.setBackground( Color.white );
        add( panel );

        // Se crean los objetos del GridBag y se le asigna este
        // layout al panel
        GridBagConstraints gbc = new GridBagConstraints();
        GridBagLayout gridbag = new GridBagLayout();
        panel.setLayout( gridbag );

        // Se indica que los componentes pueden rellenar la zona
        // visible en cualquiera de las dos direcciones, vertical
        // u horizontal
        gbc.fill = GridBagConstraints.BOTH;
        // Se redimensionan las columnas y se mantiene su relación
        // de aspecto isgual en todo el proceso
        gbc.weightx = 1.0;

        // Se crea la etiqueta que va a servir de título al
        // panel
        Label labTitulo = new Label( "GridBag Layout" );
        labTitulo.setAlignment( Label.CENTER );
        // Se hace que el componente Label sea el único que se
        // sitúe en la línea que lo contiene
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        // Se pasan al layout tanto el componente Label, como
        // el objeto GridBagConstraints que modifica su
        // posicionamiento
        gridbag.setConstraints( labTitulo,gbc );
        // Finalmente se añade la etiqueta al panel. El objeto
        // GridBagConstraints de este contenedor pone la etiqueta
        // en una línea, la redimensiona para que ocupe toda la
        // fila de la tabla, tanto horizontal como verticalmente,
        // y hace que las columnas se redimensionen de la misma
        // forma cuando la ventana cambie de tamaño
        panel.add( labTitulo );

        // Ahora se crea uno de los campos de texto, en este
        // caso para recoger un supuesto nombre
        TextField txtNombre = new TextField( "Nombre:",25 );
        // Se hace que el campo de texto sea el siguiente objeto
        // después del último que haya en la fila. Esto significa
        // que todavía se puede añadir uno o más componentes a
        // la fila, a continuación del campo de texto
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        // Se pasan al layout tanto el campo de texto, como
        // el objeto GridBagConstraints que modifica su
        // posicionamiento
        gridbag.setConstraints( txtNombre,gbc );
        // Se añade el campo de texto al panel
        panel.add( txtNombre );

        // Se crea otro campo de texto, en este caso para recoger
```

```
// la direccion del propietario del nombre anterior
TextField txtDireccion = new TextField( "Direccion:",25 );
// Se hace que este campo de texto sea el último
// componente que se sitúe en la fila en que se
// encuentre
gbc.gridwidth = GridBagConstraints.REMAINDER;
// Se pasan al layout tanto el campo de texto, como
// el objeto GridBagConstraints que modifica su
// posicionamiento
gridbag.setConstraints( txtDireccion,gbc );
// Se añade el campo de texto al panel
panel.add( txtDireccion );

// Se crea un área de texto para introducir las cosas
// que quiera el que esté utilizando el programa
TextArea txtComent = new TextArea( 3,25 );
txtComent.setEditable( true );
txtComent.setText( "Comentarios:" );
// Se pasan al layout tanto el área de texto, como
// el objeto GridBagConstraints que modifica su
// posicionamiento
gridbag.setConstraints( txtComent,gbc );
// Se añade el área de texto al panel
panel.add( txtComent );

// Estos son los dos botones de la parte inferior del
// panel, sobre los que vamos a modificar las
// propiedades del objeto GridBagConstraints que
// controla su posicionamiento dentro del panel, para
// ir mostrando el comportamiento del conjunto ante
// esos cambios
botAceptar = new Button( "Aceptar" );
botCancelar = new Button( "Cancelar" );

// Hacemos que el botón "Aceptar" no sea el último
// de la fila y que no pueda expandirse en ninguna
// dirección
gbc.fill = GridBagConstraints.NONE;
gbc.gridwidth = GridBagConstraints.RELATIVE;
// Se pasan al layout el botón y el objeto
// GridBagConstraints
gridbag.setConstraints( botAceptar,gbc );
// Se añade el botón al panel
panel.add( botAceptar );

// Se hace que el botón "Cancelar" sea el último de
// la fila en que se encuentre
gbc.gridwidth = GridBagConstraints.RELATIVE;
// Se hace que su altura no se reescale
gbc.gridheight = 1;
// Se pasan al layout el botón y el objeto
// GridBagConstraints
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );

// Se añade el receptor de eventos de la ventana
// para acabar la ejecución
addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
        }
    } );
}
```

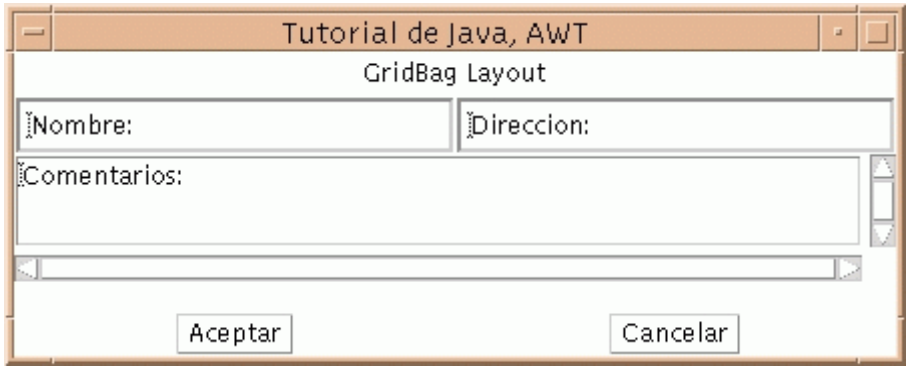
```
    public static void main( String args[] ) {
        java1332 miFrame = new java1332();

        // Fijamos el título de la ventana y la hacemos
        // visible, con los componentes en su interior
        miFrame.setTitle( "Tutorial de Java, AWT" );
        miFrame.pack();
        miFrame.setVisible( true );
        }
}
```

Tanto el código del ejemplo como los comentarios que se han insertado, deberían proporcionar al lector una idea de lo que hace cada una de las características del objeto GridBagConstraints y de cómo se usa. A continuación se describen estos parámetros, modificándolos sobre el ejemplo anterior.

gridx y gridy

Las propiedades gridx y gridy indican la fila y la columna, respectivamente, en donde se va a colocar el Componente. La primera fila de la parte superior es gridx=0, y la columna más a la izquierda corresponde a gridy=0.

Los valores para estos parámetros deben ser un entero correspondiendo a una casilla fila/columna explícita o el valor GridBagConstraints.RELATIVE, que indica que el Componente debe ser colocado en una fila/columna relativa a la fila/columna donde se ha colocado el último Componente.



Por ejemplo, cuando se añaden componentes a una fila, gridx=GridBagConstraints.RELATIVE, coloca el Componente a la derecha del último Componente que se haya colocado en la fila. Si un Componente se ha insertado en la fila 2 y columna 1, el siguiente Componente se insertará en la fila 2 y columna 2.

Cuando se añaden componente a una columna, gridy=GridBagConstraints.RELATIVE coloca el Componente por debajo del último Componente que se haya añadido a la columna. Si un Componente se ha insertado en la columna 0 y fila 1, el siguiente Componente se insertará en la columna 0 y fila 2.

Si se incorpora el siguiente trozo de código al ejemplo java1332.java, para fijar nuevos valores de gridx y gridy:

```
gbc.gridx = 1;
gbc.gridy = 3
gbc.fill = GridBagConstraints.NONE;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gridbag.setConstraints( botAceptar,gbc );
```

```
// Se añade el botón al panel
panel.add( botAceptar );

gbc.gridx = 0;
gbc.gridy = 3
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );
```

se observa que el resultado de la ejecución ahora es el mismo, pero los botones Aceptar y Cancelar están cambiados de sitio, tal como reproduce la figura siguiente:

Se puede fijar el valor de gridx y gridy para evitar la colocación secuencial de componentes que realiza el sistema por defecto. En este contexto, ni GridBagConstraints.NONE ni tampoco GridBagConstraints.REMAINDER tienen significado alguno.

gridwidth y gridheight

Los parámetros o propiedades gridwidth y gridheight indican el número de celdas en la zona de presentación que va a ocupar un determinado Componente. Los valores por defecto son una fila de ancho y una columna de alto, es decir, gridwidth=1 y gridheight=1.

Se puede conseguir que un Componente ocupe dos columnas de ancho indicando gridwith=2 o tres filas de alto con gridheight=3. Si se fija gridwidth a GridBagConstraints.REMAINDER, se indica que el Componente debería ocupar el número de celdas que queden libres en la fila. Fijándolo a GridBagConstraints.RELATIVE se indica que el Componente debería ocupar todas las celdas que queden en la fila, excepto la última.

Cuando se indica que un Componente es el último o el siguiente al último Componente de la fila, se puede proporcionar un valor de gridheight más grande que la unidad y el Componente se expandirá por varias filas. Sin embargo, en este caso, no hay forma de hacer que el Componente sea más ancho que una columna, cuando sea el último o el siguiente al último de la fila.

Si se añade el siguiente trozo de código al ejemplo principal java1332.java:

```
gbc.weigthy = 1.0;
gbc.fill = GridBagConstraints.VERTICAL;
gbc.gridheight = 2;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gridbag.setConstraints( botAceptar,gbc );
// Se añade el botón al panel
```

```
panel.add( botAceptar );

gbc.fill = GridBagConstraints.NONE;
gbc.gridheight = 1;
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );
```

El resultado que se obtiene es el que se muestra en la figura que aparece a continuación, que es la imagen capturada de la ventana:

En ella se puede observar que el botón Aceptar es dos veces más alto que el botón Cancelar, tal como se ha fijado en el código.

weightx y weighty

Cuando el usuario redimensiona una ventana que utiliza el GridBagLayout como controlador de posicionamiento de los componentes, los parámetros weightx y weighty determinan la forma en que se van a redimensionar. Por defecto, los valores de estos parámetros son weightx=0 y weighty=0, lo que significa que cuando la ventana es redimensionada, los componentes permanecen juntos y agrupados en el centro del contenedor.

Si se proporciona un valor mayor que 0 para weightx, los componentes se expandirán en la dirección x, horizontalmente. Si se proporciona un valor mayor que 0 para weighty, los componentes se expandirán en la dirección y, verticalmente.

En las dos modificaciones anteriores que se han hecho al ejemplo java1332.java, se puede comprobar el efecto de estos parámetros. En la primera modificación, donde se alteraban los valores de gridx y gridy, en el código, c.weigthx=1.0 y c.weigthy=0.0, permiten que los componentes se redimensionen junto con la ventana, horizontalmente, pero permanecen agrupados en el centro, verticalmente.

En la segunda modificación, correspondiente a gridwidth y gridheight, el valor de c.weighty se ha cambiado de 0.0 a 1.0, utilizando un valor para gridheight de 2 y un valor de fill fijado a GridBagConstraints.VERTICAL; esto hace que el botón aceptar ocupe dos filas, pero se expande solamente en dirección vertical.

fill

El parámetro fill determina la forma en que un Componente rellena el área definida por gridx/gridy/gridwidth/gridheight; y los valores que puede tomar este parámetro pueden ser:

GridBagConstraints.HORIZONTAL: El Componente se expande horizontalmente para rellenar todo el área de visualización.

GridBagConstraints.VERTICAL: El Componente se expande verticalmente para rellenar todo el área de visualización.

GridBagConstraints.BOTH: El Componente se expande completamente para ocupar la totalidad del área de visualización.

GridBagConstraints.NONE: El Componente es reducido a su tamaño ideal, independientemente del tamaño que tenga la zona de visualización.

En la modificación que se hizo del código del ejemplo java1332.java para mostrar el efecto de la modificación de los parámetros gridwidth y gridheight, el valor de c.fill se cambia de GridBagConstraints.BOTH a GridBagConstraints.VERTICAL y se utiliza con un valor de weighty fijado a 1 y un valor de 2 para gridheight, así que el botón Aceptar se expande a dos filas, pero siempre llena completamente la zona verticalmente.

Si ahora se añade el siguiente trozo de código al ejemplo java1332.java, se provocará el relleno completo de la zona de visualización en dirección horizontal.

```
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gridbag.setConstraints( botAceptar,gbc );
// Se añade el botón al panel
panel.add( botAceptar );

gbc.gridheight = 1;
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );
```

Como se puede observar en la figura siguiente, en la que se reproduce el efecto causado por el código anterior, los botones también ocupan completamente la fila, porque hay dos botones y dos columnas. El área de texto también ocupa la fila completamente porque el parámetro fill está fijado a GridBagConstraints.BOTH y no hay ningún otro Componente en la segunda columna de la fila.

anchor

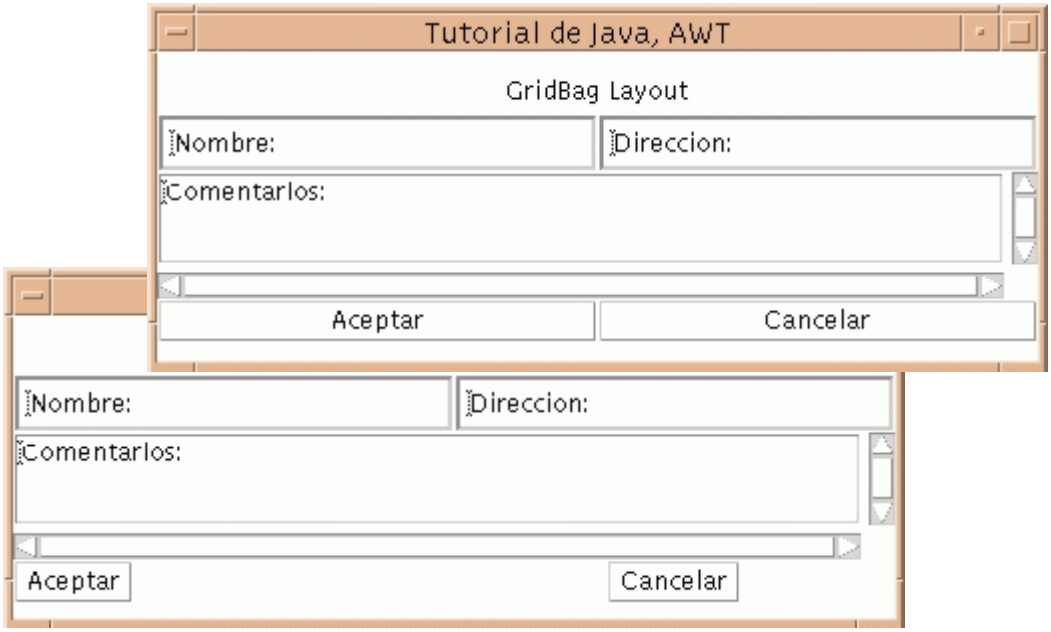
Cuando un Componente es más pequeño que la zona de visualización, se puede colocar en una determinada posición utilizando el parámetro anchor que puede tomar el valor GridBagConstrints.CENTER, que es el que toma por defecto, o cualquiera de las direcciones de los puntos cardinales: NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST.

Si se añade el siguiente trozo de código al ejemplo java1332.java, que ya debe estar mareado del todo, el botón Aceptar se desplazará hacia la parte oeste de la fila, mientras que el botón Cancelar sigue en el centro de su celda.

```
gbc.fill = GridBagConstraints.NONE;
gbc.anchor = GridBagConstraints.WEST;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gridbag.setConstraints( botAceptar,gbc );
// Se añade el botón al panel
```

```
panel.add( botAceptar );

gbc.anchor = GridBagConstraints.CENTER;
gbc.gridheight = 1;
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );
```

La imagen siguiente reproduce la situación generada por la modificación anterior, en la que se pueden observar las posiciones de los botones tal como se refiere.



A pesar de ajustar este parámetro, hay que asegurarse de haber fijado el valor de fill adecuadamente, porque si el Componente rellena la zona de visualización en la dirección equivocada, en anclaje del Componente puede quedar sin efecto. Por ejemplo, si se tiene un Componente que se expande por dos filas y tiene un fill vertical, el anclaje del Componente en el Norte, Centro o Sur no alterará en nada su apariencia. Si el fill es solamente vertical y no está fijado el fill horizontal, el anclaje en cualquiera otra de las posiciones, que no sean las tres anteriores, sí que cambiará la posición del Componente sobre el área de visualización.

ipadx e ipady

El controlador de posicionamiento calcula el tamaño de un Componente basándose en los parámetros del GridBagLayout y en los otros componentes que se encuentren sobre el layout. Se puede indicar un desplazamiento interno para incrementar el tamaño calculado para un Componente y hacer que ese Componente sea más ancho, ipadx, o más alto, ipady, que el tamaño real que calcularía el controlador, aunque no llegue a llenar completamente la zona de visualización en dirección horizontal o vertical, al estilo que hace el parámetro fill.

Los valores de defecto de ipadx e ipady son 0. Cuando se especifica un valor para ipadx, el ancho total del Componente se calcula sumando a su ancho real, dos veces la cantidad que se haya indicado en ipadx, ya que el desplazamiento se añade por ambos lados del Componente. Cuando se especifica un valor para ipady, la altura total del

Componente se calcula sumando a su altura real, dos veces la cantidad que se haya indicado en ipady, ya que el desplazamiento se añade por arriba y por abajo del Componente.

Si se incorpora al ejemplo java1332.java, el trozo de código que se reproduce a continuación, se obtendrá como resultado una imagen semejante a las anteriores, pero los botones serán 25 píxels más anchos por cada lado.

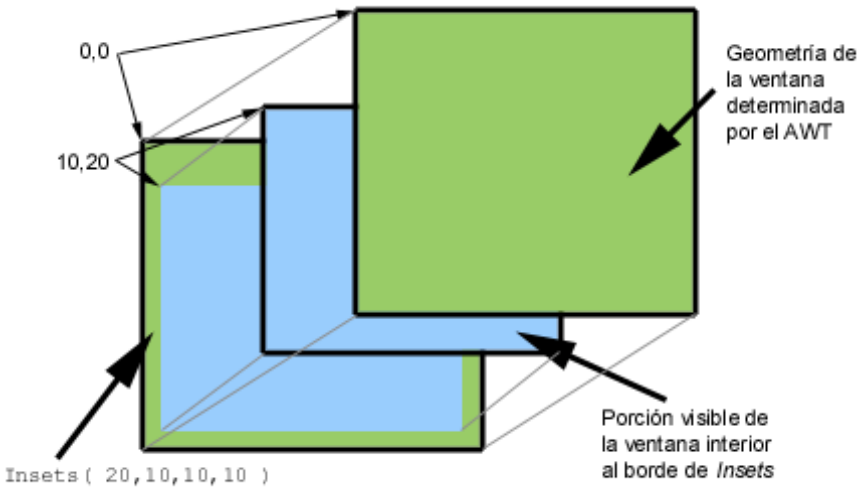
```
gbc.fill = GridBagConstraints.NONE;
gbc.ipadx = 50;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gridbag.setConstraints( botAceptar,gbc );
// Se añade el botón al panel
panel.add( botAceptar );

gbc.gridheight = 1;
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );
```

La imagen siguiente muestra el efecto del código anterior, observándose que efectivamente los botones son más anchos que los de los ejemplos anteriores, y no llegan a ocupar toda la fila como en el caso de fill.

insets

El parámetro insets permite especificar la mínima cantidad de espacio que debe haber entre el Componente y los bordes del área de visualización. Por defecto, el espacio determinado por insets se deja en blanco, pero también se puede rellenar con un color, o una imagen o un título.

La imagen siguiente muestra el gráfico explicativo, que seguro es más comprensible que lo escrito.



Como valor de este parámetro hay que pasarle un objeto de tipo Insets, al cual se le especifica el espacio que se quiere dejar en cada uno de los cuatro lados del Componente.

Si se añade el siguiente trozo del código al ejemplo java1332.java, se ampliará en 15 pixels el espacio que hay entre la fila en que se encuentran los botones Aceptar y Cancelar y el área de texto que está por encima.

```
gbc.insets = new Insets( 15,0,0,0 );
gbc.fill = GridBagConstraints.NONE;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gridbag.setConstraints( botAceptar,gbc );
// Se añade el botón al panel
panel.add( botAceptar );

gbc.gridheight = 1;
gridbag.setConstraints( botCancelar,gbc );
// Se añade el botón al panel
panel.add( botCancelar );
```

La imagen siguiente reproduce el efecto generado por la modificación anterior, en donde se puede observar el desplazamiento comentado.

Como se puede observar, el GridBagLayout es el más poderoso y flexible de los controladores de posicionamiento que proporciona el AWT. Sin embargo, el llegar a dominar los parámetros que lo pueden modificar no es nada sencillo y lo más habitual es que se generen efectos visuales indeseados a la hora de redimensionar la ventana, incluso prestando suficiente atención en la asignación de los valores de los parámetros. Por ello, el autor recomienda al lector que si puede utilizar cualquier otro controlador de posicionamiento, o incluso una combinación de ellos, que evite el uso de GridBagLayout, porque su dominio no es sencillo y, además, nunca se está seguro de cómo se va a comportar.

Layouts IV

BoxLayout

El controlador de posicionamiento BoxLayout es uno de los dos que incorpora Java a través de Swing. Permite colocar los Componentes a lo largo del eje X o del eje Y, y también posibilita que los Componentes ocupen diferente espacio a lo largo del eje principal.

En un controlador BoxLayout sobre el eje Y, los Componentes se posicionan de arriba hacia abajo en el orden en que se han añadido. Al contrario en en el caso del

GridLayout, aquí se permite que los Componentes sean de diferente tamaño a lo largo del eje Y, que es el esje principal del controlador de posicionamiento, en este caso.

En el eje que no es principal, BoxLayout intenta que todos los Componentes seal tan anchos como el más ancho, o tan alto como el más alto, dependiendo de cuál sea el eje principal. Si un Componente no puede incrementar su tamaño, el BoxLayout mira las propiedades de alineamiento en X e Y para determinar donde colocarlo.

OverlayLayout

El controlador de posicionamiento OverlayLayout, es el otro que se incopora a Java con Swing, y es un poco diferente a todos los demás. Se dimensiona para contener el más grande de los Componentes y superpone cada Componente sobre los otros.

La clase OverlayLayout no tiene un constructor por defecto, así que hay que crearlo dinámicamente en tiempo de ejecución.

LayoutAbsoluto

A continuación se desarrolla un controlador de posicionamiento de componentes propio, para que el lector compruebe que no es nada complicada la creación de layouts y que Java da esa posibilidad. En este caso se trata de un controlador de posicionamiento absoluto, es decir, es un controlador que posiciona un componente en una determinada posición indicada en coordenadas x e y, y de un anchura y una altura también especificadas, bien de forma explícita o por defecto para todos los componentes.

Interfaz LayoutManagerAbsoluto

La forma de implementar nuevos controladores de posicionamiento es a través del interfaz LayoutManager. A partir del JDK 1.1, Javasoft introdujo un nuevo interfaz, LayoutManager2, destinado a la implementación de controladores de posicionamiento basados en parámetros, en constraints, de forma que se utilicen objetos de este tipo para controlar el tamaño y posición de los componentes sobre el contenedor. No obstante, la documentación advierte que "no se proporciona soporte total, completo para los controladores de posicionamiento propios basados en constraints". Así que, para evitar esos problemas y hasta que haya soporte completo, se ha creado el interfaz LayoutManagerAbsoluto, , que añade los métodos abstractos setConstraints() y getConstraints() para poder fijar y recuperar estos parámetros.

```
public interface LayoutManagerAbsoluto extends LayoutManager,Cloneable {
    public abstract Object getConstraints( Component comp );
    public abstract void setConstraints( Component comp,Object cons );
    public abstract Object clone();
    }
```

Selección de Parámetros

Antes de poner a trabajar al nuevo controlador de posicionamiento, hay que comenzar definiendo los valores de los parámetros o constraints que se van a asignar a cada uno de los componentes que se quieren situar en el contenedor controlado por el layout recién creado. En la clase LayoutAbsolutoConstraints, , como se puede ver el el

código de la clase que se muestra, los constraints son un modelo simple de objetos que deberían ser clonables.

```
public class LayoutAbsolutoConstraints implements Cloneable {
    // Los nombres de los parámetros son semejantes a los que se utilizan
    // en el objeto de tipo GridBagConstraints
    public static final int NONE = 0;
    public static final int BOTH = 1;
    public static final int HORIZONTAL = 2;
    public static final int VERTICAL = 3;

    public static final int NORTH = 1;
    public static final int NORTHEAST = 2;
    public static final int EAST = 3;
    public static final int SOUTHEAST = 4;
    public static final int SOUTH = 5;
    public static final int SOUTHWEST = 6;
    public static final int WEST = 7;
    public static final int NORTHWEST = 8;

    public int x;
    public int y;
    public int width;
    public int height;
    public int anchor;
    public int fill;
    public Insets insets;

    // Los valores -1 en width y height significan que se ha de
    // utilizar el valor preferido para el Componente de que se
    // trate
    public LayoutAbsolutoConstraints() {
        x = 0;
        y = 0;
        width = -1;
        height = -1;
        anchor = NONE;
        fill = NONE;
        insets = new Insets( 0,0,0,0 );
        }

    public Object clone() {
        LayoutAbsolutoConstraints lCon = new LayoutAbsolutoConstraints();
        lCon.x = x;
        lCon.y = y;
        lCon.width = width;
        lCon.height = height;
        lCon.anchor = anchor;
        lCon.fill = fill;
        lCon.insets = (Insets)insets.clone();
        return( lCon );
        }
    }
```

Lo que se quiere es que cada Componente esté emplazado en una determinada posición y tenga un tamaño específico, además de algunas capacidades básicas al estilo del GridBagLayout, como pueden ser la posibilidad de anclaje o relleno. En resumen, la información que se quiere parametrizar a través de los constraints, tal como se muestra en la clase, es la siguiente:

- La posición del Componente, o coordenadas X e Y, que se almacenan en las variables x e y de LayoutAbsolutoConstraints

- El tamaño del Componente, almacenado en las variables width, para el ancho, y height para el alto; con la posibilidad de indicar -1 en el caso de que lo que se quiera es que se utilice el tamaño preferido del Componente

- Se trate o no de un Componente, debería estar anclado en una determinada posición respecto a los puntos cardinales del contenedor, como puede ser norte, este o sudeste. Esta información se guarda en la variable de instancia anchor, y los valores que puede tomar son constantes estáticas predefinidas para cada uno de los puntos cardinales y derivados, de la forma NORTH, EAST, etc.

- También se debe poder especificar si se ha de llenar completamente el contenedor bien a lo ancho o bien a lo alto. Esta información se guarda en la variable de instancia fill, que puede tomar cualquiera de los tres valores predeterminados: HORIZONTAL, VERTICAL o BOTH.

- Finalmente, se debe poder indicar la separación en pixels que ha de haber entre los bordes del Componente y los límites del contenedor.

Control de Parámetros

Una vez que se tienen los parámetros, o constraints, fijados, ya se puede crear el controlador de posicionamiento que funcione en base a esos valores. La clase LayoutAbsoluto, , implementa el interfaz LayoutManagerAbsoluto y redefine los métodos que se definen en el interfaz y también los que se definen en el interfaz LayoutManager, que implementa el interfaz LayoutManagerAbsoluto, que se van a utilizar en la clase, como son layoutContainer(), minimumLayoutSize() y preferredLayoutSize().

```
import java.awt.*;
import java.beans.*;
import java.util.*;

public class LayoutAbsoluto implements LayoutManagerAbsoluto {
    private static final int PREFERRED = 0;
    private static final int MINIMUM =   1;

    private Hashtable consTable = new Hashtable();
    private LayoutAbsolutoConstraints
    defaultConstraints = new LayoutAbsolutoConstraints();
    public int width;
    public int height;

    // Si se indica -1 en los parámetros width o height, se da a entender
    // que se desea utilizar el tamaño preferido del componente
    public LayoutAbsoluto(){
        width = -1;
        height = -1;
        }

    // Aquí se especifica el ancho y la altura que se desea para el
    // componente, una vez que se vaya a presentar en el contenedor
    public LayoutAbsoluto( int w,int h ) {
        width = w;
        height = h;
        }

    public void addLayoutComponent( String name,Component comp ) {
        // No se utiliza. El usuario debe llamar a setConstraints()
        // y después utilizar el método add(comp) genérico, al igual
```

```
        // que se hace en el caso del GridBagLayout
        }

    // Se usa para obtener los valores que actualmente tienen
    // asignados los parámetros que controlan el posicionamiento
    // de un determinado componente sobre la zola de control del
    // layout
    public Object getConstraints( Component comp ) {
        return( lookupConstraints(comp).clone() );
        }

    // Aquí es donde se controlan todos los parámetros que se
    // permiten, los constraints, para poder modificar la posición
    // y tamaño del componente, y para indicarle al contenedor las
    // características especiales de posicionamiento que se
    // quieren aplicar al componente
    public void layoutContainer( Container parent ) {
        Component comp;
        Component comps[];
        LayoutAbsolutoConstraints cons;
        Dimension d;
        Dimension pD;
        int x;
        int y;

        comps = parent.getComponents();
        Insets insets = parent.getInsets();
        pD = parent.getSize();
        for( int i=0; i < comps.length; i++ ) {
            comp = comps[i];
            // Se obtienen los parámetros actuales, que en principio
            // serán los de defecto, más los que se hayan modificado
            // en el constructor
            cons = lookupConstraints( comp );
            // Se aplican los desplazamientos "insets" a la posición
            x = cons.x + cons.insets.left + insets.left;
            y = cons.y + cons.insets.top + insets.top;
            // Se obtiene el tamaño preferido para el componente
            d = comp.getPreferredSize();
            // Si no se quiere que el componente se presente con su
            // tamaño preferido, se modifica su anchura y su altura
            // con los valores que se indiquen
            if( cons.width != -1 )
                d.width = cons.width;
            if( cons.height != -1 )
                d.height = cons.height;
            // Ahora se controla el parámetro "fill" de forma que
            // el componente ocupe todo el espacio que se le
            // indique, y en la dirección que se le diga, en caso
            // de que se quiera cambiar. Observar que en este caso
            // también hay que tener en cuenta los "insets" o
            // separaciones de los bordes que se hayan indicado,
            // porque esa es una zona que hay que respetar
            if( (cons.fill == LayoutAbsolutoConstraints.BOTH) ||
                (cons.fill == LayoutAbsolutoConstraints.HORIZONTAL) ) {
                x = insets.left + cons.insets.left;
                d.width = pD.width - cons.insets.left -
                    cons.insets.right - insets.left - insets.right;
                }
            if( (cons.fill == LayoutAbsolutoConstraints.BOTH) ||
                (cons.fill == LayoutAbsolutoConstraints.VERTICAL) ){
                y = insets.top + cons.insets.top;
                d.height = pD.height - cons.insets.top -
                    cons.insets.bottom - insets.top - insets.bottom;
                }
```

```
            // A continuación se controla el parámetro "anchor", para
            // anclar el componente en alguna de las posiciones que
            // están permitidas
            switch( cons.anchor ){
                case LayoutAbsolutoConstraints.NORTH:
                    x = ( pD.width - d.width ) / 2;
                    y = cons.insets.top + insets.top;
                    break;
                case LayoutAbsolutoConstraints.NORTHEAST:
                    x  =  pD.width  -  d.width  -  cons.insets.right  -
insets.right;
                    y = cons.insets.top + insets.top;
                    break;
                case LayoutAbsolutoConstraints.EAST:
                    x  =  pD.width  -  d.width  -  cons.insets.right  -
insets.right;
                    y = ( pD.height - d.height ) / 2;
                    break;
                case LayoutAbsolutoConstraints.SOUTHEAST:
                    x  =  pD.width  -  d.width  -  cons.insets.right  -
insets.right;
                    y = pD.height - d.height - cons.insets.bottom -
                        insets.bottom;
                    break;
                case LayoutAbsolutoConstraints.SOUTH:
                    x = ( pD.width - d.width ) / 2;
                    y = pD.height - d.height - cons.insets.bottom -
                        insets.bottom;
                    break;
                case LayoutAbsolutoConstraints.SOUTHWEST:
                    x = cons.insets.left + insets.left;
                    y = pD.height - d.height - cons.insets.bottom -
                        insets.bottom;
                    break;
                case LayoutAbsolutoConstraints.WEST:
                    x = cons.insets.left + insets.left;
                    y = ( pD.height - d.height ) / 2;
                    break;
                case LayoutAbsolutoConstraints.NORTHWEST:
                    x = cons.insets.left + insets.left;
                    y = cons.insets.top + insets.top;
                    break;
                default:
                    break;
                }
            // Y, finalmente, se fija la posción y dimensión del
            // componente, una vez tenidos en cuenta todos los
            // parámentros que permite modificar su posicionamiento
            comp.setBounds( x,y,d.width,d.height );
            }
        }

     // Devuelve el tamaño que ocupan todos los componentes en el
    // contenedor, es decir, devuelve el tamaño que debe tener el
    // contenedor para poder visulaizar todos los componentes, en
    // función de su tamaño y posición en el layout
    private Dimension layoutSize( Container parent,int tipo ) {
        int ancho;
        int alto;

        // En caso de que no se indique que el layout debe
        // considerar el tamaño preferido, hay que ir calculando
        // las posiciones y tamaños de los componentes que se van
        // a posicionar en su interior para saber el tamaño
        // mínimo que ha de tener para contenerlos a todos
```

```
        if( (width == -1) || (height == -1) ){
            Component comp;
            Component comps[];
            LayoutAbsolutoConstraints cons;
            Dimension d;
            int x;
            int y;

            ancho = alto = 0;
            comps = parent.getComponents();
            for( int i=0; i < comps.length; i++ ) {
                comp = comps[i];
                cons = lookupConstraints( comp );
                if( tipo == PREFERRED )
                    d = comp.getPreferredSize();
                else
                    d = comp.getMinimumSize();
                if( cons.width != -1 )
                    d.width = cons.width;
                if( cons.height != -1 )
                    d.height = cons.height;
                if( cons.anchor == LayoutAbsolutoConstraints.NONE ) {
                    x = cons.x;
                    y = cons.y;
                    }
                else {
                    x = cons.insets.left;
                    y = cons.insets.top;
                    }
                if( (cons.fill != LayoutAbsolutoConstraints.BOTH) &&
                    (cons.fill != LayoutAbsolutoConstraints.HORIZONTAL) )
                    ancho = Math.max( ancho,x + d.width );
                else
                    ancho = Math.max( ancho,d.width + cons.insets.left +
                        cons.insets.right );
                if( (cons.fill != LayoutAbsolutoConstraints.BOTH) &&
                    (cons.fill != LayoutAbsolutoConstraints.VERTICAL) )
                    alto = Math.max( alto,y + d.height );
                else
                    alto = Math.max( alto,d.height + cons.insets.top +
                        cons.insets.bottom );
                }
            if( width != -1 )
                ancho = width;
            if( height != -1 )
                alto = height;
            }
        else {
            ancho = width;
            alto = height;
        }

        // Una vez que se sabe el tamaño necesario para contener
        // a todos los componentes o se ha indicado un tamaño fijo,
        // se aplican los desplazamientos desde los bordes, para
        // devolver el tamaño definitivo que debe tener el
        // contenedor
        Insets insets = parent.getInsets();
        return( new Dimension( ancho + insets.left + insets.right,
            alto + insets.top + insets.bottom ) );
        }

    // Devuelve los valores de los parámetros que se han indicado
    // para el componente, o en caso de que no se haya indicado
    // ninguno en especial, devuelve los valores por defecto que
```

```
    // se asginan a todos los componentes
    private LayoutAbsolutoConstraints lookupConstraints( Component comp )
{
        LayoutAbsolutoConstraints p =
            (LayoutAbsolutoConstraints)consTable.get( comp );
        if( p == null ) {
            setConstraints( comp,defaultConstraints );
            p = defaultConstraints;
            }
        return( p );
        }

    // Devuelve el mínimo tamaño que se ha especificado para el
    // contendor que está controlado por el layout
    public Dimension minimumLayoutSize( Container parent ) {
        return( layoutSize( parent,MINIMUM ) );
        }

    // Devuelve el tamaño preferido
    public Dimension preferredLayoutSize( Container parent ) {
        return( layoutSize( parent,PREFERRED ) );
        }

    // Elimina un componente del Layout, haciendo que desaparezca
    // también de la visualización en pantalla o donde se esté
    // mapenado el contenedor
    public void removeLayoutComponent( Component comp ) {
        consTable.remove( comp );
        }

    // Aquí se aplican los valores que contiene el objeto de
    // tipo Constraints al componente que se desea controlar
    // en posición y tamaño sobre el contenedor que está siendo
    // manejado por el layout que se ha creado
    public void setConstraints( Component comp,Object cons ) {
        if( (cons == null) || (cons instanceof LayoutAbsolutoConstraints)
){
            LayoutAbsolutoConstraints pCons;
            // Si no se indica ningún objeto que contenga los valores de
            // posicionamiento, se aplican los valores de defecto
            if( cons == null )
                pCons                                                    =
(LayoutAbsolutoConstraints)defaultConstraints.clone();
            else
                pCons = (LayoutAbsolutoConstraints)
                    ( (LayoutAbsolutoConstraints)cons ).clone();
            // Una vez fijados los valores de los parámetros, se incluye
            // el componente en la lista de componentes que están siendo
            // manejados por el layout manager
            consTable.put( comp,pCons );
            // Lo siguiente es necesario para el caso en qhe se aniden
            // layout del tipo absoluto que estamos creando. Cuando los
            //constraints del componente están destinados a ser elásticos
            // o no-eláticos, es cuando se comprueba para ver si el
            // componente es en sí mismo un contenedor, con otro layout
            // absoluto como controlador de posicionamiento de los
            // componentes y, si es así, fijar el layout para que sea
            // elástico o no-elástico, según sea necesario
            if( Beans.isInstanceOf( comp,Container.class ) ) {
                if( ( (Container)Beans.getInstanceOf( comp,
                    Container.class ) ).getLayout()
                      instanceof LayoutAbsoluto ) {
                    LayoutAbsoluto layout;
                    layout = (LayoutAbsoluto)
                        ( (Container)Beans.getInstanceOf( comp,
```

```
                        Container.class ) ).getLayout();
                    layout.width = pCons.width;
                    layout.height = pCons.height;
                    }
                }
            }
        }

    // Devuelve un objeto igual, pero ya con el ancho y alto determinado
    // pos los valores de los parámetros correspondientes, sean los de
    // defecto o los que se indiquen específicamente
    public Object clone() {
        LayoutAbsoluto p = new LayoutAbsoluto( width,height );
        return( p );
        }
    }
```

Por repasar las partes más interesantes del código, lo primero que cabe destacar son los métodos minimumLayoutSize() y preferredLayoutSize(), que llaman al mismo método layoutSize(). Este hace un recorrido por todos los componentes que hay en el contenedor , comprobando los valores de los parámetros y manteniendo el total de anchura y altura que debe tener el contenedor para permitir la visualización de todos los componentes. Tiene en cuenta todas las posibilidades, desde que los valores de width y height puedan ser -1 para utilizar los valores preferidos para el Componente en cuestión, hasta la cantidad de pixels que se determinen para separar el Componente del borde del contenedor.

El siguiente trozo de código interesante corresponde a la implementación de los métodos getConstraints() y setConstraints(). Los objetos que almacenan los valores de los parámetros se guardan en una Hastable, de forma que en el método setConstraints() se fijan los pares clave:valor en la tabla. En caso de que no se especifique ningún valor válido para los parámetros, se utilizan los valores de defecto y se tiene cuidado especial en asegurarse de que los valores de los parámetros que se introducen en la tabla son únicos, y no se introducen también los de los clones.

```
public Object getConstraints( Component comp ) {
    return( lookupConstraints(comp).clone() );
    }
.
.
.
public void setConstraints( Component comp,Object cons ) {
    if( (cons == null) || (cons instanceof LayoutAbsolutoConstraints) ){
        LayoutAbsolutoConstraints pCons;
        // Si no se indica ningún objeto que contenga los valores de
        // posicionamiento, se aplican los valores de defecto
        if( cons == null )
            pCons                                                    =
(LayoutAbsolutoConstraints)defaultConstraints.clone();
        else
            pCons = (LayoutAbsolutoConstraints)
                ( (LayoutAbsolutoConstraints)cons ).clone();
        // Una vez fijados los valores de los parámetros, se incluye
        // el componente en la lista de componentes que están siendo
        // manejados por el layout manager
        consTable.put( comp,pCons );
        // Lo siguiente es necesario para el caso en qhe se aniden
        // layout del tipo absoluto que estamos creando. Cuando los
        // constraints del componente están destinados a ser elásticos
        // o no-eláticos, es cuando se comprueba para ver si el
        // componente es en sí mismo un contenedor, con otro layout
        // absoluto como controlador de posicionamiento de los
```

```
        // componentes y, si es así, fijar el layout para que sea
        // elástico o no-elástico, según sea necesario
        if( Beans.isInstanceOf( comp,Container.class ) ) {
            if( ( (Container)Beans.getInstanceOf( comp,
                Container.class ) ).getLayout()
                  instanceof LayoutAbsoluto ) {
                LayoutAbsoluto layout;
                layout = (LayoutAbsoluto)
                    ( (Container)Beans.getInstanceOf( comp,
                      Container.class ) ).getLayout();
                layout.width = pCons.width;
                layout.height = pCons.height;
                }
            }
        }
    }
```

Otro trozo de código interesante corresponde al método lookupConstraints(), que como se puede observar, es utilizado por el método getConstraints(), y también es común a layoutSize() y layoutContainer(). Este método, que se reproduce a continuación, simplemente recorre la tabla en donde se encuentran los parámetros y los devuelve.

```
private LayoutAbsolutoConstraints lookupConstraints( Component comp ) {
    LayoutAbsolutoConstraints p =
        (LayoutAbsolutoConstraints)consTable.get( comp );
    if( p == null ) {
        setConstraints( comp,defaultConstraints );
        p = defaultConstraints;
        }
    return( p );
    }
```

De nuevo, si no existe ningún constraint para un Compopnente, se utilizan los de defecto. Esto es ahora así porque Javasoft no proporciona soporte completo para los controladores de posicionamiento creados por usuarios, seguro que cuando esto se proporcione, el mecanismo de la hashtable estará encapsulado y no habrá que manejarlo del modo que aquí se hace.

Al igual que los constraints para los Componentes que están manejados por el controlador de posicionamiento, este mismo controlador dispone de sus propios parámetros de ancho y alto. Si sus valores se fijan a -1, esto significa que el contenedor será elástico, es decir, que se expandirá o encogerá para adaptarse al tamaño preferido de todos los componentes que contiene en su interior. Si se proporciona un ancho y un alto determinados, el contenedor padre siempre tendrá el mismo tamaño, independientemente del tamaño y posición de los componentes que contenga.

El último trozo de código que merece la pena resaltar es precisamente, el fragmento que define el control del caso en que se fije o indique un tamaño preferido de un componente que es a la ver un contenedor y tiene su propia posición. En este caso, se asegura que el tamaño preferido del layout anidado coincide con su correspondiente contenedor.

```
if( Beans.isInstanceOf( comp,Container.class ) ) {
    if( ( (Container)Beans.getInstanceOf( comp,
        Container.class ) ).getLayout()
          instanceof LayoutAbsoluto ) {
        LayoutAbsoluto layout;
        layout = (LayoutAbsoluto)
            ( (Container)Beans.getInstanceOf( comp,
              Container.class ) ).getLayout();
```

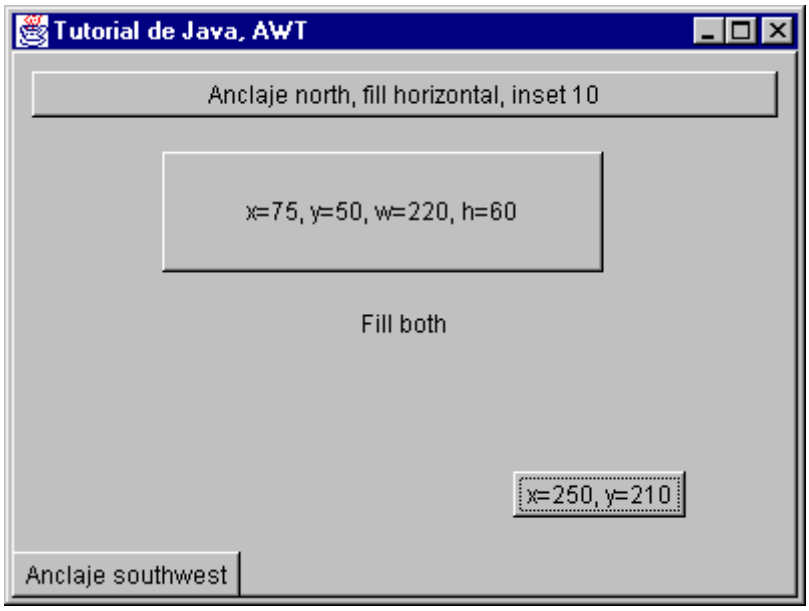
```
        layout.width = pCons.width;
        layout.height = pCons.height;
        }
    }
```

Uso del LayoutAbsoluto

Ahora ya está completo el nuevo controlador de posicionamiento y está listo para ser utilizado. En el ejemplo , se hace uso de él, creando una serie de botones que se colocan en posiciones determinadas y con tamaños específicos, utilizando el LayoutAbsoluto. Así, se pueden colocar los componentes de la forma tradicional de los compiladores antiguos, especificando sus coordenadas X e Y.

```
import java.awt.*;
import java.awt.event.*;

public class java1333 extends Frame {
    public java1333() {
        super();
        setSize( 400,300 );
        setTitle( "Tutorial de Java, AWT" );

        // Se crea y se fija un layout Absoluto
        LayoutAbsoluto posLayout = new LayoutAbsoluto();
        setLayout(posLayout);

        // Se crean varios botones con distintos valores para los
        // parámetros que los controlan sobre el layout
        LayoutAbsolutoConstraints pCons;

        pCons = new LayoutAbsolutoConstraints();
        pCons.x = 250;
        pCons.y = 210;
        Button botonTamanoPreferido = new Button( "x=250, y=210" );
        posLayout.setConstraints( botonTamanoPreferido,pCons );
        add( botonTamanoPreferido );

        pCons = new LayoutAbsolutoConstraints();
        pCons.x = 75;
        pCons.y = 50;
        pCons.width = 220;
        pCons.height = 60;
        Button botonTamanoFijo = new Button("x=75, y=50, w=220, h=60");
        posLayout.setConstraints( botonTamanoFijo,pCons );
        add( botonTamanoFijo );

        pCons = new LayoutAbsolutoConstraints();
        pCons.anchor = LayoutAbsolutoConstraints.SOUTHWEST;
        Button botonAnclado = new Button( "Anclaje southwest" );
        posLayout.setConstraints( botonAnclado,pCons );
        add( botonAnclado );

        pCons = new LayoutAbsolutoConstraints();
        pCons.anchor = LayoutAbsolutoConstraints.NORTH;
        pCons.fill = LayoutAbsolutoConstraints.HORIZONTAL;
        pCons.insets = new Insets(10, 10, 10, 10);
        Button botonInsets =
            new Button( "Anclaje north, fill horizontal, inset 10" );
        posLayout.setConstraints( botonInsets,pCons );
        add( botonInsets );

        pCons = new LayoutAbsolutoConstraints();
        pCons.fill = LayoutAbsolutoConstraints.BOTH;
        Button botonFill = new Button( "Fill both" );
```

```
        posLayout.setConstraints( botonFill,pCons );
        add( botonFill );
```




```
                                                                    //
Esta es una clase anidada anónima que se utiliza para
        // concluir la ejecución del programa cuando el usuario
        // decide cerrar el Frame
        addWindowListener( new WindowAdapter() {
            public void windowClosing( WindowEvent evt ) {
                System.exit( 0 );
                }
            } );
        }

    public static void main( String args[] ) {
        // De una tacada se crea el objeto y se hace visible
        ( new java1333() ).setVisible( true );
        }
    }
```

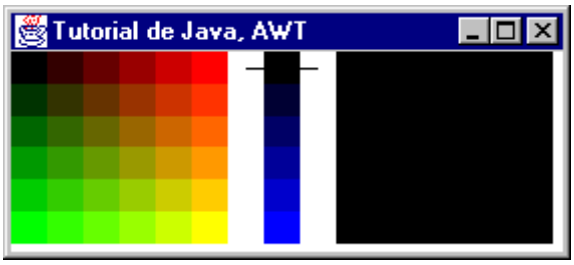
La apariencia que toman estos botones en la ventana es la que reproduce la figura siguiente, en donde se observa que aparece como título de cada uno de los botones, los valores de los parámetros o constraints que se han modificado y que hacen que aparezca de esa forma y en esa posición el botón correspondiente.

AWT - Creación de Componentes Propios

Para cubrir todos los conceptos vertidos sobre el AWT, tanto sobre los Componentes como sobre el nuevo modelo de eventos instaurado desde el JDK 1.1; y como fin de fiesta (por el momento), se va a crear un nuevo Componente de cosecha propia. Para mantener las cosas claras y no entrar en demasiadas profundidades, el ejemplo java1329.java creará un componente simple, un selector de color, y el fin que se persigue con el ejemplo es mostrar cómo se controlan los eventos y se incorporan Componentes al AWT, mas que entrar en detalles de la selección de color, que complicaría demasiado la explicación.

Interfaz Gráfica del Selector de Color

El Componente SelectorColor está formado por tres regiones, tal como muestra la figura al pie del párrafo. La zona izquierda presenta una paleta de colores, variando el tono de rojo de izquierda a derecha y el tono de verde de arriba a abajo. El usuario selecciona los niveles de rojo y verde pulsando con el ratón en esta paleta. En la zona media aparece una barra de desplazamiento, sobre la cual el usuario puede uindicar la cantidad de color azul colocando el selector de la barra en la posición deseada. La zona derecha es la que presenta el color seleccionado, que estará generado a partir de la combinación de rojo, verde y azul que el usuario haya elegido en los dos selectores. Pulsando con el ratón en esta zona se selecciona el color actual, generándose el evento AWT adecuado.



Implementación del Selector de Color

El Componente que se desea crear se implementa a través de cuatro clases:

ColorEvent, es la clase del evento propio que va a transportar el Color resultante de la selección final

ColorListener, es el interfaz a través del cual se podrán registrar otros Componentes para recibir eventos de color, ColorEvent

SelectorColor, es el Componente actual de selección de color

ColorEventMulticaster, es utilizada por la clase SelectorColor para mantener una lista de los receptores, ColorListener, registrados para recibir eventos de color, ColorEvent.

A continuación se describe cada una de las cuatro clases, para luego entrar en el funcionamiento en sí del selector de color.

Clase ColorEvent

Los eventos de tipo ColorEvent son eviados por el componente SelectorColor cuando el usuario pica en la zona derecha del interfaz gráfico del selector de color. El evento contiene un campo Color, que corresponde al color seleccionado y que se puede obtener a través del método getColor().

Cualquier evento que sea utilizado por un componente del AWT, debe sr subclase de AWTEvent, así que la declaración de ColorEvent será:

```
public class ColorEvent extends AWTEvent {
```

Todos los eventos AWT deben tener asignados identificadores enteros; un identificador válido y disponible para el programador es cualquiera por encima de RESERVED_ID_MAX, así que la línea de código siguiente es la que define al identificador para el evento que generará el selector de color.

public static final int COLOR_SELECCIONADO =

AWTEvent.RESERVED_ID_MAX + 1;

Como la clase de un evento ahora se utiliza como una característica distintiva y no solamente como un identificativo, los componentes creados por los programadores no tienen que seleccionar valores globales únicos; en lugar de eso, el identificador puede utilizarse para distinguir entre diferentes tipos de una clase de un evento determinado. Por ejemplo, se puede definir también un COLOR_ACTIVADO que indique cuando el usuario ha cambiado el color seleccionado pero todavía no ha confirmado la selección. Así, con los dos identificadores, se podría distinguir entre los dos eventos con una sola clase ColorEvent, en lugar de utilizar dos clases separadas.

El color asociado al evento se almacena en la variable color.

protected Color color;

El constructor del evento acepta como parámetro el origen del evento, origen, que en este caso será un SelectorColor y el color que se ha picado. También se declaran un método que permite al receptor del evento saber qué color se ha seleccionado, getColor(). El código correspondiente es el que se reproduce a continuación.

```
public ColorEvent( Object obj,Color color ) {
    super( obj,COLOR_SELECCIONADO );
    this.color = color;
    }

public Color getColor() {
    return( color );
    }
```

Interfaz ColorListener

Las clases interesadas en recibir eventos de tipo ColorEvent, deben implementar el interfaz ColorListener, que declara el método colorSeleccionado() a través del cual se despachan los eventos.

Todos los interfaces deben extender el interfaz EventListener, así que hay que seguir las normas:

```
import java.util.EventListener;
public interface ColorListener extends EventListener {
```

El método colorSeleccionado() deberá ser llamado por todos los interesados cuando se seleccione un color. El parámetro evt contiene el ColorEvent que les interesa.

```
public void colorSeleccionado( ColorEvent evt );
```

Clase SelectorColor

La clase SelectorColor es un componente simple que se puede añadir al interfaz gráfico como cualquier otro componente del AWT proporcionado con el JDK y, además, utiliza el nuevo modelo de delegación de eventos. El interfaz gráfico no está muy trabajado a propósito, porque la intención es mostrar los entresijos de los eventos del modelo de delegación, que es lo más complicado del diseño de componentes, y no el diseño de

interfaces, porque esto último, al fin y a la postre, va en función del gusto del programador, así que se plantea como reto al lector el conseguir un interfaz más amigable para este Selector de Color.

Aquí se extiende la clase Canvas porque el selector de color es un Componente completamente orientado a dibujo, tal como se ha diseñado. Una orientación al lector, por si ha aceptado el reto anterior, es que utilice la clase Panel, para usar otros Componentes del AWT en la creación de un mejor interfaz gráfico para el selector de color.

public class SelectorColor extends Canvas {

El selector de color cuantifica el espacio RGB en seis niveles de cada componente del color: 0, 51, 102, 153, 204 y 255, que corresponden a la paleta de colores que utiliza Netscape. Si se desea una degradación más fina, se pueden utilizar más niveles.

protected static final int NIVELES = 6;

Los actuales niveles de rojo, verde y azul , se almacenan en las variables correspondientes. En el constructor se extraen los componentes del color inicial y luego se habilitan los eventos del ratón a través del método enableEvents(). Si no se habilitan los eventos de este modo, el Componentes sería incapaz de generar eventos de ratón.

```
public SelectorColor( Color color ) {
    r = color.getRed();
    g = color.getGreen();
    b = color.getBlue();
    enableEvents( AWTEvent.MOUSE_EVENT_MASK );
    }
```

Una alternativa para este Componente sería la de recibir sus propios evetnos de ratón, para lo cual debería implementar el interfaz MouseListener y registrarse como un receptor de eventos de ratón sobre el Componente. Así, el constructor llamaría al otro constructor con un color por defecto:

```
public SelectorColor() {
    this( Color.black );
    }
```

Para determinar el tamaño correcto del Componente, se implementan los métodos getMinimumSize(), getMaximumSize() y getPreferredSize(). Las líneas de código siguientes, reproducen este último.

```
public Dimension getPreferredSize() {
    return( new Dimension( 150,60 ) );
    }
```

Siguiendo con la apariencia en pantalla, el método paint() es el que va a rellenar las tres partes en que se ha dividido el Componente. El trozo de código siguiente muestra cómo se hace, no siendo muy importante el entrar en detalles, porque el código resulta bastante autoexplicativo.

```
public void paint( Graphics g ) {
    int h = getSize().width / (NIVELES+3+NIVELES);
    int v = getSize().height / (NIVELES);

    for( int rojo=0; rojo < NIVELES; ++rojo) {
        for( int verde=0; verde < NIVELES; ++verde ) {
            g.setColor( new Color( rojo * 255 / (NIVELES-1),
                verde * 255 / (NIVELES-1),b ) );
            g.fillRect( rojo*h,verde*v,h,v );
            }
        }
```

```
    int x = NIVELES*h + h/2;
    int y = v / 2+v * (b*(NIVELES-1) / 255);
    g.setColor( getForeground() );
    g.drawLine( x,y,x+2*h-1,y );
    for( int azul=0; azul < NIVELES; ++azul ) {
        g.setColor( new Color( 0,0,azul*255 / (NIVELES-1) ) );
        g.fillRect( (NIVELES+1)*h,azul*v,h,v );
        }
    g.setColor( new Color( r,this.g,b) );
    g.fillRect( (NIVELES+3)*h,0,h*NIVELES,v*NIVELES );
    }
```

El método processMouseEvent() es invocado automáticamente por el método processEvent() del Componente cuando se genera un evento de ratón. Habrá que sobreescribir este método para que se llame al método mousePressed() que interesa, en respuesta a las pulsaciones del ratón, y luego ya se llamará al método de la superclase processMouseEvent() para realizar el tratamiento adecuado. Si hay otros receptores registrados para los eventos, el método de la superclase los mantendrá informados a través del interfaz MouseListener.

```
protected void processMouseEvent( MouseEvent evt ) {
    if( evt.getID() == MouseEvent.MOUSE_PRESSED ) {
        mousePressed( evt );
        }
    super.processMouseEvent( evt );
    }
```

Se llama a mousePressed() cuando el usuario pulsa sobre el Componente. Si el usuario pulsa en la zona de selección de la izquierda, se asignarán menos niveles de rojo y verde al color final, cuantificados al número de tonos seleccionado. Si se pulsa sobre la barra central, se asignará un nuevo tono de azul. Y, si se pulsa sobre la zona de la derecha, se confirmará la selección del color que esté presente, llamando al método postColorEvent(), para lanzar el evento correspondiente. El código siguiente muestra cómo se hace.

```
public void mousePressed( MouseEvent evt ) {
    int h = getSize().width / (NIVELES+3+NIVELES);
    int v = getSize().height / (NIVELES);

// En la zona izquierda de selección de color directo
if( evt.getX () < NIVELES*h ) {
    r = (evt.getX() / h) * 255 / (NIVELES-1);
    r = (r < 0) ? 0 : (r > 255) ? 255 : r;
    g = (evt.getY() / v) * 255 / (NIVELES-1);
    g = (g < 0) ? 0 : (g > 255) ? 255 : g;
    repaint();
// en la barra azul de enmedio
} else if( evt.getX() < (NIVELES+3) * h ) {
    b = (evt.getY() / v) * 255 / (NIVELES-1);
    b = (b < 0) ? 0 : (b > 255) ? 255 : b;
    repaint();
// en la zona derecha de resultado de la combinación
} else {
    postColorEvent();
    }
}
```

El método postColorEvent() crea un nuevo ColorEvent, con this como origen y el color actualmente seleccionado como parámetro adicional, y lo envía a la cola de eventos del sistema. También se pueden enviar eventos a la cola de eventos del sistema a través de llamadas al método dispatchEvent(). La cola de eventos del sistema es monitorizada por un

hilo de ejecución del AWT, EventDispatchThread, que lo que hace es extraer los objetos AWTEvent y llamar a dispatchEvent() sobre el Componente que sea el origen del evento.

En la práctica, los applets deberán utilizar siempre el método dispatchEvent() para lanzar eventos, porque el gestor de seguridad les retringe el acceso a la cola de eventos del sistema. Para una aplicación, cualquiera de los dos métodos es posible, la diferencia entre el uso de dispatchEvent() y el acceso directo a la cola de eventos es claramente sutil: Un evento insertado en la cola será despachado posteriormente por el hilo de ejecución del AWT y, en el otro caso, el evento será despachado inmediatamente. No obstante, debe utilizarse siempre que se pueda la cola de eventos del sistema, porque esto protege al llamador contra posibles excepciones de tipo RuntimeException, que pueda lanzar el receptor.

Aquí, tal como muestra el código siguiente, se indican los dos métodos. En cualquierea de ellos se llamará a dispatchEvent(), que invocará a su vez a processEvent(). Por lo tanto, habrá que sobreescribir este último para que pueda tratar adecuadamente el nuevo tipo de evento.

```
protected void postColorEvent() {
    ColorEvent evt = new ColorEvent( this,new Color( r,g,b ) );
    Toolkit toolkit = getToolkit();
    EventQueue cola = toolkit.getSystemEventQueue();
    cola.postEvent( evt );
    }

// Otra forma de hacerlo sería invocando al método dispatchEvent()
// dispatchEvent( new ColorEvent( this,new Color( r,g,b) ) );
```

Para seguir la folosofía implantada por el nuevo modelo de delegación de eventos, hay que mantener una lista de los receptores registrados para recibir eventos que generará el Componente.

Se puede utilizar un Vector para este propósito, aunque así se va a autilizar la clase ColorEventMulticaster para mantener esa lista. Por lo tanto, hay que incorporar el método addColorListener(), que añadirá un ColorListener a la lista utilizando el método add() de la clase ColorEventMulticaster. Este receptor añadido será notificado cuando el usuario seleccione un color a través del componente SelectorColor.

```
public synchronized void addColorListener( ColorListener l ) {
    receptorColor = ColorEventMulticaster.add( receptorColor,l );
    }
```

Del mismo modo, hay que proporcionar un método para poder eliminar de la lista un objeto que quiera darse de baja en la recepción de eventos procedentes del SelectorColor.

```
public synchronized void removeColorListener( ColorListener l ) {
    receptorColor = ColorEventMulticaster.remove( receptorColor,l );
    }
```

El método processEvent() es llamado por dispatchEvent() para distribuir eventos AWT generados por este Componente a cualquiera de los receptores registrados. Hay que sobreescribir el método para procesar los eventos de color: si el evento es de la clase ColorEvent, se llama al método processColorEvent(); y, en cualquier otro caso, se invocará al método processEvent() de la superclase, para tener controlados los eventos normales del AWT.

```
protected void processEvent( AWTEvent evt ) {
    if( evt instanceof ColorEvent )
        processColorEvent( (ColorEvent)evt );
```

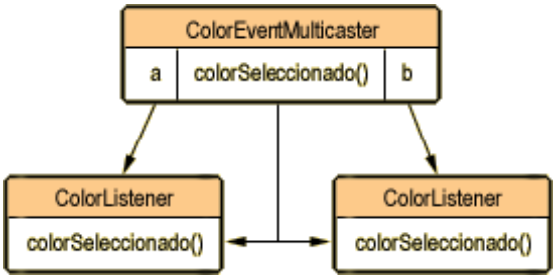
```
    else
        super.processEvent( evt );
    }
```

Ahora, el ColorEvent será distribuido a la lista de receptores registrados a través del método colorSeleccionado() del interfaz ColorListener.

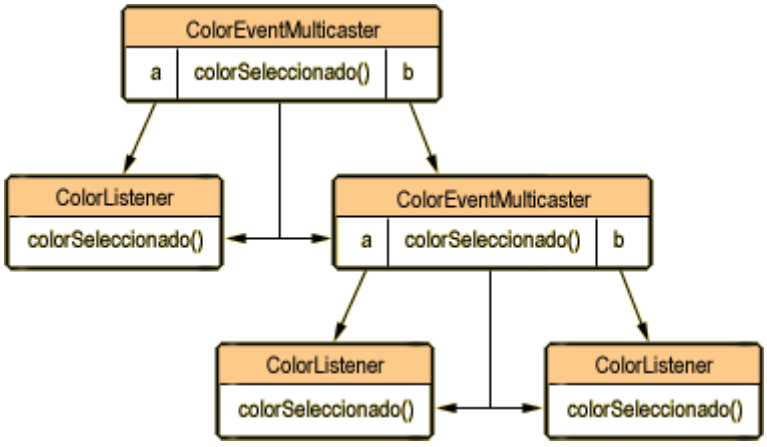
Como se ha indicado antes, también se puede utilizar un Vector de receptores que recorrerá la lista llamando a colorSeleccionado() sobre cada uno de los elemenos, y el mantenimiento de esta lista no es nada complicado.

Clase ColorEventMulticaster

La clase ColorEventMulticaster también mantiene una lista de objetos receptores de eventos del SelectorColor, ColorListener. Esta clase implementa el interfaz ColorListener y mantiene referencias a otros dos objetos ColorListener. Es la típica lista, tal como se muestra en la figura.



Cuando se invoca al método colorSeleccionado(), el evento se propaga a los dos ColorListener asociados. Si se encadenan más objetos de este tipo, se consigue una lista de receptores, tal como indica la representación gráfica de la figura



.

Se implementa el interfaz ColorListener y se mantienen referencias a otros dos objetos ColorListener, a y b:

```
class ColorEventMulticaster implements ColorListener {
    protected ColorListener a,b;
```

El constructor acepta referencias a dos objetos ColorListener. Y el método colorSeleccionado(), lo único que hará será pasar el evento a los objetos ColorListener asociados.

```
public void colorSeleccionado( ColorEvent evt ) {
    a.colorSeleccionado( evt );
    b.colorSeleccionado( evt );
    }
```

Si se quiere añadir un receptor, se usa el método estático add(). Este método devuelve un objeto ColorListener, que genera una lista con los objetos proporcionados. En el caso de que alguno de los dos objetos sea nulo, se puede devolver el otro; o se puede devolver un ColorEventMulticaster que pasará cualquier llamada al método colorSeleccionado() a los dos objetos, tal como refleja la figura siguiente.

En el caso de querer eliminar un receptor, se usará el método estático remove(). Este método devuelve un ColorListener formado por la lista de ColorListener a con el ColorListener b eliminado. Si a o b son nulos, se devuelve null. Sin embargo, si a es una lista ColorEventMulticaster, entonces se puede eliminar b de los árboles y combinar el resultado. En otro caso, se devuelve a, porque será un ColorListener. La figura siguiente intenta mostrar estas posibilidades gráficamente.

Utilización del Selector de Color

El selector de color se puede utilizar ahora como cualquier otro de los Componentes que se han visto en el AWT, es decir, se puede añadir a un Contenedor y luego registrar a los objetos interesados en recibir sus eventos a través del método addColorListener(). Cuando el usuario seleccione un color, los receptores serán notificados a través de dus métodos colorSeleccionado().

El ejemplo java1329.java, muestra el uso que se puede hacer del Componente, en este caso, muy trivial.

```
import java.awt.*;

public class java1329 implements ColorListener {
    public void colorSeleccionado( ColorEvent evt ) {
        System.out.println( "Color Seleccionado: "+evt.getColor() );
        }

    public static void main( String args[] ) {
        Frame frame = new Frame( "Tutorial de Java, AWT" );
        SelectorColor selector = new SelectorColor();
        java1329 ejemplo = new java1329();

        selector.addColorListener( ejemplo );
        frame.add( "Center",selector );
        frame.pack();
        frame.setVisible( true );
        }
    }
```

El ejemplo simplemente consiste en imprimir el color seleccionado, pero muestra claramente que el uso del Componente que se ha diseñado no difiere en absoluto de la forma de empleo de cualquier otro de los Componentes estándar del AWT.

Imprimir con AWT

La clase Toolkit es una clase que proporciona un interfaz independiente de plataforma para servicios específicos de esas plataformas, como pueden ser: fuentes de caracteres, imágenes, impresión y parámetros de pantalla. El constructor de la clase es abstracto y, por lo tanto, no se puede instanciar ningún objeto de la clase Toolkit; sin embargo, sí que se puede obtener un objeto Toolkit mediante la invocación del método getDefaultToolkit(), que devolverá un objeto de este tipo, adecuado a la plataforma en que se esté ejecutando.

De entre los muchos métodos de la clase Toolkit, el que representa el máximo interés en este momento es el método getPrintJob(), que devuelve un objeto de tipo PrintJob para usarlo en la impresión desde Java.

En Java hay, al menos, dos formas de poder imprimir. Una es coger un objeto de tipo Graphics, que haga las veces del papel en la impresora y dibujar, o pintar, sobre ese objeto. La otra, consiste en preguntar a un Componente, o a todos, si tienen algo que imprimir, y hacerlo a través del método printAll().

HolaMundo

Como el camino siempre se muestra andando, como decía el poeta, a continuación se muestra una de estas formas de imprimir para, como no, hacer aparecer el saludo ya conocido del "Hola Mundo!" en la impresora. El ejemplo HolaMundoPrn.java, cuyo código reproducen las siguiente líneas, consigue esto.

```
import java.awt.*;
class HolaMundoPrn extends Frame {
  static public void main( String args[] ) {
    // Creamos un Frame para obtener un objeto PrintJob sobre él
    Frame f = new Frame( "prueba" );
    f.pack();

    // Se obtiene el objeto PrintJob
    PrintJob pjob = f.getToolkit().getPrintJob( f,
      "Impresion del Saludo",null );
    // Se obtiene el objeto graphics sobre el que pintar
    Graphics pg = pjob.getGraphics();
    // Se fija el font de caracteres con que se escribe
    pg.setFont( new Font( "SansSerif",Font.PLAIN,12 ) );
    // Se escribe el mensaje de saludo
    pg.drawString( "Hola Mundo!",100,100 );
    // Se finaliza la página
    pg.dispose();
    // Se hace que la impresora termine el trabajo y escupa la página
    pjob.end();
    // Se acabó
    System.exit( 0 );
  }
}
```

Aunque sencillo, en el código se pueden observar las acciones y precauciones que hay que tomar a la hora de mandar algo a imprimir, y que se resumen en la siguiente lista:

El objeto PrintJob se debe crear sobre un Frame, con lo cual se pordrían asociar siempre a las aplicaciones visuales.

Cuando se crea una clase PrintJob, el sistema presenta el cuadro de diálogo de control de la impresora, en donde se puede seleccionar el tipo de impresora, el tamaño del papel o el número de copias que se desean obtener.

Aunque el objeto PrintJob se ha de crear sobre un Frame, no es necesario que éste sea visibleo tenga un tamaño distinto de cero.

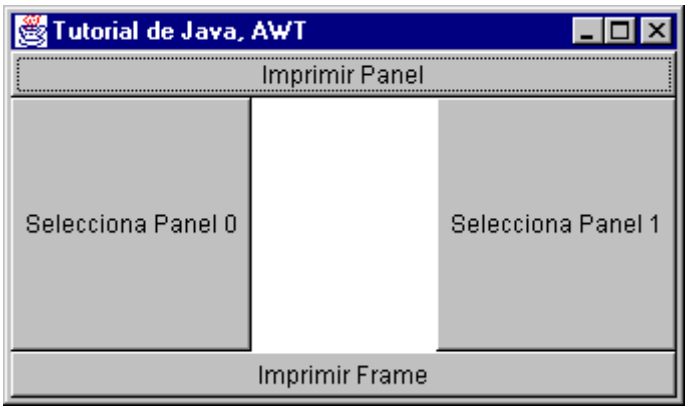
Antes de escribir nada en la impresora, es necesario seleccionar la fuente de caracteres con que se desea hacerlo, el sistema no proporciona ningún font de caracteres por defecto.

La impresión se consigue pintando sobre el objeto Graphics de la impresora.

La impresión se realiza página a página, de tal modo que cada una de ellas tiene su propio objeto Graphics. El método dispose() se utiliza para completar cada una de las páginas y que la impresora la lance.

Imprimir Componentes

El propósito del ejemplo java1330.java es mostrar la capacidad para imprimir de los Componentes del AWT que se encuentran en un Contenedor, bien sea éste el Contenedor raiz o se encuentre incluido en otro Contenedor. La imagen siguiente reproduce la ventana que se presenta en pantalla al ejecutar el ejemplo.



El programa coloca uno de dos objetos Panel seleccionables y cuatro objetos Button, sobre un objeto Frame. Uno de los botones tiene un receptor que hace que el Panel que se encuentre seleccionado y todos los Componentes que se encuentren en ese Panel sean impresos. Otro de los botones tiene un receptor que hace que el Frame raiz y todos los Componentes sean enviados a la impresora. En realidad, el Frame no se puede imprimir a sí mismo, sino que hace que todos sus Componentes sean impresos, lo cual contradice un poco lo que aparece en la documentación de JavaSoft, que dice, literalmente: "Imprime este componente y todos sus subcomponentes". Lo mismo, probablemente, le pase a Panel, pero este objeto Panel no tiene ninguna característica que permita saber si está siendo o no impreso.

Los dos botones anteriores comparten el mismo objeto receptor, pero la acción que realizan es la descrita. Los otros dos botones se utilizan para seleccionar los dos paneles. Es decir, el usuario puede seleccionar entre los dos paneles diferentes y hace que el que se esté visualizando en el Frame se envíe a la impresora.

Cuando el Panel seleccionado se está imprimiendo, los otros Componentes del Frame son ignorados. Sin embargo, cuando el Frame se está imprimiendo, todos los Componentes del Frame, incluido el Panel seleccionado, serán enviados a imprimirse.

El contenido de los paneles es solamente como muestra, por ello uno contiene una etiqueta, un campo de texto y un botón no activo, y el otro contiene una etiqueta, un campo de texto y dos botones inactivos.

A continuación se repasan los fragmentos de código que pueden resultar interesantes del ejemplo. El primer fragmento de código interesante es el que muesta la clase de control, con el método main() que instancia un objeto de la clase IHM, que es el que en realidad aparece en la pantalla.

```
public class java1330 {
    public static void main( String args[] ) {
        // Se instancia un objeto de la clase Interfaz Gráfico
        IHM ihm = new IHM();
        }
    }
```

En el código siguiente se muestra el comienzo de la clase IHM, incluyendo la declaración de diferentes variables de referencia.

```
class IHM {
    // El contenedor miFrame y todos los componenete que contiene, serán
    // impresos o enviados a un fichero de impresora cuando se pulse el
    // botón con el rótulo "Imprimir Frame"
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );

    // El contenedor panelAImprimir y todos los componentes que contiene,
    // serán impresos o enviados a un fichero de impresora cuando se
    // pulse el botón con el rótulo "Imprimir Panel"
    Panel panelAImprimir = null;

    // Referencias a los dos paneles seleccionables
    Panel panel0;
    Panel panel1;
```

El contenedor miFrame y todos los Componentes que contiene serán enviados a la impresora, o a un fichero de impresión, cuando se pulse el botón rotulado "Imprimir Frame". El contenedor panelAImprimir y todos sus Componentes, serán impresos o enviados a un fichero de impresión al pulsar el botón "Imprimir Panel". Las variables de referencia de los objetos Panel, son referencias a los dos paneles seleccionables.

Tanto el código correspondiente al constructor de la clase IHM, como el utilizado para construir los paneles es muy semejante al que se ha visto en secciones anteriores, por lo que no merece la pena revisarlo, aunque no deja de tener un cierto interés.

Las líneas de código que aparecen a continuación sí que ya resultan interesantes para el objetivo de esta sección. Muestran la definición de una clase anidada de la clase IHM, que es utilizada por el objeto Panel, referenciado por panelAImprimir, o el objeto Frame, referenciado por miFrame, sean impresos.

```
class PrintActionListener implements ActionListener {
    public void actionPerformed( ActionEvent evt ) {
        // Coge un objeto PrintJob. Esto hace que aparezca el
        // diálogo estándar de impresión, que si se cierra sin
        // imprimir devolverá un nulo
        PrintJob miPrintJob = miFrame.getToolkit().
            getPrintJob( miFrame,"Tutorial de Java, AWT",null );
```

Esta clase es un receptor de eventos de tipo Action, y el código del método actionPerformed() es el que hace que tenga lugar la impresión. Las líneas de código anteriores, correspondientes al comienzo de la clase anidada, son utilizados para conseguir un objeto de tipo PrintJob. Esto hace que el diálogo estándar de selección de impresora

aparezca en la pantalla. Si el usuario cierra este diálogo mediande el botón "Cancelar", es decir, sin activar la impresión, el método getPrintJob() devolverá null.

El siguiente código, de la misma clase anidada, comprueba que efectivamente, el usuario quiere imprimir para proceder con esa impresión.

```
if( miPrintJob != null ) {
    // Coge el objeto gráfico que va a imprimir
    Graphics graficoImpresion = miPrintJob.getGraphics();

    if( graficoImpresion != null ) {
        // Invoca la método printAll() del objeto Panel, o del
        // objeto Frame para hacer que los componentes del
        // que sea se dibujen sobre el objeto gráfico y se
        // pinten sobbre el papel de la impresora
        if( evt.getActionCommand().equals( "Imprimir Panel" ) )
            panelAImprimir.printAll( graficoImpresion );
        else
            miFrame.printAll( graficoImpresion );
```

El primer paso, en caso afirmativo, es utilizar el método getGraphics() para obtener un objeto de tipo Graphics, que representará el papel de la impresora. Este objetos erá el que sea requerido por el método printAll(), que se invocará seguidamente.

Una vez comprobada la validez del objeto Graphics, hay que determinar los objetos que se van a aimprimir. El contenedor externo es un objeto Frame, y el contenedor interno es un objeto Panel. Esto se hace invocando el método getActionCommand() sobre el objeto que ha sido origen del evento ActionEvent. Luego, depende del origen del evento que el método printAll() sea invocado sobre el Panel o el Frame, pasando el objeto Graphics como parámetro.

Como lo que se están imprimiendo son gráficos, la calidad de la impresión dependerá de la impresora, en la Laserjet-5MP PostScript que utiliza el autor, la calidad en blanco y negro es realmente buena.

Una vez concluida la impresión, es necesario hacer que el papel salga y, también, liberar todos los recursos que hayan sido cogidos por el objeto Graphics. Esto se consigue invocando al método dispose() sobre el objeto Graphics, tal como se muestra en el código que se reproduce a continuación.

```
            // Hacemos que se libere el papel de la impresora y los
             // recursos del sistema que estaba utilizando el
             // objeto gráfico
             graficoImpresion.dispose();
             }
        else
            System.out.println( "No se puede imprimir el objeto" );
        // Se concluye la impresión y se realiza la limpieza
        // necesaria
        miPrintJob.end();
        }
    else
        System.out.println( "Impresion cancelada" );
    }
}
```

En el código anterior también se llama al método end() sobre el objeto PrintJob, porque hay que ser obedientes y hacer las cosas tal como indica JavaSoft para "finalizar la impresión y realizar cualquier limpieza necesaria" (sic).

Las líneas de código anteriores también contienen la parte del else correspondiente al if que comprueba la validez del objeto Graphics. Si no es válido, el proceso de impresión

se corta y aparecen algunos mensajes en pantalla, aunque probablemente estaría más elegante el lanzar una excepción, pero como ejemplo es suficiente.

Las dos definiciones de las clases ActionListener que siguen en el código del ejemplo son los utilizados para el control del proceso de selección del panel que se va a visualizar. El lector debería echarles un vistazo, aunque aquí no se reproduzcan por no ser el tema concreto de la sección.

El siguiente ejemplo, java1331.java, muestra la capacidad para imprimir selectivamente Componentes del AWT correspondientes a un Contenedor embebido en otro Contenedor.

La palabra selectivamente se usa para diferenciar el método de impresión del visto en el ejemplo anterior, porque en el programa se utilizaba el método printAll() para imprimir todos los Componentes del Contenedor y, en este nuevo programa, se ha incorporado la capacidad de seleccionar los Componentes que van a ser impresos y también, la posibilidad de seleccionar información de estos Componentes para enviarla a imprimir.

De forma semejante al ejemplo anterior java1330.java, el programa coloca uno de los dos objetos Panel seleccionables y tres objetos Button sobre un objeto Frame. El objeto Panel sabe cómo imprimir sus Componentes a través del método paint() que está sobrescrito.

Uno de los botones tiene un receptor que hace que el Panel seleccionado se imprima, lo cual requiere que cada Panel tenga un método paint() sobrescrito y para ello, cada Panel creado debe extender la clase Panel. El método paint() define la forma en que se va a imprimir el Panel.

Los otros dos botones se utilizan para seleccionar entre los dos paneles a la hora de presentarlos en pantalla e imprimirlos. En otras palabras, el usuario puede seleccionar entre los dos paneles y hacer que el que esté presente en el Frame se imprima. Cuando este Panel seleccionado se está imprimiendo, los otros Componentes del Frame son ignorados.

```
import java.awt.*;
import java.awt.event.*;

public class java1331 {
    public static void main( String args[] ) {
        // Se instancia un objeto de la clase Interfaz Gráfico
        IHM ihm = new IHM();
        }
    }
class IHM {
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );

    // El contenedor panelAImprimir y todos los componentes que contiene,
    // serán impresos o enviados a un fichero de impresora
    Panel areaAImprimir = null;

    // Se colocan dos paneles sobre el Frame de forma que se pueda
    // seleccionar cualquiera de ellos. Son paneles propios, es decir,
    // que son creados exprofeso, ya que su apariencia es diferente y
    // la forma de imprimirse es distinta
    MiPanel0 panel0;
    MiPanel1 panel1;

    public IHM() {
```

```
        // Este botón hace que el contenedor que esté actualmente
        // referenciado
        // por areaAImprimir se imprima a sí mismo
        Button botonImprimir = new Button( "Imprimir" );
        botonImprimir.addActionListener( new PrintActionListener() );
        miFrame.add( botonImprimir,"North" );

       // Los siguientes botones son los que se utilizan para seleccionar
       // cual de los dos paneles se presentará en pantalla y será el que
       // se imprima
        Button botonPanel0 =  new Button( "Selecciona Panel 0" );
        botonPanel0.addActionListener( new Panel0Listener() );
        miFrame.add( botonPanel0,"West" );

        Button botonPanel1 =  new Button( "Selecciona Panel 1" );
        botonPanel1.addActionListener( new Panel1Listener() );
        miFrame.add( botonPanel1,"East" );

        // Aquí se construyen los paneles que luego se asignarán a la
        // referencia areaAImprimir al hacer una selección. La rutina
        // de impresión hará que el contenedor referenciado por
        // areaAImprimir y todos sus componentes sean impresos
        panel0 = new MiPanel0();
        panel1 = new MiPanel1();

        // Es necesaria una referencia válida en areaAImprimir para
        // evitar la presencia de una excepción por puntero nulo
        // al realizar la selección e intentar eliminar la referencia
        // anterior
        areaAImprimir = panel0;

        miFrame.setSize( 340,200 );
        miFrame.setVisible( true );

        // Esta es una clase anidada anónima que se utiliza para
        // concluir la ejecución del programa cuando el usuario
        // decide cerrar el Frame
        miFrame.addWindowListener( new WindowAdapter() {
            public void windowClosing( WindowEvent evt ) {
                System.exit( 0 );
                }
            } );
        }

// Esta es una clase anidada utilizada para imprimir el
// contenedor referenciado por areaAImprimir. Esto se consigue
// aceptando un contexto de impresora y pasándoselo al
// método paint() del contenedor referenciado por areaAImprimir.
class PrintActionListener implements ActionListener {

  public void actionPerformed( ActionEvent evt ) {
      // Coge un objeto PrintJob. Esto hace que aparezca el
      // diálogo estándar de impresión, que si se cierra sin
      // imprimir devolverá un nulo
      PrintJob miPrintJob = miFrame.getToolkit().
          getPrintJob( miFrame,"Tutorial de Java, AWT",null );
      if( miPrintJob != null ) {
 // Coge el objeto gráfico que va a imprimir
          Graphics graficoImpresion = miPrintJob.getGraphics();
          if( graficoImpresion != null ) {
   // Invoca la método paint() del objeto Panel que se
   // ha creado para hacer que los componentes del
   // que sea se dibujen sobre el objeto gráfico y se
```

```
        // pinten sobbre el papel de la impresora
                areaAImprimir.paint( graficoImpresion );
          // Hacemos que se libere el papel de la impresora y los
          // recursos del sistema que estaba utilizando el
          // objeto gráfico
                graficoImpresion.dispose();
                }
             else
                System.out.println( "No se puede imprimir el objeto" );

          // Se concluye la impresión y se realiza la limpieza
          // necesaria
               miPrintJob.end();
               }
           else
               System.out.println( "Impresion cancelada" );
           }
      }

    // Esta es una de las clases propias que extienden a la clase
    // Panel, para conseguir el panel que se desea. Los objetos de
    // esta clase saben cómo imprimirse a sí mismos cuando se
    // invoca a su método paint() pasándole como parámetro un
    // objeto de tipo PrintGraphics. En el método sobrescrito
    // paint() es donde se indica la forma en que se imprime.
    class MiPanel0 extends Panel {
        // Estos son los componentes que contiens los datos que se
        // van a imprimir
        Label labPanel0;
        TextField textoPanel0;
        Button botonPanel0;

    // Este es el constructor para los objetos de la clase
    // Panel que se ha creado
        MiPanel0() {
            labPanel0 = new Label( "Panel 0" );
            this.add( labPanel0 );
            textoPanel0 = new TextField( "Texto" );
            this.add( textoPanel0 );
            botonPanel0 = new Button( "Boton Panel 0" );
            this.add( botonPanel0 );
            this.setBackground( Color.yellow );
            }

   // Este es el método sobrescrito paint(), que ni tan
   // siquiera hace el intento de manipular el tamaño de
   // la fuente de caracteres, porque con la versión del
   // JDK que estoy utilizando, hay un crash bestial
        public void paint( Graphics g ) {
            // Hay que separar el pintado sobre la pantalla de la
            // impresión sobre papel, de tal forma que esto último
            // solamente se ejecute en caso de que se pase como
            // parámetro un objeto de tipo PrintGraphics. En
            // cualquier otro caso, la información y componentes
            // aparecerá sobre la pantalla
            if( g instanceof PrintGraphics ) {
                // Esta versión de paint() se limita a imprimir una
                // línea de cabecera y a extraer datos de los componentes
                // del Panel, imprimiéndolos en líneas sucesivas
        int margenIzqdo = 10; // Posición X de cada línea
        int margenSup = 20;  // Posición Y de la primera línea
        int pasoLinea = 13;  // Incremento o salto entre líneas

        // El cotnexto de impresión no tiene una fuente de
```

```
        // caracteres por defecto, así que hay que proporcionársela
                g.setFont( new Font( "Serif",Font.BOLD,18 ) );
                g.drawString( "Hola desde el Panel 0 del TUTORIAL",
                    margenIzqdo,margenSup += pasoLinea );
                g.setFont( new Font( "Serif",Font.PLAIN,10 ) );
                g.drawString(      "Texto      de      la      Etiqueta:
"+labPanel0.getText(),
                    margenIzqdo,margenSup += pasoLinea );
                g.drawString( "Texto del Campo: "+textoPanel0.getText(),
                    margenIzqdo,margenSup += pasoLinea );
                g.drawString(         "Rotulo         del         Boton:
"+botonPanel0.getLabel(),
                    margenIzqdo,margenSup += pasoLinea );
                }
      // En el caso de que g no sea un objeto de tipo
      // PrintGraphics
            else
        // Se invoca el método paint() de la clase Panel
                super.paint( g );
            }
        }

 // Esta es la otra de las clases propias que extienden a la clase
 // Panel, para conseguir el panel que se desea. Los objetos de
 // esta clase saben cómo imprimirse a sí mismos cuando se
 // invoca a su método paint() pasándole como parámetro un
 // objeto de tipo PrintGraphics. En el método sobrescrito
 // paint() es donde se indica la forma en que se imprime.
    class MiPanel1 extends Panel{
    // Estos son los componentes que contiens los datos que se
    // van a imprimir
        Label labPanel1;
        TextField textoPanel1;
        Button botonPanel_0;
        Button botonPanel_1;

        MiPanel1(){
            labPanel1 = new Label("Panel 1");
            this.add(labPanel1);
            textoPanel1 = new TextField("Texto");
            this.add(textoPanel1);
            botonPanel_0 = new Button("Un Boton");
            this.add(botonPanel_0);
            botonPanel_1 = new Button("Otro Boton");
            this.add(botonPanel_1);
            this.setBackground(Color.red);
            }

    // Este es el método sobrescrito paint(), que ni tan
    // siquiera hace el intento de manipular el tamaño de
    // la fuente de caracteres, porque con la versión del
    // JDK que estoy utilizando, hay un crash bestial
        public void paint( Graphics g ) {
      // Hay que separar el pintado sobre la pantalla de la
      // impresión sobre papel, de tal forma que esto último
      // solamente se ejecute en caso de que se pase como
      // parámetro un objeto de tipo PrintGraphics. En
      // cualquier otro caso, la información y componentes
      // aparecerá sobre la pantalla
            if( g instanceof PrintGraphics ) {
        // Esta versión de paint() se limita a imprimir una
        // línea de cabecera y a extraer datos de los componentes
        // del Panel, imprimiéndolos en líneas sucesivas
                int margenIzqdo = 10; // Posición X de cada línea
                int margenSup = 20;   // Posición Y de la primera línea
```

```
                int pasoLinea = 13;   // Incremento o salto entre líneas

        // El contexto de impresión no tiene una fuente de
        // caracteres por defecto, así que hay que proporcionársela
                g.setFont(new Font("Serif", Font.BOLD, 18));
                g.drawString( "Hola desde el Panel 1 del TUTORIAL",
                    margenIzqdo,margenSup += pasoLinea );
                g.setFont( new Font( "Serif",Font.PLAIN,10 ) );
                g.drawString(      "Texto      de      la      Etiqueta:
"+labPanel1.getText(),
                    margenIzqdo,margenSup += pasoLinea );
                g.drawString( "Texto del Campo: "+textoPanel1.getText(),
                    margenIzqdo,margenSup += pasoLinea );
                g.drawString(         "Rotulo         del         Boton:
"+botonPanel_0.getLabel(),
                    margenIzqdo,margenSup += pasoLinea );
                g.drawString(         "Rotulo         del         Boton:
"+botonPanel_1.getLabel(),
                    margenIzqdo,margenSup += pasoLinea );
                }
      // Esto en el caso de que g no se un objeto de tipo
      // PrintGraphics
            else
                super.paint( g );
            }
        }

  // Esta es una clase anidada que permite seleccionar e
  // imprimir el panel0. Evidentemente, esta clase y la quue
  // sigue, se pueden combinar en una sola que utilizaría
  // el origen del evento para determinar el panel que debe
  // enviar a la impresora
    class Panel0Listener implements ActionListener {
        public void actionPerformed( ActionEvent evt ) {
            miFrame.remove( areaAImprimir );
            areaAImprimir = panel0;
            miFrame.add( areaAImprimir,"Center" );
            miFrame.invalidate();   // Fuerza el repintado
            miFrame.setVisible( true );
            }
        }

    // Esta es una clase anidada que permite seleccionar e
    // imprimir el panel1
    class Panel1Listener implements ActionListener{
        public void actionPerformed( ActionEvent evt ) {
            miFrame.remove( areaAImprimir );
            areaAImprimir = panel1;
            miFrame.add( areaAImprimir,"Center" );
            miFrame.invalidate();   // Fuerza el repintado
            miFrame.setVisible( true );
            }
        }
    }
```

En este caso, el formato de impresión definido en el método paint() hace que el texto situado en el objeto Label y en el objeto TextField se impriman y también los títulos de los objetos Button. Esto es solamente como ejemplo, porque utilizando la misma técnica, el programador tiene completa libertad a la hora de asociar la información que será impresa con cada uno de los Componentes del programa.

El código del ejemplo, como el lector habrá podido observar, es casi idéntico o muy similar al del ejemplo anterior, por lo que no se repite su explicación. La diferencia más

significativa reside en el hecho de la declaración de los objetos Panel propios, porque no pueden ser objetos de tipo Panel como en el ejemplo anterior, ya que es necesario que tengan sobrescrito el método paint(). Además, deben ser de diferentes tipos porque tanto su apariencia como la forma en que se imprime su contenido son diferentes.

Si se vuelven a repasar los fragmentos de código más interesantes del ejemplo, se pueden reproducir en primer lugar las líneas que crean la clase IHM, que muestran las referencias a los dos paneles seleccionables que se van a instalar sobre el objeto Frame.

```
class IHM {
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );

    // El contenedor panelAImprimir y todos los componentes que contiene,
    // serán impresos o enviados a un fichero de impresora
    Panel areaAImprimir = null;


    // Se colocan dos paneles sobre el Frame de forma que se pueda
    // seleccionar cualquiera de ellos. Son paneles propios, es decir,
    // que son creados exprofeso, ya que su apariencia es diferente y
    // la forma de imprimirse es distinta
    MiPanel0 panel0;
    MiPanel1 panel1;
```

La diferencia con el programa anterior estriba fundamentalmente en el tipo del Panel, MiPanelX. En este caso no pueden ser de tipo Panel, como en el programa anterior, porque va a ser necesario sobreescribir su método paint(), con lo cual no queda más remedio que extender la clase Panel. Además, han de ser de tipos diferentes porque su apariencia y comportamiento a la hora de la impresión van a ser diferentes. Por ello, el constructor difiere del ejemplo anterior en lo indicado, en el resto es muy semejante, tal como se muestra en el código que sigue.

```
public IHM() {
...
    // Aquí se construyen los paneles que luego se asignarán a la
    // referencia areaAImprimir al hacer una selección. La rutina
    // de impresión hará que el contenedor referenciado por
    // areaAImprimir y todos sus componentes sean impresos
    panel0 = new MiPanel0();
    panel1 = new MiPanel1();
...
    }
```

Ahora le toca el turno a la clase anidada de la clase IHM que se utiliza para que el objeto Panel referenciado por la variable de referencia areaAImprimir, se imprima. Esto se consigue solicitando un contexto para imprimir y pasándoselo al método sobrescrito paint() del panel referenciado por la variable anterior. La mayor parte del código es similar al ejemplo anterior, excepto por la línea que se ha dejado huérfana en el siguiente código.

```
public void actionPerformed( ActionEvent evt ) {
...
            areaAImprimir.paint( graficoImpresion );
...
    }
```

El fragmento de código se encuentra dentro del método actionPerformed() de la clase PrintActionListener, donde se invoca al método paint() propio, lo cual hace que el el objeto se imprima. Todo esto viene desde la clase en la cual se instanciaron los paneles, ya que los objetos de esta clase saben cómo imprimirse a través del método paint() sobreescrito. EN este método paint() es donde hay que definir el formato de impresión que se desea.

Una cosa a resaltar es que la clase extiende a la clase Panel, para poder sobreescribir su método paint(). Dentro del método es necesario separar el pintado en pantalla del pintado en impresora, porque de no hacer pueden aparecer cosas en pantalla que en realidad están destinadas a la impresora.

Se hace una comprobación inicial para ejecutar el código del método paint() solamente si el objeto Graphics es de tipo PrintGraphics; sino, se invoca al método paint() de la superclase para seguir preservando la posibilidad de pintar en pantalla. El método paint() sobrescrito imprime una línea de cabecera y extrae datos de los componentes del panel, imprimiéndolos en sucesivas líneas. Aquí es donde se puede colocar el código que formatee la salida impresa al gusto.

Hay que tener en cuenta que la impresión directa no tiene una fuente de caracteres por defecto, así que hay que proporcionarle una, porque sino, el sistema puede quejarse. El siguiente código reproduce el método paint().

```
public void paint( Graphics g ) {
    // Hay que separar el pintado sobre la pantalla de la
    // impresión sobre papel, de tal forma que esto último
    // solamente se ejecute en caso de que se pase como
    // parámetro un objeto de tipo PrintGraphics. En
    // cualquier otro caso, la información y componentes
    // aparecerá sobre la pantalla
    if( g instanceof PrintGraphics ) {
        // Esta versión de paint() se limita a imprimir una
        // línea de cabecera y a extraer datos de los componentes
        // del Panel, imprimiéndolos en líneas sucesivas
    int margenIzqdo = 10; // Posición X de cada línea
    int margenSup = 20;  // Posición Y de la primera línea
    int pasoLinea = 13;   // Incremento o salto entre líneas

        // El contexto de impresión no tiene una fuente de
        // caracteres por defecto, así que hay que proporcionársela
        g.setFont( new Font( "Serif",Font.BOLD,18 ) );
        g.drawString( "Hola desde el Panel 0 del TUTORIAL",
            margenIzqdo,margenSup += pasoLinea );
        g.setFont( new Font( "Serif",Font.PLAIN,10 ) );
        g.drawString( "Texto de la Etiqueta: "+labPanel0.getText(),
            margenIzqdo,margenSup += pasoLinea );
        g.drawString( "Texto del Campo: "+textoPanel0.getText(),
            margenIzqdo,margenSup += pasoLinea );
        g.drawString( "Rotulo del Boton: "+botonPanel0.getLabel(),
            margenIzqdo,margenSup += pasoLinea );
        }
    // En el caso de que g no sea un objeto de tipo
    // PrintGraphics
    else
        // Se invoca el método paint() de la clase Panel
        super.paint( g );
    }
```

Como se puede comprobar, la impresión consiste simplemente en invocar al método drawString() sobre el objeto PrintGraphics del mismo modo que se hace en cualquier otro programa o applet, incluso en el básico HolaMundo. Lo interesante, si el lector observa, es que se está pintando sobre el papel; es decir, que no hay que resignarse a imprimir solamente texto, sino que la imaginación es la que impone el límite.

La parte final del código anterior, correspondiente a la parte del else, es una invocación al método paint() de la superclase, Panel en este caso, cuando el método paint() sobrescrito es invocado, pero no se le pasa como parámetro un objeto de tipo

PrintGraphics. Esto es necesario porque hay que seguir preservando la posibilidad de pintar sobre la pantalla.

La clase anterior está seguida por otra semejante para el otro panel, así que no se va a insistir sobre ello.

Clase Impresora

Quizá estas clases parezcan un acercamiento un tanto primitivo a la impresión, ya que hay que manejar directamente cada salto de línea, cada fuente de caracteres y controlar cada página por separado. Así que se puede intentar hacer una clase un poco más general como ejemplo de que la complejidad se detiene en donde uno se lo proponga. El ejemplo java1334.java crea una clase Impresora que oculta alguna de la complejidad inherente a la impresión en Java.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class java1334 extends Frame implements ActionListener,
WindowListener {
  Button botonImp;
  Button botonVer;
  Impresora impresora;

  public java1334() {
     super( "Tutorial de Java, Impresora" );
     addWindowListener( this );

     // Panel para contener los botones
     Panel panel = new Panel();
     add( "South",panel );
     // Se añaden los botones que permiten imrpimir o previsualizar el
     // texto que se va a imprimir
     botonImp = new Button( "Imprimir" );
     panel.add( "South",botonImp );
     botonImp.addActionListener( this );
     botonVer = new Button( "Visualizar" );
     panel.add( botonVer );
     botonVer.addActionListener( this );

     // Se crea un objeto de la clase que se ha creado para controlar la
     // impresión
     impresora = new Impresora( this );

     setBounds( 100,100,250,100 );
     setVisible( true );
     // Se crea el texto que se va a imprimir, como ejemplo
     imprimirTexto();
   }

  // Este método es el que crea las líneas de texto que se van a
  // imprimir, utilizando los métodos de formateo que se han creado
  // para la clase Impresora, colocando ese texto en el objeto de
  // tipo Impresora que se ha creado al arrancar
  private void imprimirTexto() {
    int i = 1;

    impresora.setFont( new Font( "SanSerif",Font.PLAIN,12 ) );
    impresora.print( i++ + "." );
    impresora.tab( 10 );
    impresora.println( "Ejemplo para la impresora" );
```

```
    impresora.print( i++ + "." );
    impresora.tab( 10 );
    impresora.setFont( new Font( "Helvetica",Font.BOLD,12 ) );
    impresora.println( "Texto con otro tipo de fuente, negrilla" );

    impresora.print( i++ + "." );
    impresora.tab( 10 );
    impresora.setFont( new Font( "Helvetica",Font.ITALIC,18 ) );
    impresora.println( "Lo mismo pero cayendo a derechas" );
  }

  // Aquí es donde se controla la acción de los botones, para
  // imprimir o visualizar, según lo que se haya seleccionado
  public void actionPerformed( ActionEvent evt ) {
    Object obj = evt.getSource();

    // Una vez que se sabe el botón que es, se realiza la acción
    if( obj == botonImp ) {
      // Se genera el salto de página y se finaliza el Job, de forma
      // que la impresora sacará a fuera la página
      impresora.saltoPagina();
      impresora.finImpresion();
    }

    if( obj == botonVer ) {
      // Se manda lo mismo a la pantalla
      preVisualizacion();
    }
  }

   private void preVisualizacion() {
      Dimension dim = impresora.tamPagina();
      setBounds( 0,0,dim.width,dim.height );
      impresora.setBounds( 0,0,dim.width,dim.height );
      impresora.setVisible( true );
   }

  static public void main( String argv[] ) {
      new java1334();
   }

   public void windowClosing( WindowEvent evt ) {
     // Cuando se pica el botón de salir se corta la ejecución
    System.exit( 0 );
   }
  // Los demás métodos de control de la ventana hay que
  // sobreescribirlos porque se está utilizando el interfaz
  // y hay que implementar todos los métodos, aunque no se
  // haga nada en ellos
    public void windowClosed( WindowEvent evt ){}
    public void windowOpened( WindowEvent evt ){}
    public void windowIconified( WindowEvent evt ){}
    public void windowDeiconified( WindowEvent evt ){}
    public void windowActivated( WindowEvent evt ){}
    public void windowDeactivated( WindowEvent evt ){}
}


// Esta es la clase Impresora objeto real del ejemplo
class Impresora extends Canvas {
  Frame f;      // Frame padre
  PrintJob pjob;  // Objeto de impresión
  Graphics pg;      // Objeto donde pintar lo que se imprimirá
    Vector objetos;  // Array de instrucciones a la impresora
```

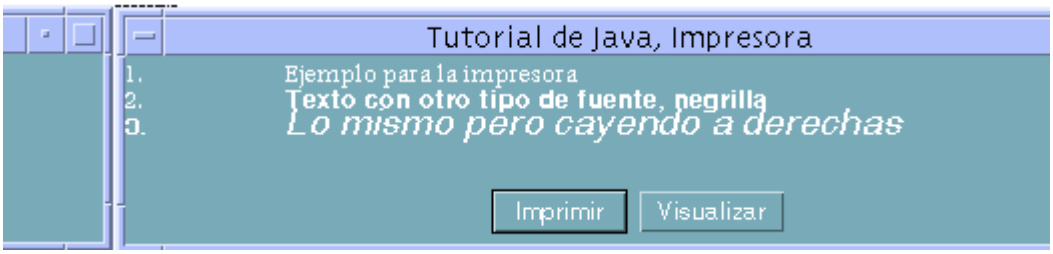
```
  Point pt;     // Posición actual de la impresión
  Font fnt;          // Font actual
  Font tabFont;   // Font para calcular los tabulados

public Impresora( Frame frm ) {
  // Se crea el Frame y se añade el objeto
  f = frm;
  f.add( this );
  // Pero no se visualiza
  setVisible( false );
  // Todavía no hay nada que imprimir
  pjob = null;

  // Posición inicial de impresión
  pt = new Point( 0,0 );
  // Inicialización del array de instrucciones
   objetos = new Vector();
   // Fuentes a utilizar por defecto
   // Para los tabuladores utilizamos una fuente "no proporcional" para
   // que los espacios tengan el mismo tamaño que las letras, de forma
   // que una "m" ocupe exactamente el mispo espacio que " "
   tabFont = new Font( "MonoSpaced",Font.PLAIN,12 );
   fnt = new Font( "SansSerif",Font.PLAIN,12 );
   }

  // Método para fijar la fuente de caracteres a utilizar en la
  // impresión hasta que no se indique otro
  public void setFont( Font f ) {
    objetos.addElement( new printFont( f ) );
  }

  // Este método imprime una cadena, pero no realiza el salto de
  // línea, de tal modo que lo que se imprima luego, se hará
  // a continuación de lo que se envíe a la impresora en este
  // método
  public void print( String s ) {
    objetos.addElement( new printString( s ) );
  }

  // Este método imprime una cadena y salta a la línea siguiente
  public void println( String s ) {
    print( s );
    objetos.addElement( new saltoLinea() );
  }

  // Este método realiza el salto de página. Como la impresión
  // se realiza página a página, hay que concluir el trabajo y
  // posicionarse de nuevo en la posición incial para pintar la
  // siguiente página
  public void saltoPagina() {
    if( pjob == null ) {
      pjob = getToolkit().getPrintJob( f,"Impresora",null );
    }
    pg = pjob.getGraphics();
    print( pg );
    pg.dispose();
    pt = new Point( 0,0 );  // posición inicial en la página
    objetos = new Vector(); // objetos a imprimir
  }

  // Este método es el que concluye totalmente la impresión
  // comprobando que ya no quedan objetos que enviar a la
  // impresora
  public void finalize() {
```

```
    if( objetos.size() > 0 )
      saltoPagina();
    finImpresion();
  }

  // Concluye el trabajo de impresión
  public void finImpresion() {
    pjob.end();
  }

  // Incluye un tabulador, colocando tantos espacios como
  // saltos de tabulador
  public void tab( int tabstop ) {
    objetos.addElement( new printTab( tabFont,tabstop ) );
  }

  // Este método controla el tamaño de la página que se va a
  // imprimir, asegurándose de que hay un tamaño por defecto
  // para poder visualizar el contenido
  public Dimension tamPagina() {
    if( pjob == null ) {
      return( new Dimension( 620,790 ) );
    }
    else {
      pjob = getToolkit().getPrintJob( f,"Impresora",null );
      return( pjob.getPageDimension() );
    }
  }

  // Sobrescritura del método paint() para que se pueda realizar
  // la previsualización de lo que se va a imprimir
  public void paint( Graphics g ) {
    pt = new Point( 0,0 );
    print( g );
  }


  // Este es el método que realmente realiza la impresión sobre
  // la impresora, recorriendo la lista de objetos que se ha
  // construido y enviando todos sus elementos al objeto Impresora
  // que se ha creado, pintándolos igual que si lo estuviese
  // naciendo en la pantalla
  public void print( Graphics g ) {
    objImpresora imp;

    // Siempre hay que comenzar con alguna fuente
    f.setFont( fnt );
    for( int i=0; i < objetos.size(); i++ ) {
      imp = (objImpresora)objetos.elementAt( i );
      imp.draw( g,pt );
    }
  }
}


// Clase abstracta que define el método draw() para que
// las clases que implementan los distintos objetos que
// se van a controlar para facilitar el trabajo de imprimir
// con la impresora, lo sobreescriban pintando lo que les
// corresponda
abstract class objImpresora {
   abstract void draw( Graphics g,Point p );
}

// Salto de linea. Vuelve a la x=0 e incrementa la y en
```

```
// lo que ocupa la fuente de caracteres con que se está
// pintando
class saltoLinea extends objImpresora {
  public void draw( Graphics g,Point p ) {
    p.x = 0;
    p.y += g.getFontMetrics( g.getFont() ).getHeight();
  }
}

// Cadena. Pinta una cadena en la posición en que se encuentre
// posicionado el puntero de impresión y desplaza a este en la
// anchura de la cadena
class printString extends objImpresora {
  String cadena;

  public printString( String s ) {
    cadena = s;
  }

  public void draw( Graphics g,Point p ) {
    g.drawString( cadena,p.x,p.y);
    p.x += g.getFontMetrics( g.getFont() ).stringWidth( cadena );
  }
}

// Fuente de caracteres. Fija la fuente de caracteres que se va a
// utilizar para seguir imprimiendo
class printFont extends objImpresora {
  Font fuente;

  public printFont( Font f ) {
    fuente = f;
  }

  public void draw( Graphics g,Point p ) {
    g.setFont( fuente );
    if( p.y <= 0 ) {
      p.y = g.getFontMetrics( fuente ).getHeight();
    }
  }
}

// Tabulador. Desplaza el puntero que recorre el eje de la X en
// tantas posiciones como se indique. Para saber el desplazamiento
// que corresponde a cada uno de los puntos de tabulación, se
// utiliza el ancho de la letra M, que suele ser la más ancha,
// en previsión de que la fuente por defecto que se utilice para
// la tabulación sea una proporcional
class printTab extends objImpresora {
  static int tabulador = 0;
  int tab_dist;
  Font tabFnt;

  public printTab( Font tbFont,int tabdist ) {
    tabFnt = tbFont;
    tab_dist = tabdist;
  }

  public void draw( Graphics g,Point p ) {
    if( tabulador == 0 ) {
      tabulador = g.getFontMetrics( tabFnt ).stringWidth( "M" );
    }
    if( p.x < ( tab_dist * tabulador ) ) {
      p.x = tab_dist * tabulador;
    }
```

```
    }
}
```

Si se ejecuta el programa, en pantalla aparecerá una ventana con dos botones que permiten la impresión y la previsualización del contenido que se puede imprimir. La figura siguiente muestra precisamente la previsualización, en donde se observan las líneas de texto escritas con distintos fonts de caracteres. Si se activa el botón de impresión, aparecerá el diálogo del sistema que permite la selección de impresora y características de la impresión.



A continuación se repasan los trozos de código más interesantes del ejemplo. En primer lugar, hay que tener en cuenta que se debe levantar todo sobre un Frame, pero se puede observar que la clase Impresora se construye sobre un Canvas; es decir, se puede utilizar cualquier Contenedor que se pueda incorporar a un Frame, sin tener que usar un Frame en sí mismo. Bajo condiciones normales, el objeto Canvas permanecerá invisible y se ajustará a su tamaño mínimo: 1x1 pixels. Pero como se está realizando el dibujo dentro del lienzo, se puede aprovechar la circunstancia para crear un método de previsualización de lo que se va a imprimir, simplemente presentando el objeto Canvas y llamando a su método paint().

```
class Impresora extends Canvas {
  Frame f;      // Frame padre
```

Como capacidades que se quieren incoporar a la clase están la de porder imprimir cadenas, imprimir líneas completas, cambiar la fuente de caracteres, imprimir en columnas en base a topes de tabulador y, la ya comentada posibilidad de previsualización del contenido antes de realizar la impresión. Para que esto último sea posible, es necesario que la página esté completa antes de que se imprima. Para que todas estas características estén presentes, se ha dotado a la clase Impresora de los métodos adecuados, y para que la previsualización pueda implementarse, los distintos comandos que se envíen a la impresora son almacenados dentro del objeto Impresora y luego reproducidos, bien sobre la pantalla en el caso de la previsualización o sobre la impresora en el caso de querer imprimir. El lector puede incorporar sus propios métodos para que la clase sea capaz de imprimir imágenes o gráficos, por ejemplo.

```
// Inicialización del array de instrucciones
objetos = new Vector();
```

La línea de código anterior es la que, en el constructor de la clase Impresora, crea un objeto Vector, donde se van a ir guardando todas las intrucciones o comandos que se quieran enviar a la impresora. La impresión se realizará recorriendo el contenido del Vector y llamando al método draw() del comando correspondiente. Esto es vital, ya que no se puede saber a priori, cuál es el comando que sigue en la lista, así que hay que dotarlos a todos de la posibilidad de que se impriman sin necesitar de nada más.

```
abstract class objImpresora {
  abstract void draw( Graphics g,Point p );
}
```

Por ello, la clase abstracta define el método draw(), al que hay que indicar el objeto Graphics sobre el que se va a imprimir y la posición en la que se va a iniciar esa impresión, ya sea en la pantalla o directamente en la impresora. Luego, los comandos derivarán de esta clase y proporcionarán una implementación adecuada del método draw(), de acuerdo a las características de la información que van a imprimir. La ventaja adicional que proporciona esta aproximación, aparte de que es un buen diseño orientado a objetos, es que no hay que comprobar el tipo de objeto que se quiere imprimir a la hora de hacerlo, sino que cada uno de los objetos correspondientes a los comandos sabe cómo tiene que imprimirse.

```
for( int i=0; i < objetos.size(); i++ ) {
  imp = (objImpresora)objetos.elementAt( i );
  imp.draw( g,pt );
}
```

Las líneas de código anteriores son las que se encargan de la impresión de todos los objetos de los comandos de impresión. Obsérverse que hay que hacer un moldeo de los elementos del Vector al tipo objImpresora, porque no se sabe el tipo del elemento que se está imprimiendo, él tiene su propia forma de imprimirse, con su implementación particular del método draw(), como se ha indicado.

Otra parte interesante del código del ejemplo, corresponde a las clases que implementan cada uno de los comandos de impresión que va a soportar el objeto Impresora que se ha creado. Cada una de ellas extiende la clase abstracta objImpresora y disponen de un constructor particular en el cual se indica la información que corresponde al comando, ya que el constructor no se puede colocar en la clase abstracta. En el código se indica la acción que realiza cada una de las clases.

```
class printTab extends objImpresora {
  static int tabulador = 0;
  int tab_dist;
  Font tabFnt;

  public printTab( Font tbFont,int tabdist ) {
    tabFnt = tbFont;
    tab_dist = tabdist;
  }

  public void draw( Graphics g,Point p ) {
    if( tabulador == 0 ) {
      tabulador = g.getFontMetrics( tabFnt ).stringWidth( "M" );
    }
    if( p.x < ( tab_dist * tabulador ) ) {
      p.x = tab_dist * tabulador;
    }
  }
```

De las clases, quizá la que merezca un comentario sea la clase que implementa la tabulación, cuyo código reproducen las líneas anteriores. Como se puede observar, esta clase avanza la posición x una cantidad igual al númeroi del ancho del carácter que se le haya indicado. La pregunta que surge es, ¿cuál debe ser el carácter y la fuente de caracteres a utilizar? La respuesta, por supuesto, es que no hay forma de determinar la anchura de un carácter tabulador, así que hay que inventárselas. Para asegurar que siempre se utilice la misma anchura, la variable que indica esta cantidad se hace estática, de forma que todas las copias de la clase tomen como referencia un valor igual. Por defecto, si no se indica ninguno, se toma como tamaño del tabulador la anchura del carácter M en la fuente de

caracteres que se indique, que en principio es la no proporcional. Una vez fijado este valor, el código no volverá a ejecutarse.

Imprimir un tabulador de n posiciones, consistirá simplemente en desplazarse en el eje X ese número de anchuras del carácter tabulador.

Además de los métodos que implementan los comandos, son necesarios un método para el salto de página y otro para el fin del trabajo, que son los métodos saltoPagina() y finImpresion(), respectivamente. Este último, si por negligencia no es llamado, se llama en el método finalize().

```
public void paint( Graphics g ) {
  pt = new Point( 0,0 );
  print( g );
}
```

Y ya, por fin, las líneas anteriores son las que quedan como más interesantes, ya que permiten la previsualización del contenido de los objetos y ver cómo aparecerán (más o menos) en la impresión sobre papel. Como se puede ver, el método es muy simple, consiste en llamar al mismo método print(), utilizando la pantalla en vez de la impresora como objeto. Y, ¿dónde está ese objeto pantalla? Pues se coge del método paint() estándar, que es llamado por el método print().