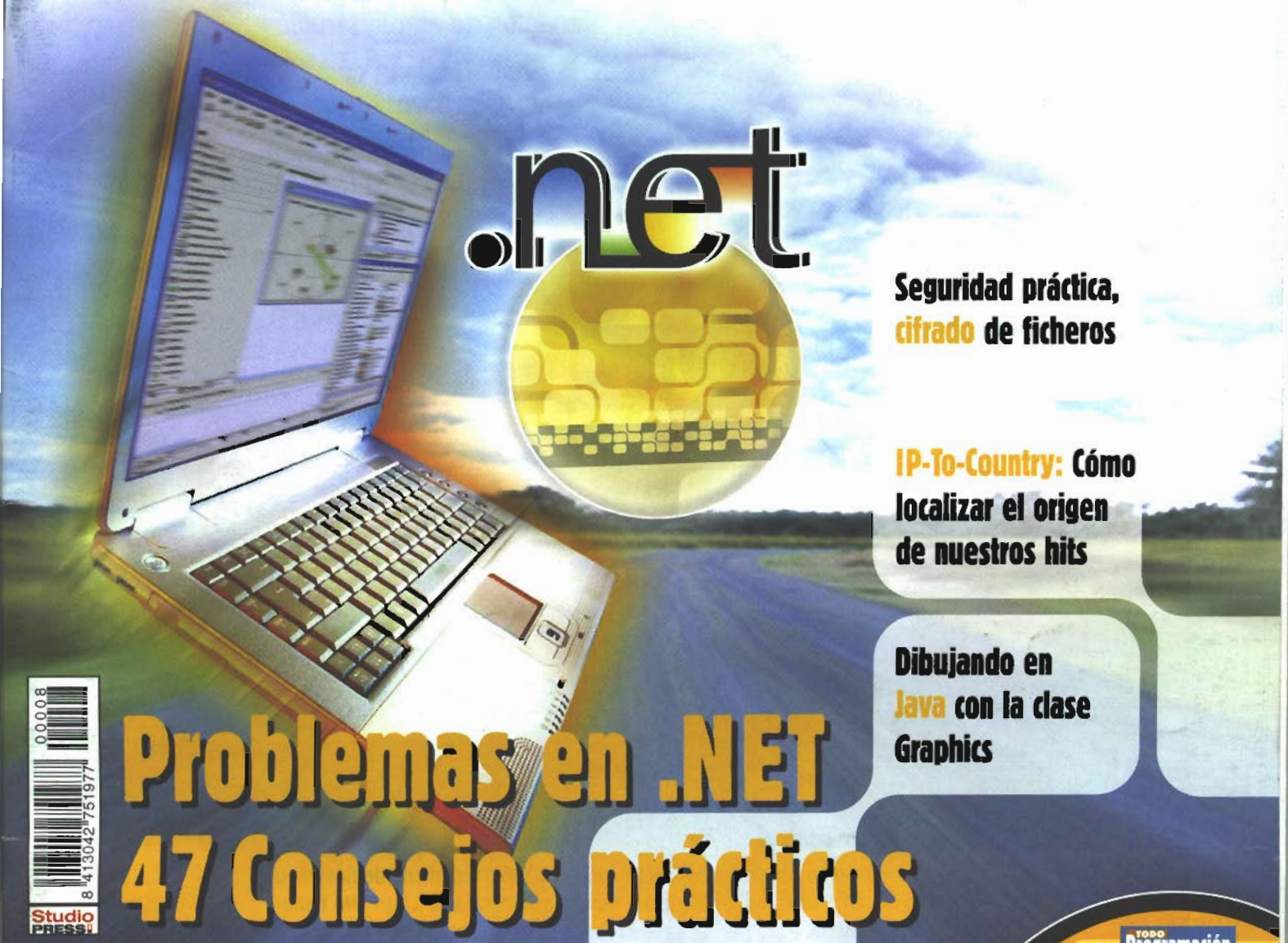
Año 1 • Número 8 • **5,98** euros

TODO Programación

La Revista mensual para entusiastas de la programación

www.iberprensa.com



Seguridad práctica, cifrado de ficheros

IP-To-Country: Cómo localizar el origen de nuestros hits

Dibujando en Java con la clase Graphics

Problemas en .NET
47 Consejos prácticos

Studio PRESS

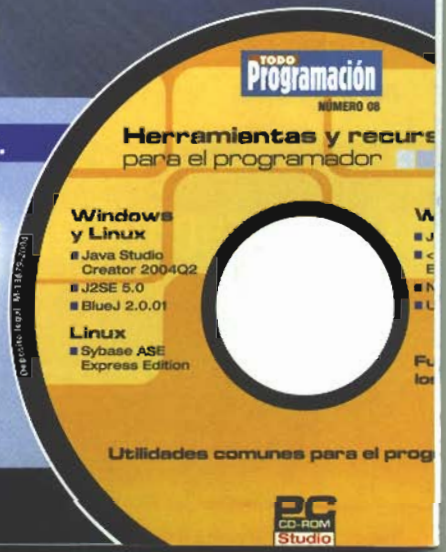
■ CD-ROM ESPECIAL HERRAMIENTAS JAVA: SUN STUDIO CREATOR, J2SE 5.0, BLUEJ, JEXT, ETC.

Zona Linux

- Mono 1.0: Interfaces y enumeraciones en C#
- GNOME: Acceso a Gconf desde programas

Zona Windows

- .NET: Programación con Atributos
- JBuilder 2005 y Studio Creator: Desarrollo Java



JUEGOS JAVA: PROGRAMACIÓN DE JUEGOS EN MIDP

TODO Programación

La Revista mensual para entusiastas de la programación

DIRECTOR
Eduardo Toribio
etoribio@iberprensa.com

REDACCIÓN
Yenifer Trabadela
yenifer@iberprensa.com

COLABORADORES
Antonio M. Zugaldía
(azugaldia@iberprensa.com)
David Santo Orcero
(orcero@iberprensa.com)
Manuel Domínguez
(mdominguez@iberprensacom)
Fernando Escudero
(fescudero@iberprensa.com)
José Manuel Navarro
(jnavarro@iberprensa.com)
Marcos Prieto
(mprieto@iberprensa.com)
Guillermo "el Guille" Som
(elguille@iberprensa.com)
Santiago Márquez
(smarquez@iberprensa.com)
Nacho Prendes
(nprendes@iberprensa.com)
José Rivera
(jrivera@iberprensa.com)
Jaime Anguiano
(janguiano@iberprensa.com)
Alejandro Serrano
(aserrano@iberprensa.com)

DISEÑO PORTADA
Antonio Gª Tomé

MAQUETACIÓN
Antonio Gª Tomé

DIRECTOR DE PRODUCCIÓN
Carlos Peropadre
cperopadre@iberprensa.com

ADMINISTRACIÓN
Marisa Cogorro

SUSCRIPCIONES
Tel: 91 628 02 03
suscripciones@iberprensa.com

FILMACIÓN: Fotpreim Duvial
IMPRESIÓN: I. G. Printone
DUPLICACIÓN CD-ROM: M.P.O.

DISTRIBUCIÓN
S.G.E.L.
Avda. Valdelaparra 29 (Pol. Ind.)
28108 Alcobendas (Madrid)
Tel.: 91 657 69 00

EDITA: Studio Press
www.iberprensa.com
REDACCIÓN, PUBLICIDAD Y ADMINISTRACIÓN
C/ del Río Ter, Nave 13.
Polígono "El Nogal"
28110 Algete (Madrid)
Tel.: 91 628 02 03*
Fax: 91 628 09 35
(Añada 34 si llama desde fuera de España.)

DEPÓSITO LEGAL: M-13679-2004
Número 08 • Año 1

EDITORIAL

Eduardo Toribio

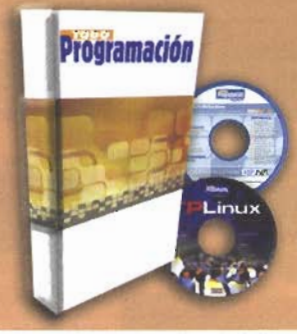
Versiones

Bienvenidos a un nuevo número de **Todo Programación**, estamos en plena época de lanzamientos y los grandes actores de este circo sacan a la arena sus apuestas. Es lo normal y lo habitual, todos estamos ya acostumbrados a la guerra de las versiones. El fabricante presenta el producto, mejor dicho, la revisión anual del mismo, y nos trata de convencer de las numerosas modificaciones y nuevas características que lo hacen mucho más poderoso, estable, rápido, escalable, y un largo etcétera de adjetivos similares. En ocasiones esto es cierto, y empresas serias trabajan duro en ofrecer revisiones mucho más completas que aporten realmente valor a sus clientes, y que por tanto hagan que el desembolso sea justificable. Pero también es una realidad que en demasiadas ocasiones el interés por parte de un fabricante de aumentar su facturación a costa de los *upgrades* de turno suele ser indirectamente proporcional a los intereses reales de sus clientes.

Cambiando de tema, me gustaría llamar la atención sobre el próximo congreso de javaHispano que tendrá lugar en el mes de diciembre, nosotros estaremos allí patrocinando el evento, y creo que puede ser muy útil para todos los interesados en el cada día más amplio mundo del desarrollo de servicios web. Finalmente y para despedirme, agradecer la confianza a los nuevos suscriptores que nos habéis depositado vuestra confianza estos últimos dos meses, esperamos devolver con creces vuestra inversión. Y a los que no os habéis suscrito todavía, os remito a nuestra contraportada donde encontraréis una oferta muy interesante

SUSCRIPCIONES

Como oferta de lanzamiento existe la posibilidad de suscribirse durante un año (12 números) a **Todo Programación** por solo 61 euros lo que significa un ahorro del 15% respecto el precio de portada. Además de regalo se incluye un archivador para coleccionar y guardar las revistas con sus CD-ROMs.
Más información en: www.iberprensa.com



SERVICIO TÉCNICO

Todo Programación dispone de una dirección de correo electrónica y un número de Fax para formular preguntas relativas al funcionamiento del CD-ROM de la revista.
e-mail: todoprogramacion@iberprensa.com
Fax: 91 628 09 35

LECTORES

Comparte con nosotros tu opinión sobre la revista, envíanos tus comentarios, sugerencias, ideas o críticas.

Studio Press
(Todo Programación)
C/ Del Río Ter, Nave 13
Pol. "El Nogal"
28110 Algete. Madrid

DEPARTAMENTO DE PUBLICIDAD

Si le interesa conocer nuestras tarifas de publicidad no dude en ponerse en contacto con nuestro departamento comercial:
- **Tel. 91 628 02 03**
- **e-mail: publicidad@iberprensa.com**

Sumario

Número 8

A quién vamos dirigidos

Todo Programación (TP) es una revista para programadores escrita por programadores y con un enfoque eminentemente práctico. Trataremos de ser útiles al programador, tanto al profesional como al estudiante. Si hay algo cierto en este sector es que nunca podemos parar, vivimos en un continuo proceso de reciclaje. Ahí es donde tratará de encajarse TP: información, actualidad, cursos y prácticas de los lenguajes más demandados y formación en sistemas.

En portada

Migración a .NET

Consejos para la solución de problemas típicos

.net ZONA WINDOWS

Ejemplos y código fuente

Cada CD-ROM de la revista incluye una carpeta denominada *fuentes* en la que se encuentra el material complementario para seguir cada uno de los cursos: por ejemplo, los listados completos, los ejemplos desarrollados en diversos lenguajes, compiladores, editores, utilidades y en general, cualquier herramienta que se mencione o cite en la respectiva sección.
Todo con la finalidad de completar la formación y facilitar el seguimiento de cada artículo por parte del lector.

CONTENIDO DEL CD-ROM

Especial desarrollo Java

64 Este mes, y aprovechando la aparición del nuevo J2SE, centramos nuestro CD en algunas de las herramientas Java más importantes del momento. Destacan Sun Java Studio Creator 2004, J2SE 5, y BlueJ para Windows y Linux. Otras utilidades de desarrollo que se encuentran en el CD son NetXP, un conjunto de utilidades pensadas para el programador de .NET y Unite .NET, un entorno de pruebas también para .NET framework.

TALLER PRÁCTICO

IP-To-Country: localizacion de visitantes >>

35 En muchas ocasiones nos puede interesar conocer el origen geográfico de un visitante del sitio web que desarrollamos, mantenemos o administramos. Hay empresas que prestan tal servicio pero nosotros podemos desarrollarlo con este tutorial.

Dibujando en Java: la clase Graphics >>

39 Estamos acostumbrados a trabajar con entornos gráficos de desarrollo rápido que nos porporcionan todos los elementros visuales que requerimos, pero a veces puede ser necesario modificar o crear alguno nuevo en Java.

Criptografía y seguridad informática >>

43 Comenzamos una miniserie práctica que tiene como objetivo mostrar las líneas maestras de las técnicas más habituales utilizadas en el mundo de la criptografía. Primero una pequeña introducción y nos ponemos manos a la obra.

ZONA LINUX

Configuración de aplicaciones GNOME mediante Gconf >>

18 Gconf es el sistema de configuración de GNOME donde se almacenan las preferencias del usuario. Vamos a ver un ejemplo de cómo manejar sus parámetros desde nuestros programas.

Mono: Interfaces y enumeraciones en C# >>

21 Este mes abordamos el estudio de dos elementos distintos de C#: los interfaces y las enumeraciones, que pueden resultar útiles en diferentes ocasiones.

NOTICIAS

6. Novell Brainshare Europe 2004.
6. Mundo Oracle 2004 en Madrid y Barcelona.
7. Nuevos lanzamientos Sun.
7. Máster en desarrollo de videojuegos.
8. Tecnologías Panda TruPrevent.
8. Máster en plataformas web.
8. javaHispano.net.
8. MicroStrategy 7i.
9. Nueva familia de workstations HP.
9. Videocámara digital IRIS N604.
9. Bull lanza nuevos servidores Escala AIX.

Y ADEMÁS...

Juegos Java: Creación de juegos en MIDP >>

54 Visto cómo programar el interfaz gráfico del juego, pasamos ahora a centrarnos en los conceptos básicos necesarios para programar juegos sobre MIDP.

Bases de datos: Paquetes en Oracle PL/SQL >>

60 En el artículo anterior aprendimos los conocimientos básicos para trabajar con subprogramas. Ahora vamos a estudiar algunas características avanzadas de la definición y uso de subprogramas en PL/SQL.

CUADERNOS DE PRINCIPIANTES

Programación orientada a objetos >>

48 Acabamos nuestro curso para principiantes, hemos aprendido los conceptos básicos de la programación y algunas técnicas que nos facilitarán a partir de ahora nuestro trabajo diario como desarrolladores. Pero antes dedicamos este mes a la denominada POO.

NIVEL DE DIFICULTAD:

MATERIAL INCLUIDO EN EL CD-ROM:
Carpeta Ejemplos

Dibujando sobre lienzos en Java

MANUEL DOMÍNGUEZ
mdominguez@iberprensa.com

Hoy en día todo el mundo que programa en cualquier lenguaje utiliza entornos para el desarrollo rápido de interfaces de usuario, esto facilita mucho el trabajo porque se hace de forma visual el insertado de botones, tablas, barras de desplazamiento, etcétera. Sin embargo, cuando se necesita modificar los componentes gráficos estándar, salirse de lo normal, mucha gente "patina". En este artículo veremos cómo podemos llevarlo a cabo con unos sencillos pasos.

Dibujar en Java requiere los mismos elementos que dibujar un óleo: esto es, una superficie donde pintar, pinceles, colores, un conjunto de primitivas básicas e imaginación. En Java todos estos elementos (a excepción de la imaginación, claro) los proporciona una única clase llamada *Graphics*. Para hacer uso de esta clase debemos importar el paquete *Java awt* en nuestra aplicación. Veamos primero de forma detallada cómo se muestran los componentes graficos en Java, antes de continuar

LA CLASE COMPONENT

La clase *Component* es abstracta y es la superclase de todos los componentes de Java que tienen una representación gráfica. Las subclases que extiendan la clase *Component* hacen uso de un lienzo proporcionado por ésta. En él dibujan el aspecto del componente gráfico en cuestión, por ejemplo, un botón, un panel, una barra de herramientas, etc. Como todo componente es por herencia un *Component*, ofrece a su vez la posibilidad de acceder a su lienzo, el que nosotros deberemos obtener para representar lo que queramos en él. Aunque se podría usar directamente el lienzo proporcionado por *Component*, se suele emplear el de sus clases derivadas, ya que seguramente nos facilitará las cosas. Generalmente se utiliza el lienzo proporcionado por un *JLabel* o un *JPanel* para dibujar sobre él. En el lienzo, las coordenadas de dibujo vienen representadas en píxeles donde (0,0) es el punto de la esquina superior izquierda, y su orden es creciente a medida que nos alejamos de dicha esquina.

Obtener un lienzo

La clase *Graphics* no se puede instanciar porque es abstracta. Sin embargo, se puede obtener una referencia *Graphics* al lienzo de un componente, utilizando el método *Graphics getGraphics()* de dicho componente. Por ejemplo, observemos el código que se puede ver en el cuadro Listado 1 de la página siguiente.

En él tenemos una aplicación que consta de una ventana de tipo *MiVentana* que tiene insertado un panel *PanelDeDibujo* en su interior. Accedemos a dicho panel para obtener una referencia *Graphics* a su zona de dibujo *Lienzo*, y a partir de ahí podríamos dibujar lo que quisiésemos en el panel, a través de la variable *Lienzo*. No lo haremos en este ejemplo sino más adelante.

LA CLASE GRAPHICS

Ya tenemos un lienzo de tipo *Graphics*. Esta clase de Java proporciona una gran cantidad de métodos, donde la mayor parte de ellos tienen como finalidad representar algo en el contexto gráfico que es el *Graphics* en cuestión. En las siguientes líneas veremos las primitivas gráficas más usuales de esta clase, finalizando con un ejemplo

Seleccionar el color frontal

Lo más normal es que queramos seleccionar el color con el que vamos a dibujar en el lienzo. Tanto para rellenar como para el color de las líneas la clase *Graphics* ofrece el mismo método *void (c)*. El parámetro *c* es de tipo *Color*, que está definido en *Java awt.Color* y tiene distintos constructores, alguno de los cuales nos permite especificar cualquier color posible a través de sus componentes RGB. Sin embargo, la mayor parte de las veces no crearemos un color sino que usaremos los que están estáticamente definidos en *Java.awt.Color*, que ya están creados. Los colores más comunes están definidos en esta clase con su nombre en inglés: *BLUE, GREY, RED*... Así, por ejemplo, para decir que deseamos dibujar con el color rojo deberíamos poner el siguiente código, suponiendo que tengamos el lienzo llamado *Lienzo*:

```
Lienzo.setColor(Color.RED);
```

A partir de ese momento, y hasta que seleccionemos otro color para dibujar, todo lo que representemos en el lienzo será dibujado en rojo.

Dibujar líneas

Para dibujar líneas, *Graphics* ofrece el método void (*int x1, int y1, int x2, int y2*). La tónica general a la hora de nombrar los métodos de la clase *Graphics* es llamar como *drawAlgo* al método que dibuje una forma en el lienzo, y *fillAlgo* al que dibuje una forma en el lienzo y la rellene con el color seleccionado. En este caso, este método dibuja un segmento de línea entre dos puntos dados por los parámetros. Los puntos entre los cuales se dibuja la línea sería *P1=(x1,y1)* y *P2=(x2,y2)*. Siendo *x1, y1, x2* e *y2* los parámetros del método. Así, si deseamos trazar en nuestro lienzo una línea entre los puntos *P1=(20, 20)* y *P2=(134,44)*, usaríamos el siguiente código:

```
Lienzo.drawLine(20, 20, 134, 44);
```

Todos los componentes gráficos Java tienen un método llamado Paint

Lo que dibujaría la línea deseada con el color que hubiésemos seleccionado. La clase *Graphics* no permite seleccionar el tipo de línea que queremos dibujar. Así que podríamos cambiar el color de la línea, pero la línea sería continua, de un píxel de ancho y sin suavizar; no solo para dibujo de líneas sino para cualquier otra forma.

Para dibujar un punto también se usa este mismo método, pero poniendo como puntos origen y fin de la línea el mismo valor.

Dibujar formas

Si lo que deseamos es dibujar formas poligonales cerradas como triángulos, pentágonos o cualquier otra, la clase *Graphics* nos ofrece el método *void (int[] xPoints, int[] yPoints, int nPoints)*.Este método acepta por parámetros *xPoints*, que es un vector con las coordenadas X de todos los puntos a dibujar, *yPoints*, que es un vector con las coordenadas Y de los puntos a dibujar y *nPoints* que indica el número de puntos a dibujar y que debe coincidir con la longitud de *xPoints* e *yPoints*. Este método toma los puntos por parámetro y dibuja líneas entre ellos en el mismo orden en el que se encuentren especificados, uniendo finalmente el último punto con el primero. Así consigue el dibujo mediante líneas de un polígono (cerrado, por definición). Para verlo más claro, si deseamos dibujar un triángulo, debemos especificar sus tres vértices, por ejemplo *V1=(10,100)*, *V2=(40,20)* y *V3=(100,130)*. El código necesario para dibujarlo sería:

```
Xs = new int[3];
Xs[0] = 10;
Xs[1] = 40;
Xs[2] = 100;
Ys = new int[3];
Ys[0] = 100;
Ys[1] = 20;
Ys[2] = 130;
Lienzo.drawPolygon(Xs, Ys, 3);
```

Ciertamente es un poco pesado especificar un triángulo de este modo, pero es tremendamente útil y potente para dibujar polígonos con, por ejemplo, 100 vértices. En cualquier caso podemos usar otra versión del método que acepta por parámetros un objeto tipo *Polígon*, pero el trabajo sería el mismo porque en cualquier otro lugar deberíamos haberlo creado y su proceso es igual de tedioso.

Graphics aporta además otros métodos como son *fillPolygon(...)* y *drawPolyline(...)*, ambos con los mismos atributos que drawPolygon(...). El primero de ellos dibuja la misma forma pero rellenándola con el color seleccionado, y el segundo dibuja la misma forma pero sin unir el último vértice con el segundo, quedando por tanto abierta

Dibujar óvalos y círculos

Un círculo es un caso especial de un óvalo, por lo que no hay algo del tipo *drawCircle(.)* en la clase *Graphics*. En cambio hay un método llamado *void (int x, int y, int width, int height)*.Que permite dibujar un óvalo y, en su caso, un círculo. Este método dibuja el óvalo en el interior de un rectángulo imaginario que viene dado por los parámetros. Así *x* e *y* serían la esquina superior izquierda de dicho rectángulo imaginario, y *width* y *heigth* serían el ancho y el alto del mismo, respectivamente. Si el rectángulo imaginario no es un rectángulo sino un cuadrado, el óvalo que se dibuja no es un óvalo, sino un círculo.

En muchos lenguajes, la forma de dibujar un círculo es especificar el punto del centro del mismo y el radio. No hay que confundirse, en este caso no es así puesto que el método es para dibujar un óvalo y éste no puede dibujarse solo con esos parámetros

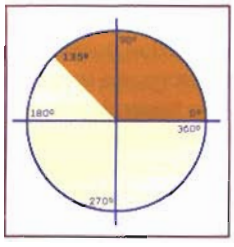
Si queremos dibujar un círculo en el cuadrado imaginario que empieza en (0,0) y que su diámetro sea de 50, usaríamos el siguiente código:

```
Lienzo.drawOval(0, 0, 50, 50);
```

Pero en ningún caso el centro del mismo sería (0,0). En este caso sería (25, 25). La clase *Graphics* ofrece también el método *fillOval(...)* con los mismos atributos que el método que acabamos de ver, pero que además de dibujar el óvalo, lo rellena con el color seleccionado

Dibujar arcos

Un arco en Java también es un poco distinto a un arco en otros lenguajes, del mismo modo que ocurre con los círculos y los óvalos. Volviendo a recordar cómo eran los óvalos, entendemos fácilmente que un arco es un trozo sin cerrar de un óvalo. Por ello, el método que nos ofrece la clase *Graphics* se define de forma muy similar a un óvalo: *void (int x, int y, int width, int height, int startAngle, int arcAngle)*. El significado de *x*, *y*, *width* y *height* es exactamente el mismo que el caso de *drawOval(...)*. *starAngle* y *arcAngle* especifican el ángulo de comienzo y fin del arco. En la figura de arriba se ve más claro cómo especificar los ángulos. Un ejemplo de arco de circunferencia de de 90 grados (desde 0° a 90°) sería como sigue:



Así se toman los grados para dibujar arcos en Java.

```
Lienzo.drawArc(0, 0, 50, 50, 0, 90);
```

Graphics también proporciona el método *fillArc(...)* que, con la misma forma, dibuja un arco relleno del color seleccionado, o sea, la típica porción de las gráficas de tarta.

Listado 1. Ejemplo1.Java

```
package Ejemplo1;
import Java.awt.*;
public class MiVentana extends Javax.swing.JFrame {
public MiVentana() {
initComponents();
PanelDeDibujo.setSize(300, 300);
setResizable(false);
}
private void initComponents() {
// En este método se inician los componentes,
// incluyendo la variable PanelDeDibujo.
// El código se encuentra en el CD.
}
private void exitForm(Java.awt.event.WindowEvent evt) {
System.exit(0);
}
public static void main(String args[]) {
new MiVentana().show();
Graphics Lienzo = PanelDeDibujo.getGraphics();
}
private static Javax.swing.JPanel PanelDeDibujo;
}
```

Dibujar imágenes

Dibujar imágenes en Java es incluso más fácil si cabe que en otros lenguajes de programación. La clase *Graphics*, nuestro lienzo particular, ofrece un método llamado *drawImage(...)* que nos sirve precisamente para eso. Aunque este método está sobrecargado (existen varias versiones) nosotros usaremos la versión *boolean (img, int x, int y, observer)*que es la que más sencilla es de entender. Toda imagen tiene forma de rectángulo. En este método *x* e *y* indican dónde queremos situar la esquina superior izquierda de nuestra imagen en el lienzo; a partir de ahí la imagen se mostrará hacia la derecha y hacia abajo. *Img* es un objeto tipo *Image*, que será la instancia ya cargada de nuestra imagen a mostrar; no es el propósito de este artículo explicar las API AWT y SWING de Java simplemente decir que es fácil obtener un objeto *Image* a partir de un *ImageIcon*. Por último, *observer* de tipo *ImageObserver*, es un objeto que se encarga de controlar cuándo está la imagen disponible, y su uso está particularmente enfocado a la creación de *applets* Java que descarguen imágenes de Internet (por la lentitud que ello conlleva). Si vamos a realizar aplicaciones donde las imágenes están en local, se puede obviar este parámetro y ponerlo a *NULL*.

Supongamos que tenemos un *ImageIcon* llamado *miImagen*. Una forma para mostrarlo en pantalla "a pelo" sería:

```
Lienzo.drawImage(miImagen.getImage(),
10, 10, null);
```

Lo que mostraría la imagen a partir del pixel *P=(10,10)*, en todo su tamaño.

Escribir texto

Para escribir textos simples en cualquier lugar que deseemos, la clase *Graphics* tiene un método sobrecargado llamado *drawString(...)*. De las variantes de este método usaremos *void (str, int x, int y)* que es la más sencilla de entender, rápida y útil. *X* e *y* son las coordenadas donde queremos que aparezca el texto (de ahí hacia la derecha) dentro de nuestro lienzo y *str*, es el texto que deseamos mostrar. Aunque *Graphics* ofrece alguna alternativa para mostrar tipos de letras, estilos, etc , la verdad es que no es su punto fuerte, por lo que dejaremos para un poco más adelante en este artículo el adentrarnos en este tema.

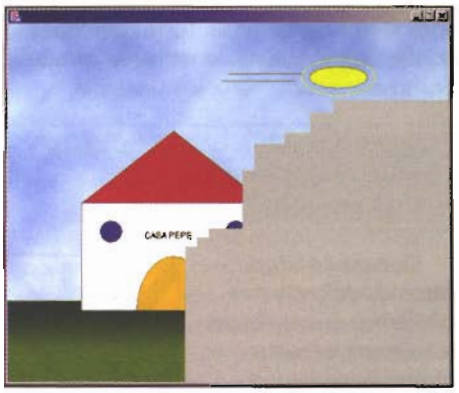
Un ejemplo de uso de este método sería:

```
Lienzo.drawString("¡Hola mundo!",
20, 30);
```

Que mostraría el texto "¡Hola mundo!" con una fuente estándar a partir del pixel de nuestro lienzo *P=(20,30)*

UN PROBLEMA COMÚN

Si hemos hecho las cosas siguiendo las indicaciones anteriores, todo funcionaría correctamente, en el sentido de que las cosas se dibujan donde deben. Sin embargo, si habéis probado los ejemplos, habréis observado que en el ejemplo 2 del CD-ROM, al minimizar la ventana o pasar otra por delante o incluso al iniciar, cuando tiene el ratón en el lugar que aparece la ventana, su contenido se borra. Podemos ver este efecto en la figura de abajo. Esto es debido a cómo se pintan los componentes gráficos, y es común a la mayor parte de los lenguajes de programación.



Si no se sobrescribe el método Paint, la imagen se borra.

El método paint (Graphics G)

Si usamos el *Graphics* de un *JPanel*, el *JPanel* por defecto es un rectángulo gris. Como todos los componentes gráficos Java, tiene un método llamado *void paint(Graphics G)* que el gestor de ventanas se encarga automáticamente de llamar cuando se producen algunas situaciones: pasar una ventana por encima de otra, minimizar, maximizar y restablecer, etc. Así se repinta de nuevo el componente, pero... ¿qué pasa si desde fuera modificamos el aspecto del lienzo? Pues que todo va estupendamente hasta que se vuelve a dibujar el *JPanel* y perdemos lo que hemos dibujado. Es como si cada cinco minutos se pintara una misma pared de blanco. Si nosotros hemos dibujado un triángulo en la pared, se verá bien hasta que pase el pintor y vuelva a pintarla de blanco. En nuestro caso el pintor es el método *void paint(Graphics g)* y pasa cada vez que sea necesario de forma automática. Este método no podemos llamarlo directamente, y además no pertenece a la clase *Graphics*, sino a la clase *Component*. Veremos cómo solucionar este problema un poco más adelante.

El método repaint()

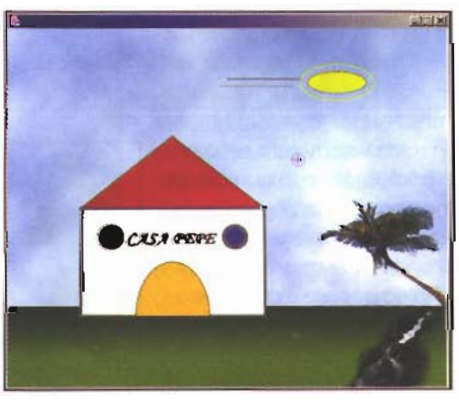
El método *void repaint()*, de la clase *Component*, hace que el gestor de ventanas llame al método *paint(Graphics g)*, ya que nosotros no debemos llamarlo directamente, aunque no haya ninguna razón para hacerlo. Es decir, el método *paint(Graphics g)* se llama de dos formas: automáticamente cuando es necesario, lo cual no podemos controlar, y a nuestra merced, si invocamos al método *repaint()*.

Extender un componente

Sabemos cómo hacer que se ejecute el método *paint(Graphics g)*, pero no cómo conseguir que dibuje lo que deseamos. La única forma que tenemos de hacerlo es modificando el contenido de este método, y para ello tenemos que usar la herencia y la redefinición de métodos. Si queremos usar un *JPanel* para dibujar en su *Graphics* lo que deseamos, y no queremos que se nos borre automáticamente, debemos crear una clase derivada de *JPanel* y en ella redefinir el método *void paint(Graphics g)* para que haga lo que queremos. Ahora cada vez que haga falta repintar todo o cuando pase porque sea necesario, en lugar de ejecutarse el *paint(Graphics g)* del *JPanel*, se ejecutará automáticamente el de la clase que hemos creado, y por tanto lo que hemos dibujado será permanente Los ejemplos 2 y 3 que se incluyen en el CD de la revista utilizan esta técnica, y a partir de ahora consideraremos que ésta es la opción elegida.

LA CLASE GRAPHICS2D

Graphics2D es una especialización (subclase) de *Graphics*. Esto significa que los métodos que hemos visto hasta ahora siguen siendo válidos, pero además se añaden un buen número de importantes mejoras. A continuación veremos las más usadas, aunque son tantas y de tal calibre que podríamos dedicar una serie de diez números para poder comentarlas.



Captura de los ejemplos usando Graphics2D.

Obtener un lienzo Graphics2D

Lo primero de todo, al igual que con *Graphics*, es obtener un lienzo de tipo *Graphics2D*. La forma indicada por Javasoft para ello es exactamente igual que la que ya hemos visto, y únicamente hay que hacer un *casting* de tipos para obtener un *Graphics2D*. Según lo explicado al comienzo de este artículo, sería:

```
Graphics2D Lienzo = (Graphics2D)
UnComponente.getGraphics();
```

Donde *UnComponente* puede ser un *JLabel*, un *JPanel*, o cualquiera de los componentes gráficos de la API de Java. Si estamos usando la opción de redefinir el método *paint(Graphics g)* tras heredar de un componente existente, se tendría que hacer esto dentro de este método:

```
void paint(Graphics g) {
  Graphics2D Lienzo = (Graphics2D) g
  // A partir de aquí, usamos
  Lienzo.loQueSea(…);
}
```

Esta última opción es la mejor y la que se seguirá en los ejemplos.

Activar el suavizado

Sin duda, una de las mejoras estrella de todas las que incorpora *Graphics2D* es la posibilidad de añadir suavizado al lienzo y, por tanto, a todo lo que en él aparezca; esto significa perder de vista de un plumazo el "efecto escalera" de letras, líneas, formas, etc. ¿Quién no ha visto alguna vez el espantoso efecto de dibujar un círculo sin suavizado? Con *Graphics2D*, esto se acabó. Para activar el suavizado hay que usar un nuevo método de la clase *Graphics2D*: (hintKey, hintValue)que permite especificar opciones de renderizado (*hintKey*) y su valor (*hitValue*). Concretamente la línea exacta que hay que escribir para activar el suavizado es, tal cual, la que sigue:

```
Lienzo.setRenderingHint
(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
```

Una vez puesto esto en el código, todo lo que dibujemos en el lienzo estará suavizado. Es importante comprender que mayor calidad solo es posible a costa de perder rendimiento y rapidez, por lo que habrá que sopesar si se desea o no activar el suavizado.

Letra normal y suavizada.

Seleccionar el tipo de línea

Ya vimos que la clase *Graphics* no permitía modificar el tipo de línea que queríamos usar para las formas que dibujáramos. *Graphics2D* ha mejorado mucho en este aspecto; permite seleccionar el grosor de la línea, definir si es continua o discontinua, el periodo de discontinuidad, cómo serán los extremos de la línea e incluso cómo deben realizarse las uniones entre líneas cuando se dibujan polígonos. En cualquier caso, las líneas se dibujan de la misma forma que para *Graphics*. Simplemente hay que definir antes de ello las características de la línea, del mismo modo que hacemos con el color. Para ello *Graphics2D* nos proporciona el método *void (s)*. El único parámetro de este método es *s*, de tipo *Stroke*. *BasicStroke* es la única clase que implementa el interfaz *Stroke* y su cometido es el de configurar parámetros de las líneas. Tiene diversos constructores que no comentaremos aquí por su extensión. Veamos un ejemplo:

```
Lienzo.setStroke(new BasicStroke(4f,
BasicStroke.CAP_ROUND,
BasicStroke.JOIN_MITER));
```

Con este código estamos diciendo que queremos que al dibujar, la línea sea de cuatro píxeles de gruesa, el extremo de las líneas sea redondeado y las uniones entre líneas terminen en pico. En la figura de la derecha se muestra los tipos de terminaciones y uniones.

Distintas uniones y terminaciones de línea.

Seleccionar el tipo de letra y mostrar texto

La representación de texto es una de las mejoras importantísimas de Java en su clase *Graphics2D*. Se permite seleccionar cualquier fuente instalada en el sistema, estilo, o tamaño; que el texto se ajuste a una forma especificada, etcétera. Todo lo relativo al texto es un poco enrevesado y requiere una gran explicación para la que no tenemos espacio así que comentaremos lo más importante. Debemos elegir primero el tipo de letra que deseamos utilizar. Para ello creamos un objeto de tipo *Font* cuyo constructor permite especificar el nombre de una fuente *TrueType* instalada en el sistema, y características de la misma (cursiva, negrita...). Como en el siguiente ejemplo:

```
Font tipoLetra = new Font("Arial",
Font.BOLD | Font.ITALIC, 27);
```

Donde se elige el tipo de letra Arial, cursiva y negrita en tamaño 27. Luego tenemos que crear un contexto de renderizado, que se encargará de interpretar cómo ha de verse el texto final. Esto se hace obteniendo una instancia de *FontRendererContext*, que nos la proporciona el método *getFontRendererContext()* de la clase *Graphics2D*. Veamos cómo.

```
FontRenderContext frc =
Lienzo.getFontRenderContext();
```

Esto nos proporciona un contexto de renderizado asociado a nuestro lienzo *Graphics2D*. Ambas cosas, el tipo de letra y el contexto, se lo debemos pasar ahora a una capa de texto que debemos crear, y que será la que en última instancia contendrá la representación final del texto. Una capa de texto es del tipo *TextLayout* y su constructor nos pedirá el texto que deseamos escribir, el tipo de letra y un contexto de renderizado. Veamos el código.

```
TextLayout capaTexto =
new TextLayout("¡Hola mundo!",
tipoLetra, frc);
```

Así que tenemos una capa de texto cuyo contenido es el texto "¡Hola mundo!" escrito en Anal, negrita y cursiva de tamaño 27. Si además el lienzo tenía activado el suavizado, el texto presentará este efecto. Solo nos queda mostrar en el lienzo la capa de texto y todo habrá finalizado. Esta vez no es el lienzo (*Graphics2D*) el que tiene un método para mostrar la capa de texto, sino que es la capa de texto la que tiene un método para escribir en un lienzo; el método es *draw(Graphics2D g, int x, int y)*. Donde *g* es el lienzo donde queremos presentar el texto y *x* e *y* las coordenadas donde éste debe aparecer. Por ejemplo:

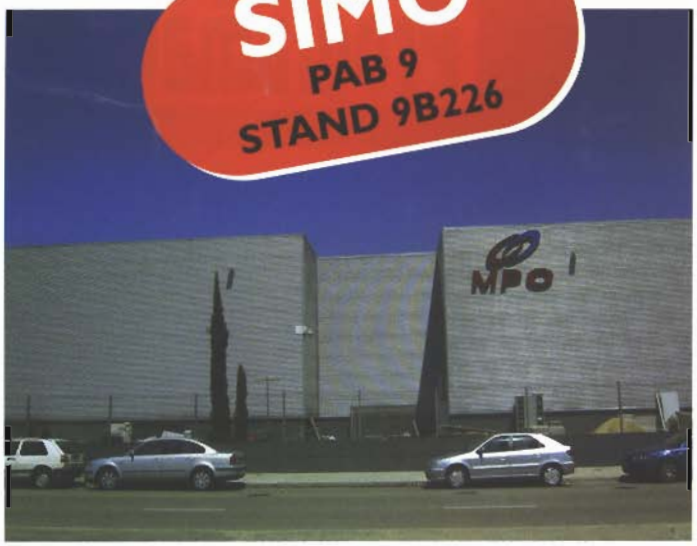
```
capaTexto.draw(Lienzo, 100, 100);
```

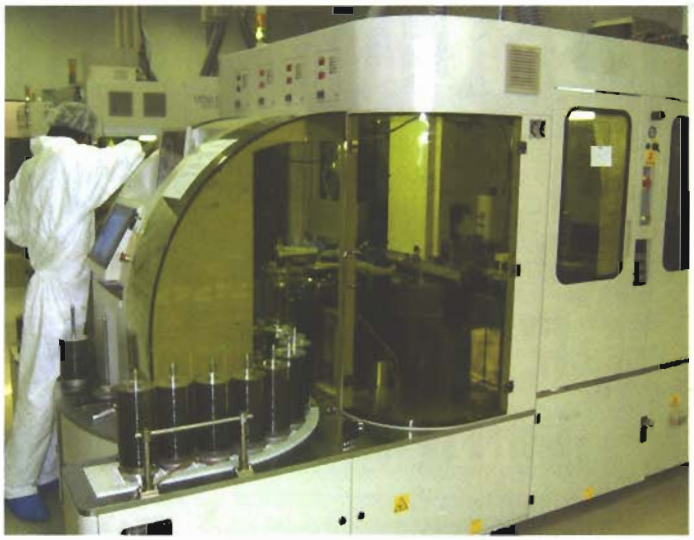
Lo que mostrará la capa de texto en nuestro lienzo a partir del píxel *P=(100,100)*.

CONCLUSIONES

Java ofrece muchas facilidades para crear software que no haga uso de los interfaces comunes y estándares. Por ejemplo, si queremos crear un juego, dibujar objetos, etc., Java facilita mucho la tarea. Han quedado algunas cosas por hacer, por lo que animo a lector a estudiar más detalladamente la clase *Graphics* y observar características como las transparencias, los degradados, las curvas Bézier,... que harán que sus programas además de ser portables y robustos, sean orginales.

SIMO
PAB 9
STAND 9B226





DISTRIBUCIÓN Y LOGÍSTICA

MPO

DUPLICACIÓN DVD & CD

MPO LOGÍSTICA: C/ EBANISTAS 12, POL. IND. URTINSA - 28925 ALCORCÓN - MADRID
TLF : 91 643 12 38 - FAX : 91 641 27 66 - E-MAIL : infologistica@mpo.es