

# Interfaces de Usuario Gráficas

## Librerías para crear Interfaces de Usuario Gráficas

- AWT y Swing

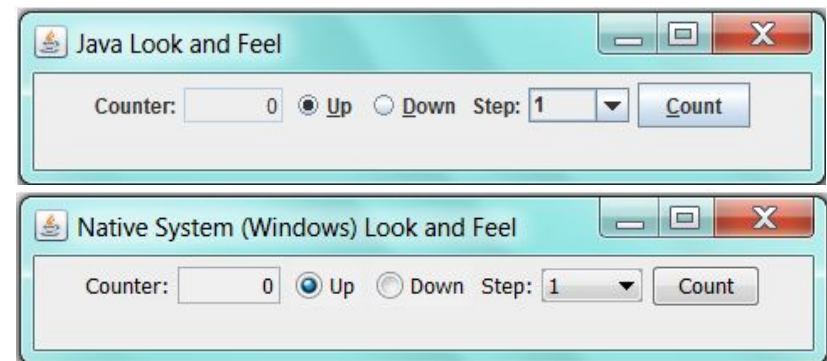
### Abstract Window Toolkit (AWT)

- La clase Component
- La clase Container
- Las interfaces LayoutManager y LayoutMnager2
- Layout Managers
- Contenedores y componentes de GUI
- La clase Graphics

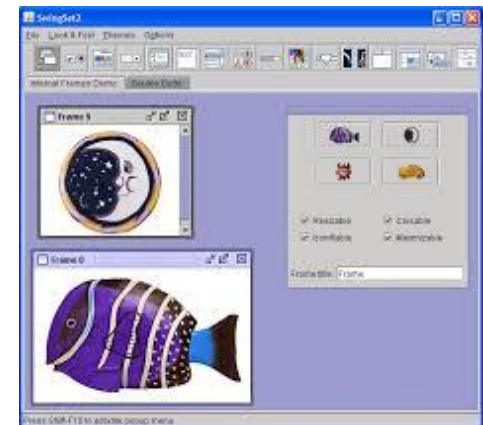
# AWT y Swing

La mayoría de las interacciones entre un usuario y una aplicación ocurren a través de una GUI (Graphical User Interface). Las GUIs proveen un medio para que el usuario ingrese datos al programa y para que el programa muestre datos. En la plataforma estándar de JAVA existen dos librerías para crear GUIs: AWT y Swing.

**AWT** es el primer *framework* JAVA multiplataforma para escribir interfaces gráficas de usuario. Fue introducido en el JDK 1.0 y está basado en las componentes nativas del Sistema Operativo. Brinda un conjunto limitado de componentes textuales, no personalizables y que retienen el *look&feel* de la plataforma donde ejecutan.



**Swing** es una librería mucho más integral, que mejora AWT, fue incorporada a la plataforma como parte de la Java Foundation Classes (JFC), a partir del JDK 1.1. Provee una enorme colección de componentes que facilitan la creación de Interfaces de Usuario de alta calidad. El *look&feel* de las aplicaciones Swing puede configurarse y no depende de la plataforma subyacente.



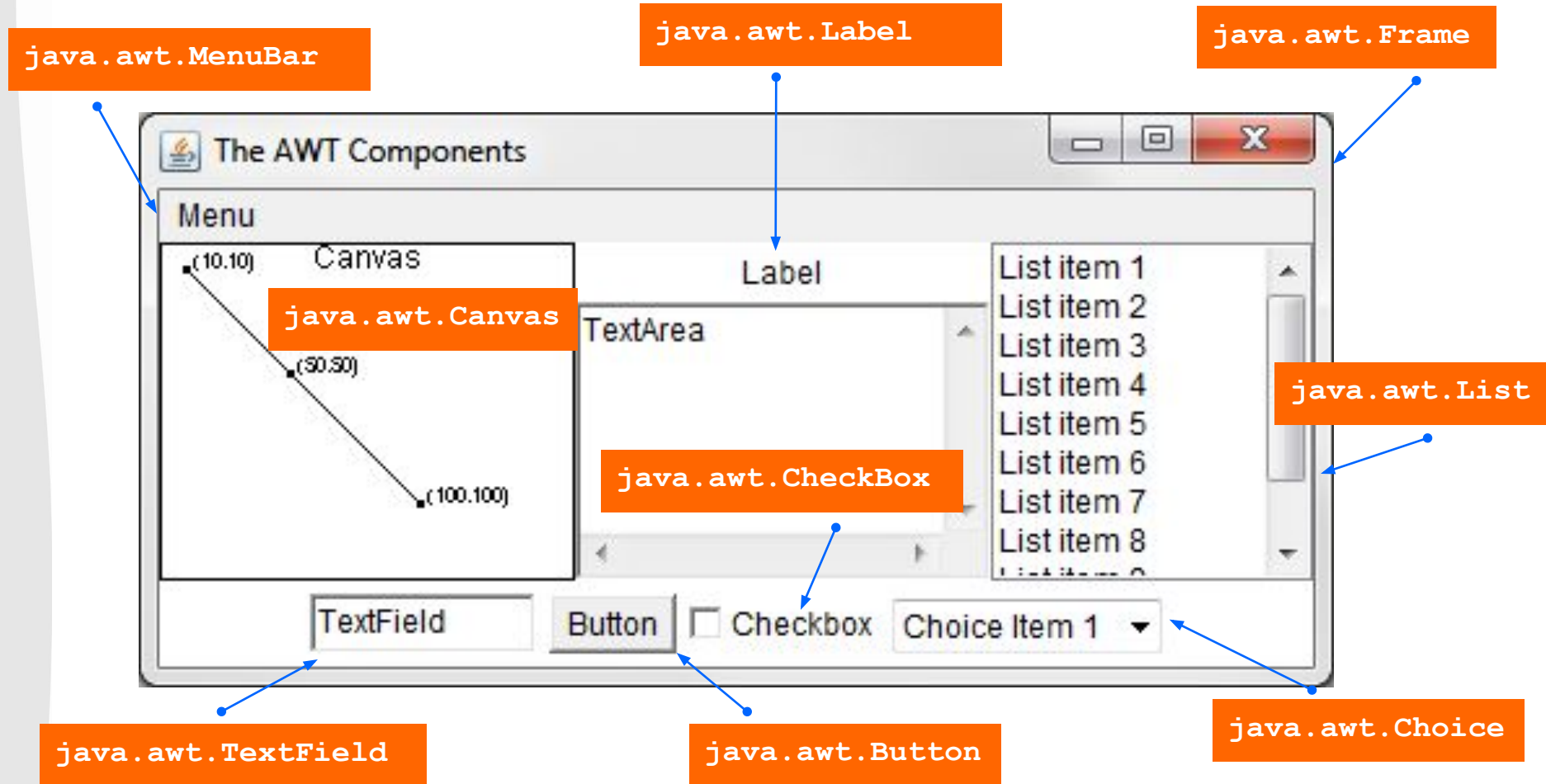
# Abstract Window Toolkit -AWT-

El AWT es el primer *framework* JAVA multiplataforma para escribir interfaces gráficas de usuario. Está basado en las componentes nativas del Sistema Operativo. Aproximadamente la mitad de las clases del AWT extienden de la clase `java.awt.Component`.

El fundamento del AWT lo constituyen:

- La clase `Component`: es una clase abstracta que agrupa componentes GUI estándares como botones, menús, listas, etiquetas, etc. Declara los atributos y comportamientos comunes de todas las subclases de `Component`.
- La clase `Container`: es una clase abstracta, subclase de `Component`. Tiene la capacidad de contener múltiples componentes. `Applet`, `Panel`, `Window`, `Dialog` y `Frame` son subclases de `Container`. Es la superclase de todas las componentes de nivel de ventana.
- Las interfaces `LayoutManager`, `LayoutManager2`: definen métodos para la ubicación, la distribución y cambio de tamaño de las componentes dentro de un contenedor. La API provee clases que implementan estas interfaces.
- La clase `Graphics`: es una clase abstracta que define métodos para realizar operaciones gráficas sobre una componente (mostrar imágenes, texto, establecer colores y fonts). Toda componente AWT tiene asociado un objeto `Graphics` donde dibujarse.

# Abstract Window Toolkit -AWT- Componentes estándares



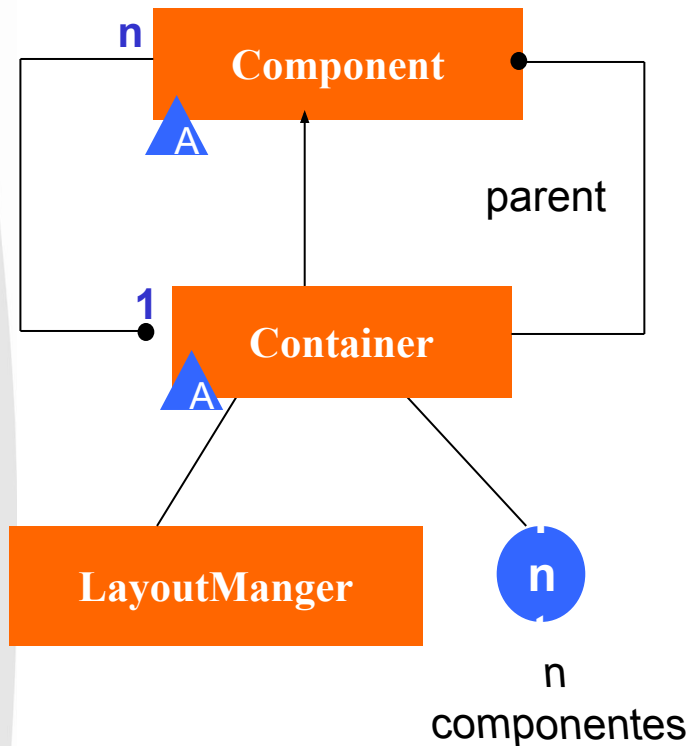
# Abstract Window Toolkit -AWT- Componentes

La clase `java.awt.Component` es una clase abstracta que encapsula la funcionalidad común de todas las componentes AWT.

Todas las componentes AWT como botones, listas, campos de texto, etc. son subclases de la clase `java.awt.Component`. Cada componente AWT tiene asociada la siguiente información:

- Un objeto Graphics (donde dibujarse)
- Posición
- Tamaño
- Peer nativo (componente de GUI del sistema nativo)
- Contenedor padre
- Fonts (tipos y tamaños de letras)
- Colores de fondo y de frente
- Tamaño mínimo, máximo y preferido

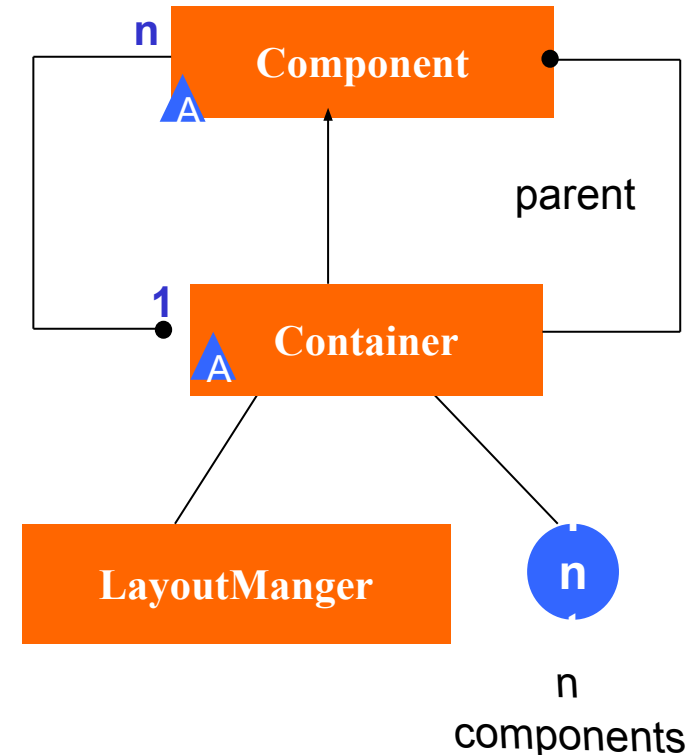
# Componentes, Contenedores y Administradores de disposición



- AWT establece una relación simple y fundamental entre objetos **Component** y **Container**: un contenedor puede contener múltiples componentes.
- Cada objeto **Component** tiene un **Container** padre.
- Todos los *contenedores tienen asociado un layout manager* que establece la ubicación y el tamaño de las componentes que aloja. Cada *contenedor* tiene asociado un único *layout manager*.
- Las responsabilidades del administrador de disposición, las definen las interfaces **java.awt.LayoutManger** y **java.awt.LayoutManger2**

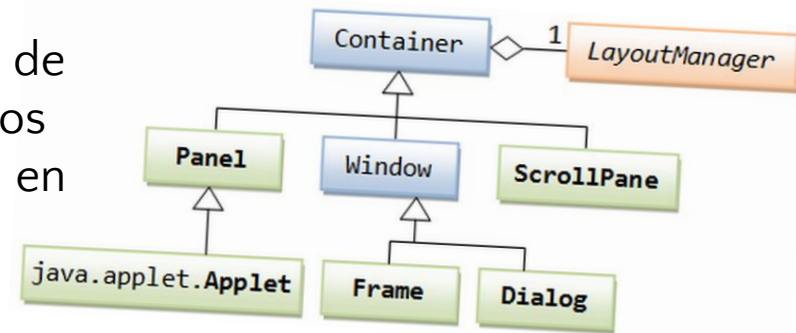
# Contenedores AWT

- La clase `java.awt.Container` es una clase abstracta, subclase de `java.awt.Component`. Los objetos `Container` son simplemente componentes AWT que pueden contener otras componentes. Hay que tener en cuenta que los contenedores pueden contener otros contenedores ya que un objeto `Container` es un `Component`.
- Todo objeto `Container` tiene una referencia a un objeto `LayoutManager`, en el que delega la responsabilidad de ubicar y distribuir a las componentes que aloja. Cada vez que ocurre un evento que provoca que el contenedor tenga que acomodar sus componentes (por ej. cambió el tamaño de la ventana), el objeto *layout manager* del contenedor es invocado.



# Contenedores AWT

Cada programa con GUI tiene un contenedor de nivel superior o *top-level container*. Los contenedores de nivel superior más usados en AWT son: **Frame**, **Dialog** y **Applet**.



Contender	Descripción
<b>Applet</b>	Es subclase de Panel. Es la superclase de todas los applets.
<b>Dialog</b>	Puede ser modal o no-modal. Tiene borde y puede agrandarse, achicarse o minimizarse. No tiene barra de menús.
<b>Frame</b>	Es el contenedor de las aplicaciones de escritorio. Tiene borde, puede contener un menubar y agrandarse, achicarse o minimizarse. Tiene barra de título.
<b>Panel</b>	Es un contenedor simple. Es un área rectangular dentro de la que se pueden incluir componentes de GUI. Se debe ubicar adentro de un contenedor de nivel superior
<b>ScrollPane</b>	Permite hacer un desplazamiento de una componente a través de barras .
<b>Window</b>	Es el contenedor más simple, no tiene barra de menú, ni borde, ni barra de título y no puede agrandarse ni achicarse.

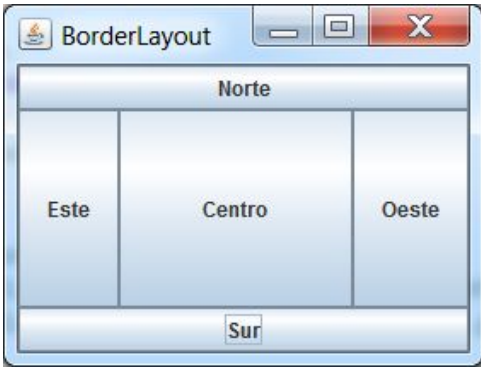
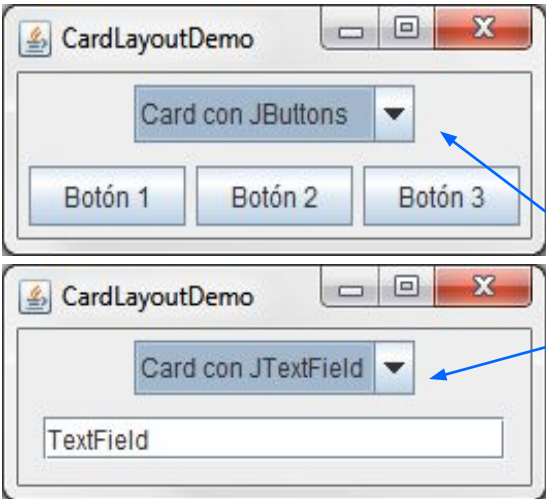


# Layout Managers

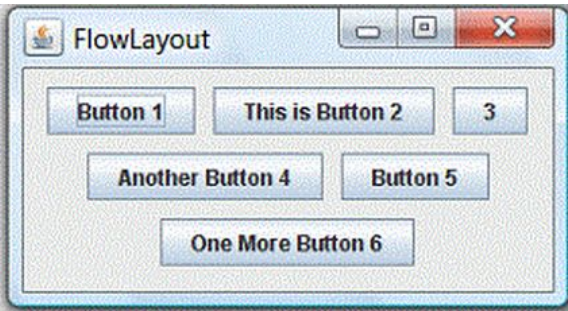
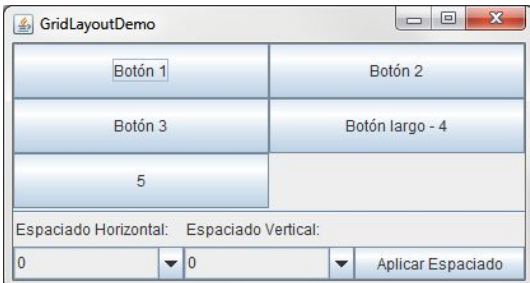
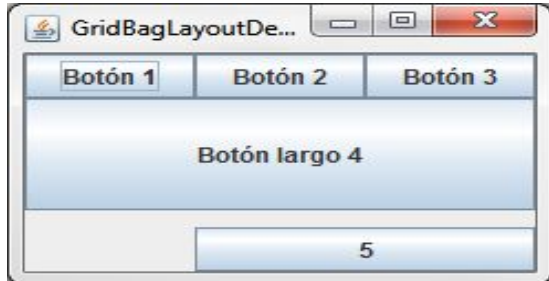
- Los contenedores delegan en el *layout manager* todo lo relacionado a la ubicación, tamaño, espaciado y distribución de sus componentes en la pantalla. Las interfaces **LayoutManager** y **LayoutManager2** definen métodos para calcular tamaños mínimos y preferidos de sus contenedores y ubicar las componentes en un contenedor.
- Dentro de la API de JAVA existen múltiples implementaciones de las interfaces **LayoutManager** y **LayoutManager2**. Las más usadas son: **BorderLayout**, **CardLayout**, **FlowLayout**, **GridBagLayout** y **GridLayout**.
- **LayoutManager2** es subinterface de **LayoutManager** y define métodos que permiten establecer restricciones sobre las componentes
- Un contenedor puede *setear* su layout manager invocando al método **setLayout(LayoutManager)**. También es posible crear Layout Managers especiales implementando algunas de las interfaces o establecer un **layout manager** en **null** y posicionar cada componente de GUI explícitamente.

# Layout Managers

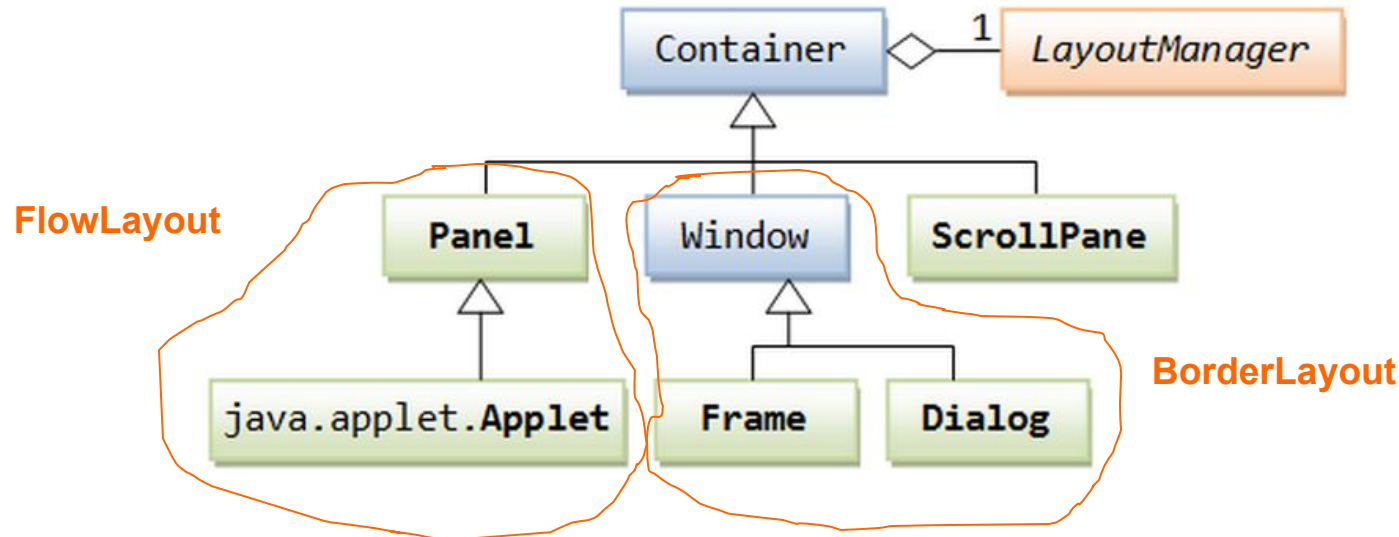
La tabla describe los *Layout Managers* de la API de Java más comúnmente usados

Contenedor	Descripción
<p><b>BorderLayout</b></p>  A screenshot of a Java Swing window titled 'BorderLayout'. The window is divided into five regions: 'Norte' (North) at the top, 'Este' (East) on the left, 'Centro' (Center) in the middle, 'Oeste' (West) on the right, and 'Sur' (South) at the bottom. Each region contains a small rectangular component. <p>Es el <i>layout manager</i> predeterminado de objetos <b>Window</b>. Acomoda y redimensiona a las componentes en cinco regiones: norte, sur, este, oeste y centro. Cada región puede contener sólo una componente y está identificada por una de las siguientes constantes de la clase <b>BorderLayout</b>: <b>NORTH</b>, <b>SOUTH</b>, <b>EAST</b>, <b>WEST</b>, <b>CENTER</b>.</p> <p>Las componentes se agregan al Contenedor especificando la zona.</p> <pre>public void agregarComponentes(Container panel) {     panel.setLayout(new BorderLayout());     Button boton = new Button("Norte");     panel.add(boton, BorderLayout.NORTH);     boton = new JButton("Centro");     panel.add(boton, BorderLayout.CENTER);     ... }</pre>	
<p><b>CardLayout</b></p>  Two screenshots of a Java Swing window titled 'CardLayoutDemo'. The top screenshot shows a dropdown menu with 'Card con JButtons' selected, and three buttons labeled 'Botón 1', 'Botón 2', and 'Botón 3' are visible. The bottom screenshot shows the same window with the dropdown menu set to 'Card con JTextField', and a text field labeled 'TextField' is visible. Blue arrows point from the text 'Se puede programar para que se muestre card1 o card2 en diferentes momentos.' to the dropdown menus in both screenshots. <p>Es usado para contener en una misma área distintos componentes en momentos diferentes. El contenedor actúa como una pila y en cada momento, solo una componente de la pila puede ser visualizada. En general, las componentes de la pila son instancias de <b>JPanel</b>.</p> <p>Se puede programar para que se muestre <b>card1</b> o <b>card2</b> en diferentes momentos.</p> <pre>public void agregarComponentes(Container panel) {     panel.setLayout(new CardLayout());     JPanel card1 = new JPanel();     card1.setName("botones");     card1.add(new JButton("Botón 1")); // idem 2 y 3     JPanel card2 = new JPanel();     card2.setName("texto");     card2.add(new JTextField("TextField", 20));     panel.add(card1, "botones");     panel.add(card2, "texto");     ... }</pre>	

# Layout Manager (cont.)

Contenedor	Descripción
<p><b>FlowLayout</b></p> 	<p>Es el <i>layout manager</i> predeterminado de objetos <code>Panel</code>. Ubica las componentes de izquierda a derecha, usando nuevas filas si es necesario. Es posible establecer la espaciado entre componentes y la alienación de las mismas.</p> <pre>public void agregarComponentes(Container panel) {     panel.setLayout(new FlowLayout());     Button boton = new Button("Boton 1");     panel.add(boton);     boton = new JButton("Boton 2");     panel.add(boton);     ... }</pre>
<p><b>GridLayout</b></p> 	<p>Divide el contenedor en una cuadrícula de manera que las componentes se ubican en filas y columnas. Todas las celdas tienen igual tamaño.</p> <pre>public void agregarComponentes(Container panel) {     panel.setLayout(new GridLayout(2,3));     panel.add(new JButton("Botón 1"));     panel.add(new JButton("Botón 2"));     panel.add(new JButton("Botón 3"));     panel.add(new JButton("Botón largo-4"));     panel.add(new JButton("5"));     ... }</pre>
<p><b>GridBagLayout</b></p> 	<p>Es el layout más sofisticado y flexible. Ubica las componentes en una grilla de filas y columnas permitiendo que algunas componentes usen más de una celda. No todas las filas tienen la misma altura ni todas las columnas el mismo ancho.</p>

# Contenedores AWT y Layout Managers



Cada **Contenedor** tiene asociado un objeto **Layout Manager** por defecto. Como alternativa:

- Es posible establecer un **Layout Manager** en `null`. En este caso se debe definir un tamaño para cada componente y posicionarlas explícitamente en el contenedor. Un **layout manager** en `null` no puede reaccionar ante los cambios, por ejemplo, de tamaño de las componentes. En estos casos, el algoritmo de posicionamiento de las componentes tiene que escribirse en el contenedor.
- Se puede implementar un **Layout Manager** propio implementando alguna de las interfaces: **LayoutManager** y **LayoutManager2**.

# AWT

## Paquetes de la API y clases básicas

AWT tiene varios paquetes, sin embargo los dos más utilizados por los programadores son dos: `java.awt` y `java.awt.event`

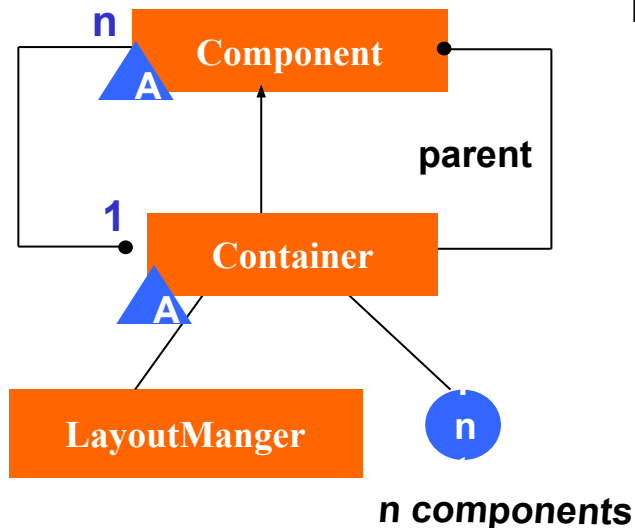
- El paquete `java.awt` contiene las clases gráficas principales de AWT:
  - Las componentes de GUI (como `Button`, `TextField`, y `Label`)
  - Los contenedores (como `Frame`, `Panel`, `Dialog` y `ScrollPane`)
  - Layout managers (como `FlowLayout`, `BorderLayout` y `GridLayout`),
  - Clases para gráficos customizados (como `Graphics`, `Color` y `Font`).
- El paquete `java.awt.event` soporta el manejo de eventos:
  - Clases que modelan eventos de la interfaz (`ActionEvent`, `MouseEvent`, `KeyEvent`, `WindowEvent`, etc.)
  - Interfaces que manejan los eventos de la interfaz (`ActionListener`, `MouseListener`, `KeyListener`, etc.)
  - Clases que implementan interfaces que manejan eventos (`MouseAdapter`, `KeyAdapter`, `WindowAdapter`, etc.)

AWT facilita la escritura de programas con GUI que corran sobre todas las plataformas como Windows, Mac, and Linux, independientemente del dispositivo que se utilice.

# AWT

## Paquetes de la API y clases básicas

Como se ha mencionado, la clase **Container** es una clase abstracta, subclase de **Component** que tiene la capacidad de contener múltiples componentes, entre ellos contenedores. Los contenedores responden a métodos que permiten agregar, eliminar y recuperar componentes, establecer, recuperar o configurar su objeto **LayoutManager**, entre otras funcionalidades.



Los métodos más utilizados de la clase **Container** son:

```
public Component add(String name, Component comp)
public Component add(Component comp, int index)
public Component add(Component comp)
```

```
public void remove(int index)
public void remove(Component comp)
public void removeAll()
```

```
public LayoutManager getLayout()
public void setLayout(LayoutManager mgr)
```

# AWT

## Paquetes de la API y clases básicas – Un ejemplo

Este es un ejemplo de una aplicación con GUI simple, sin manejo de eventos:

```
package interfaces.graficas;
import java.awt.*;

public class ContadorAWT extends Frame {
    private Label lblContador;    // Declara una componente Label
    private TextField tfContador; // Declara una componente TextField
    private Button btnContador;   // Declara una componente Button
    private int Contador = 0;     // Se declara e inicializa Contadorador

    // El constructor se suele usar para configurar las componentes de GUI
    public ContadorAWT() {
        setLayout(new FlowLayout()); // this.setLayout(new FlowLayout());
        lblContador = new Label("Contador");
        add(lblContador);
        tfContador = new TextField("0", 10);
        tfContador.setEditable(false);
        add(tfContador);
        btnContador = new Button("Contador");
        add(btnContador);
        setTitle("Contador AWT");
        setSize(250, 100);
        setVisible(true);
    }

    public static void main(String[] args) {
        ContadorAWT app = new ContadorAWT();
    }
}
```



Se cambia el tamaño de la ventana



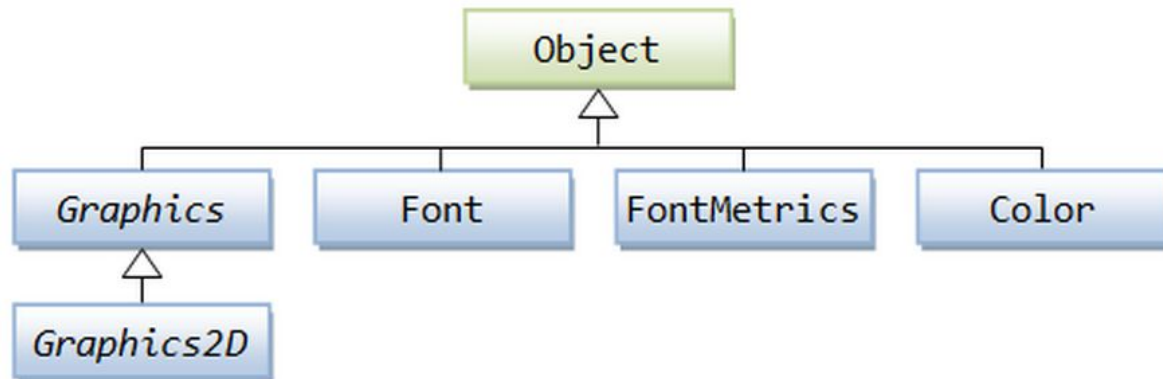
Se re-acomodan las componentes automáticamente



# La clase `Graphics`: contexto gráfico y pintado customizado

Un contexto gráfico provee capacidades para dibujar en la pantalla. Un contexto mantiene un *estado*, como por ejemplo el color y el font, usados para dibujar o escribir (*drawing*), así como también funcionalidades para interactuar con el SO para lograr esa escritura o dibujado.

En Java, el pintado es hecho a través de la clase `java.awt.Graphics`, la cual maneja un contexto gráfico, y provee un conjunto de métodos para dibujar texto, figuras, imágenes sobre diferentes plataformas. La clase `Graphics` es una clase abstracta, y cada plataforma provee una subclase de `Graphics` para hacer el dibujado bajo esa plataforma, pero conforme a la especificación definida en `Graphics`.





# La clase Graphics: métodos para dibujado

La clase **Graphics** provee métodos para dibujar tres tipos de objetos gráficos:

- **Textos o strings**: vía el método `drawString()`. Notar que `System.out.println()` imprime a la salida estándar no a la pantalla.
- **Formas geométricas** (como gráficos vectoriales): vía los métodos `drawXxx()` y `fillXxx()`, donde Xxx podría ser una línea, un rectángulo, un óvalo, un círculo, etc.
- **Imágenes** (como mapa de bits - bitmap): vía el método `drawImage()`.

Algunos de los métodos de **Graphics** son:

```
// Dibujando (o imprimiendo) texto sobre la pantalla gráfica
drawString(String str, int xBaselineLeft, int yBaselineLeft);
```

```
// Dibujando figuras
```

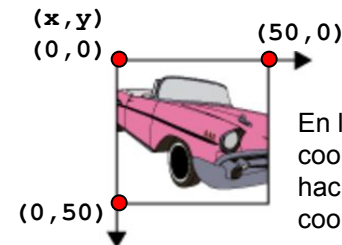
```
drawLine(int x1, int y1, int x2, int y2);
draw3DRect(int xTopLeft, int yTopLeft, int width, int height, boolean raised);
```

```
// Dibujando Figuras rellenas
```

```
fillRect(int xTopLeft, int yTopLeft, int width, int height);
fillOval(int xTopLeft, int yTopLeft, int width, int height);
fillPolygon(int[] xPoints, int[] yPoints, int numPoint);
```

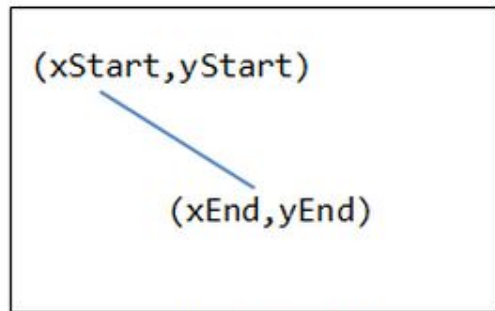
```
// Dibujando (o mostrando) imágenes:
```

```
drawImage(Image img, int xTopLeft, int yTopLeft, ImageObserver obs);
drawImage(Image img, int xTopLeft, int yTopLeft, int width, int height, ImageObserver o);
```

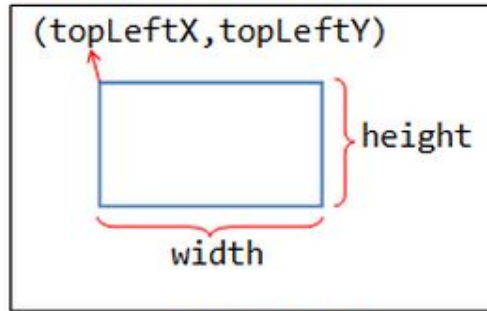


En la pantalla, la coordenada *x* incrementa hacia la derecha y la coordenada *y* hacia abajo

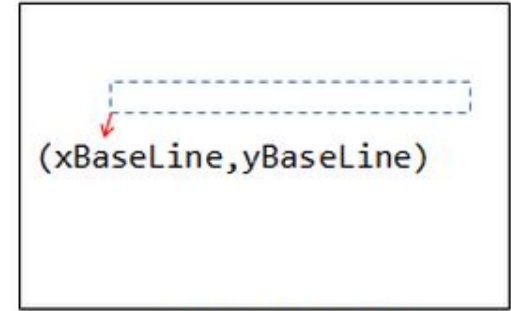
# La clase Graphics: métodos para dibujado de textos y figuras



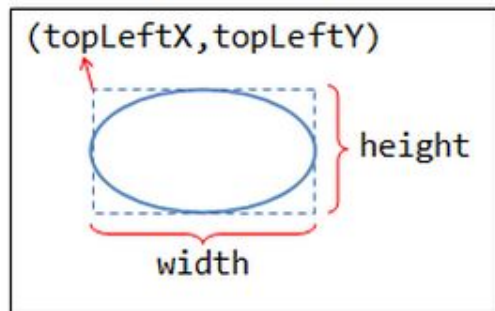
**drawLine()**



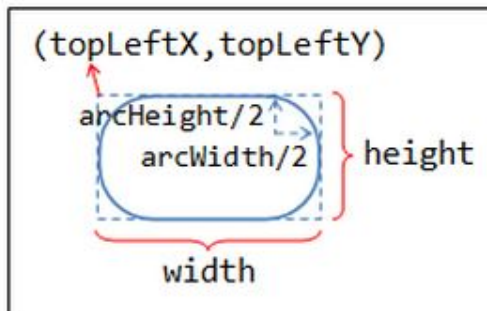
**drawRect()**



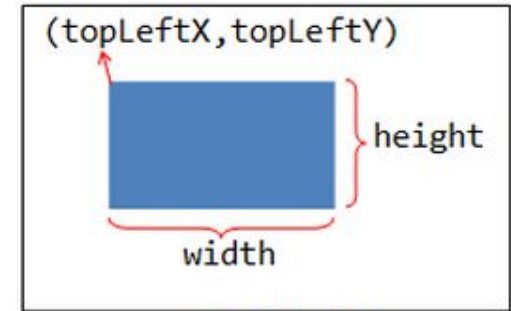
**drawString()**



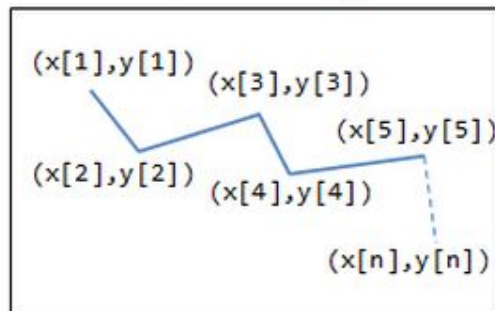
**drawOval()**



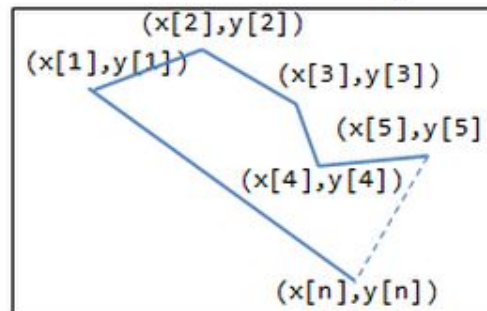
**drawRoundRect()**



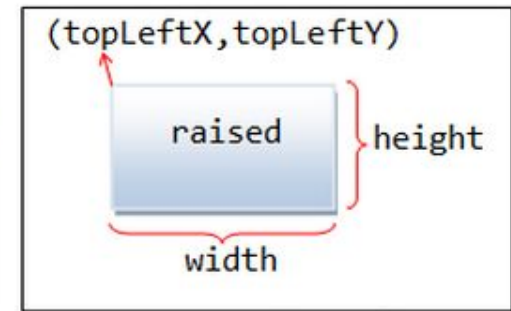
**fillRect()**



**drawPolyline()**



**drawPolygon()**



**fill3DRect()**

# La clase Graphics: métodos para dibujo de imágenes

- Primero se lee o recupera una imagen con la clase `ImageIO`

La manera más fácil de cargar una imagen en el programa es usar el método de clase `read()` de la clase `javax.imageio.ImageIO` el cual retorna un objeto `java.awt.image.BufferedImage`.

Este método está sobrecargado:

```
public static BufferedImage read(URL imagePath) throws IOException
public static BufferedImage read(File imagePath) throws IOException
public static BufferedImage read(InputStream stream) throws IOException
public static BufferedImage read(ImageInputStream stream) throws IOException
```

- Luego se dibuja con `drawImage()`

La clase `Graphics` tiene varios métodos sobrecargados para dibujar una imagen.

Algunos de estos métodos son:

```
public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y, int width, int height,
                                   ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y, Color bgcolor,
                                   ImageObserver observer)
```

# La clase Graphics: métodos para configurar Font y Color

El contexto gráfico mantiene estado (o atributos) como el color actual para pintado, el font para dibujar texto, y el área actual de pintado (llamada *clip*). Para estas funcionalidades se pueden usar los métodos: `setColor()`, `getColor()`, `getFont()`, `setFont()`, `getClipBounds()`, `setClip()`. Cualquier pintado fuera del área del clip es ignorado.

A continuación se muestran algunos de estos métodos de la clase **Graphics**:

```
// Color actual del contexto Graphics
```

```
void setColor(Color c)
```

```
Color getColor()
```

```
// Font actual del contexto Graphics
```

```
void setFont(Font f)
```

```
Font getFont()
```

```
// Set/Get del área clip actual. El área Clip es rectangular y nada será mostrado fuera de ese área.
```

```
void setClip(int xTopLeft, int yTopLeft, int width, int height)
```

```
void setClip(Shape rect)
```

```
public abstract void clipRect(int x, int y, int width, int height)
```

```
Rectangle getClipBounds() // retorna un Rectangle
```

# AWT

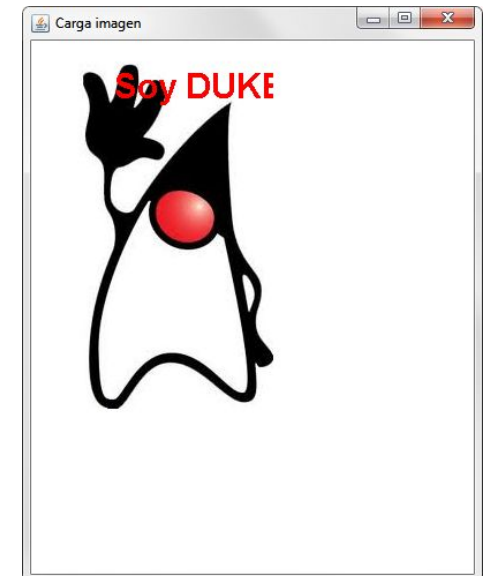
## Carga de una imagen

```
package interfaces.graficas;  
import java.awt.*;  
import java.net.URL;  
import javax.imageio.ImageIO;
```

Relativo a la raíz del proyecto (carpeta /src)

```
public class LoadImageDemo extends Frame {  
    private String imgFileName = "imagenes/duke.jpg";  
    private Image img; // es un objeto BufferedImage
```

```
public LoadImageDemo() {  
    URL imgUrl = getClass().getClassLoader().getResource(imgFileName);  
    if (imgUrl == null) {  
        System.err.println("No se encuentra el archivo:"+imgFileName);  
    } else {  
        try {  
            img = ImageIO.read(imgUrl); // carga imagen en img  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
    setTitle("Carga imagen");  
    setSize(400, 500);  
    setVisible(true);  
    dibujar();  
}  
public void dibujar() { ... }  
    . . .  
}
```



# AWT

## Dibujado en una imagen

```
package interfaces.graficas;
```

```
public class LoadImageDemo extends Frame {  
    private String imgFileName = "imagenes/duke.jpg";  
    private Image img; // es un objeto BufferedImage
```

```
    public LoadImageDemo() {  
        . . .  
        setTitle("Carga imagen");  
        setSize(400, 500);  
        setVisible(true);  
        dibujar();  
    }
```

Se invoca explícitamente al  
dibujar() para que se pinte Duke

```
    public void dibujar() {  
        Graphics gr = img.getGraphics();  
        gr.setColor(Color.RED);  
        gr.setFont(new Font(Font.DIALOG, Font.BOLD, 30));  
        gr.drawString("Soy DUKE", 30, 30);  
        this.getGraphics().drawImage(img, 50, 50, null);  
    }
```

Trabajamos con el objeto  
Graphics de la imagen

Trabajamos con el objeto  
Graphics de la ventana

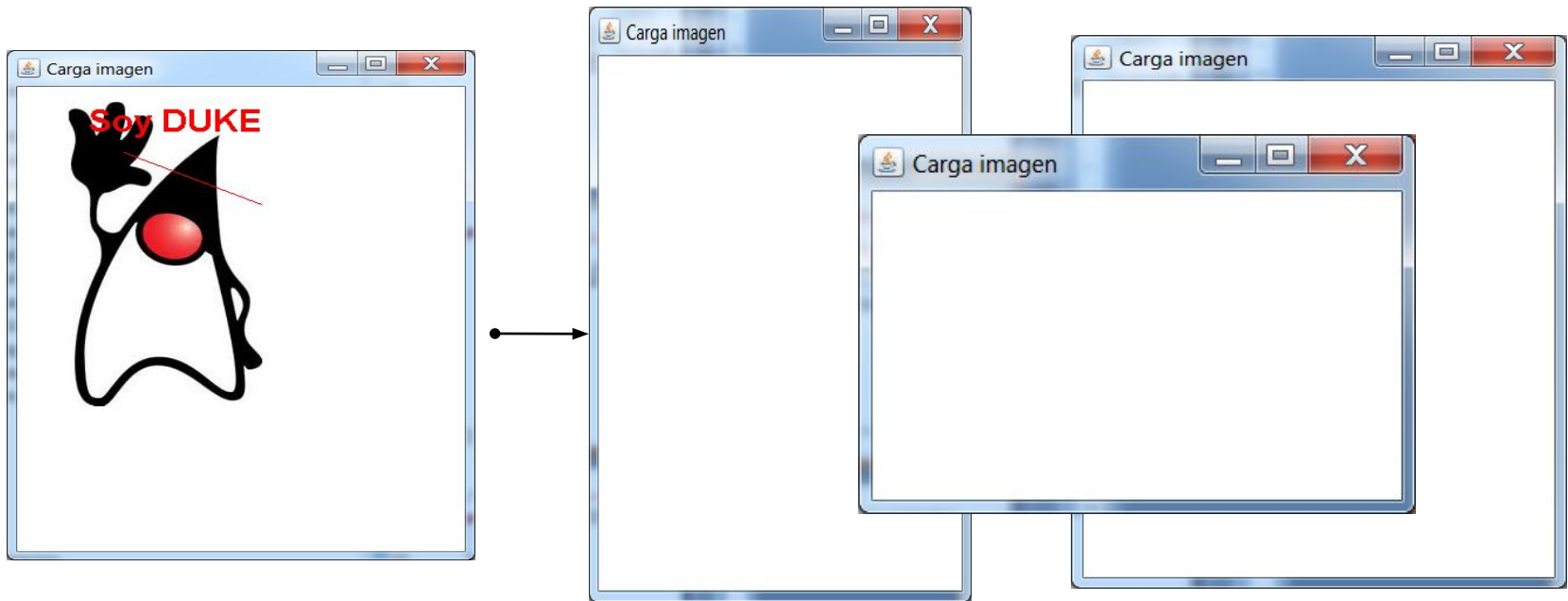
```
    public static void main(String[] args) {  
        LoadImageDemos app = new LoadImageDemo();  
    }  
}
```



# AWT

## ¿Qué pasa cuándo cambiamos el tamaño?

Cuando cambiamos el tamaño de la ventana, el sistema AWT invoca automáticamente al método `repaint()`, el cual primero invoca al `update()` para limpiar la componente y luego al `paint()` para que se repinte. Como nuestra clase no implementa el `paint()`, nuestro DUKE no se pinta mas!



El método `paint()` debe tener las instrucciones para pintar la componente.

El `paint()` no debe invocarse explícitamente desde el programa.

El método `repaint()` controla el pintado invocando la secuencia: `update()` □ `paint()`.

Invocar al `repaint()` para pedir que la componente sea repintada/redibujada.

# AWT

## ¿Qué pasa cuando cambiamos el tamaño?

Cuando la ventana cambia de tamaño, **el sistema AWT invoca automáticamente al método `repaint()`**, el cual primero invoca al `update()` para limpiar la componente y luego al `paint()` para que se repinte. Ahora si la imagen se verá siempre

```
package interfaces.graficas;
public class LoadImageDemo extends Frame {
    private String imgFileName = "imagenes/duke.jpg";
    private Image img;    // es un objeto BufferedImage

    public LoadImageDemo() { . . . }

    public void dibujar() {
        Graphics gr = img.getGraphics();
        gr.drawLine(150, 150, 280, 150);
        gr.setColor(Color.RED);
        gr.setFont(new Font(Font.DIALOG, Font.BOLD, 30));
        gr.drawString("Soy DUKE", 30, 30);
        gr.fillRect(480, 290, 20, 20);
        this.getGraphics().drawImage(img, 50, 50, null);
    }
    @Override
    public void paint(Graphics g) {
        super.paintComponents(g);
        dibujar();
    }
    public static void main(String[] args) {
        LoadImageDemos app = new LoadImageDemo();
    }
}
```

Como nuestro **Frame** tiene un pintado especial, se debe sobrescribir el método **paint()** para que vuelva a pintarse.



# AWT

## paint() y repaint()

El método `repaint()` controla el pintado invocando la secuencia: `update()` □ `paint()`.

Invocar al `repaint()` para pedir que la componente sea repintada/redibujada.

Como ya mencionamos, no es recomendable invocar desde un programa directamente al método `paint()`. En su lugar debe invocarse a alguna de las versiones sobrecargadas del método `repaint()` de la clase `java.awt.Component`:

El método `repaint()` sin parámetros causa que TODA la componente se repinte.

```
public void repaint()
```

Los métodos `repaint(...)` con argumentos permiten definir la región que necesita repararse.

```
public void repaint()  
public void repaint(long tm)  
public void repaint(int x, int y, int width, int heigth)  
public void repaint(long tm, int x, int y, int width, int heigth)
```

**TIP: En Swing el sistema de pintado funciona de manera similar. Si se usan componentes SWING en vez de sobrescribir el `paint()` se debe sobrescribir el método `paintComponent()`.**

# Referencias

## **Documentación de la API de Java:**

`http://docs.oracle.com/javase/7/docs/api/java/awt/Component.html`

## **Sitio oficial de ORACLE:**

`http://www.oracle.com/technetwork/java/painting-140037.html`

`http://docs.oracle.com/javase/tutorial/uiswing/painting/closer.html`

`http://docs.oracle.com/javase/7/docs/api/java/awt/Component.html`