**Academia Java – JAVA**

***Una Historia muy Gráfica***

Tabla de contenido

[UNA HISTORIA MUY GRÁFICA 3](#_Toc314488193)

[1. Todo empieza por una ventana 4](#_Toc314488194)

[2. Pon widgets en la ventana 4](#_Toc314488195)

[2.1. Crear un GUI es fácil! 4](#_Toc314488196)

[3. Tu primera GUI: un botón en un frame 5](#_Toc314488197)

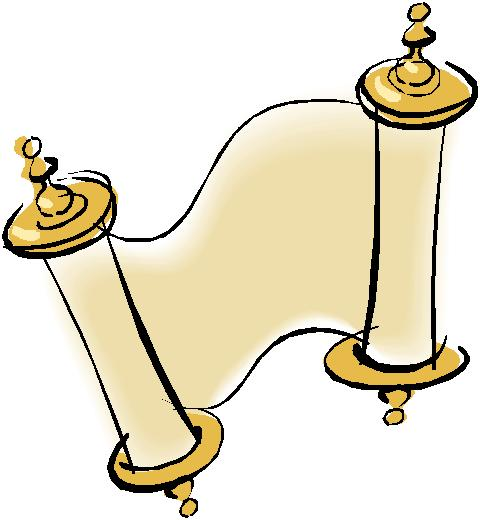
[4. Listeners, Sources y Events 16](#_Toc314488198)

[5. Regresando a los gráficos 18](#_Toc314488199)

[6. Crea tu propio widget 20](#_Toc314488200)

[7. Cosas divertidas que puedes hacer con el método paintComponent() 21](#_Toc314488201)

# UNA HISTORIA MUY GRÁFICA

Afróntalo, necesitas hacer GUIS. Si estas construyendo aplicaciones que otras personas usaran, necesitas una interfaz gráfica.

Aún si crees que el resto de tu vida lo pasaras escribiendo código del lado del servidor, donde la interfaz es una página web, tarde o temprano necesitaras escribir herramientas, y quedrás una interfaz gráfica.

Claro, las aplicaciones basadas en lineas de comando son retro, pero no son buenas. Son débiles, inflexibles y poco amigables.

Usaremos dos capítulos estudiando las GUIS, y utilizar herramientas claves en Java incluyendo los “Event Handler” y las “Inner Classes”.

En este capítulo, ubicaremos un botón en la pantalla, el cual ejecutara algo cuando haces click en él. Pintaremos la pantalla, mostraremos una imagen JPEG, e incluso realizaremos una animación.

# Todo empieza por una ventana

|  |  |
| --- | --- |
| Un JFrame es el objeto que representa una ventana en la pantalla. Es donde ubicas los elementos de la interfaz como buttons, checkboxes, textfields, y más. Puede tener un clásico menu bar con menu items. Además tiene esos pequeños iconos de ventana, cualquiera sea la plataforma en la que te encuentres…si esos que te permiten minimizar, maximizar y cerrar la ventana. |  |

El JFrame cambia su apariencia dependiendo de la plataforma. Asi se observa en Mac OS X:

|  |  |
| --- | --- |
|  | Un JFrame con un menu bar y dos “widgets” (un button y un radio button). |

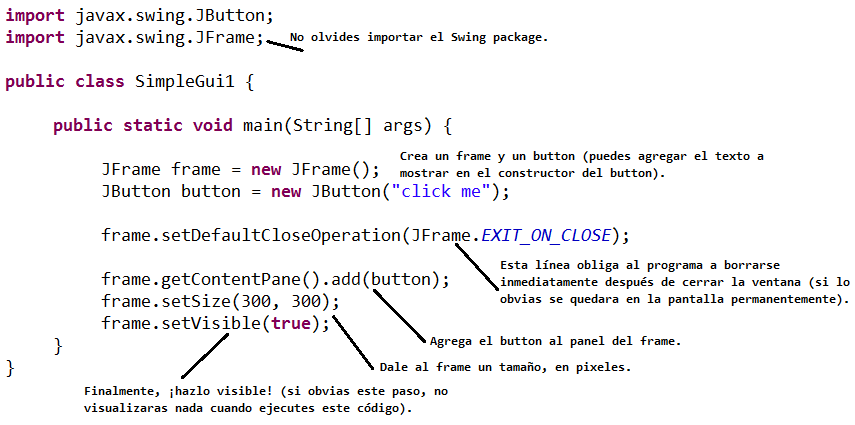
# Pon widgets en la ventana

Una vez que tengas un JFrame, puedes ubicar componentes (“widgets”) agregándolos al JFrame. Hay bastantes componentes que puedes agregar; revísalos en package javax.swing. Los más comunes son JButton, JRadioButton, JCheckbox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField y JTable. La mayoría son muy fáciles de usar, pero algunos (como el JTable) pueden ser un poco más complicados.

# Crear un GUI es fácil!

|  |  |
| --- | --- |
|  | Crear un JFrame. |
|  | JFrame frame = new JFrame(); |
|  | Crear un widget (button, textfield, etc). |
|  | JButton button = new JButton(“click me!”); |
|  | Agregar el widget al JFrame. |
|  | |  | | --- | | frame.getContentPane().add(button); | | Nunca se agrega elementos al frame de manera directa. Piensa en el frame como un corte alrededor de la ventana, al cual le agregas un panel y es al panel al que le agregas los elementos. | |
|  | Muéstralo (dándole un tamaño y haciéndolo visible). |
|  | frame.setSize(300, 300);  frame.setVisible(true); |

# Tu primera GUI: un botón en un frame



**Veamos qué pasa cuando lo ejecutamos:**

|  |  |
| --- | --- |
|  | ¡Vaya! Ese si es un botón muy grande.  El botón ocupa todo el espacio disponible. Más tarde aprenderemos como controlar el tamaño y la ubicación del botón en el frame. |

**Pero ¿qué es esto? ¡Nada sucede cuando hago click!**

No es del todo cierto. Cuando presionas el button se ve “presionado” o “apretado” (la forma en la que se observa varía dependiendo de la plataforma en la que te encuentres, pero siempre se observan cambios al ser presionado).

La verdadera pregunta es...“¿Cómo consigo que el button ejecute algo especifico cuando el usuario hace click?”

**Necesitamos dos cosas:**

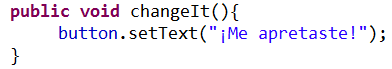
|  |  |
| --- | --- |
|  | Un **método** que sea invocado cuando el usuario hace click. (Lo que deseas que se haga como resultado del click). |
|  | Una forma de saber **cuándo** invocar ese método. En otras palabras, una manera de saber cuando el usuario hace click en el button. |

|  |  |
| --- | --- |
| Queremos saber cuando el usuario hace click.  Estamos interesados en el evento “usuario presiona el button”. |  |

|  |  |
| --- | --- |
| P: ¿Un button tendrá la apariencia de un button de windows cuando lo vea en windows?  R: Si lo eliges, sí. Puedes elegir de unas pocas clases “look and feels”, que te permitirán controlar la apariencia de la interfaz. En la mayoría de los casos, puedes elegir entre al menos dos “look and feels” diferentes: el standard de Java, también conocida como ***Metal***, y la forma nativa de la plataforma en la que estés. Algunas de las pantallas que se muestran en Mac, usan el “look and feel” OS X ***Aqua***; o el “look and feel” ***Metal***.  P: ¿Puedo lograr que la apariencia se vea como Aqua todo el tiempo? ¿Incluso en Windows?  R: No. No se encuentran disponibles todos los “look and feels” en todas las plataformas. Para asegurarte de la apariencia, especifica el “look and feel” Metal para saber cómo se verá la aplicación sin importar la plataforma en la que te encuentres; si no lo especificas, se verá usara el “look and feels” por defecto.  P: He oído que Swing es lento y nadie lo usa.  R: Esto fue cierto en el pasado, pero ya no más. En máquinas de bajo rendimiento, se sentiría cierto retraso al usar swing. Pero en maquinas más recientes, y usando Java versión 1.3 para adelante, no sentirás la diferencia entre usar swing GUI y la que viene por defecto en la plataforma. Swing es muy usado hoy, en todo tipo de aplicaciones. | C:\Users\ingas\Desktop\Atencion2.png |

**Capturando un user event**

Imagina que deseas cambiar el texto en el button de “¡Hazme click!” a “¡Me apretaste!”, cuando el usuario presione el button. Primero escribiremos un método que cambie el texto del button (una revisión rápida a través de la API te enseñará el método).

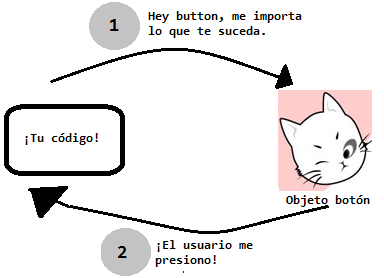


Pero... ¿Y ahora qué? ¿Cómo sabemos cuando este método se debe ejecutar? ¿Cómo sabemos cuando el button fue presionado?

En Java, el proceso de capturar y manejar un event del usuario (user event) es llamado “event handling”. Existen diferentes tipos de eventos en Java, aunque la mayoría involucra a los que se ejecutan cuando el usuario interactúa con la GUI. Si el usuario hace click en el button, ese es un evento. Un evento que dice “el usuario quiere que la acción de este button se ejecute”. Si es un “desaparecer” button, el usuario quiere que la acción de desaparecer ocurra. Si es un botón de “enviar” en un cliente de mensajería, el usuario quiere que la acción “enviar mi mensaje” se ejecute. Es así que el evento más directo es cuando el usuario hace click, indicando que una acción se ejecute.

Con el button, usualmente no te preocupas por eventos de tipo intermedio como “esta siendo presionado”, o “está siendo liberado”. Con esto le dices al button: “No me importa como el usuario interactúe, cuánto tiempo presione el mouse, etc. ¡**Sólo dime cuando el usuario quiere algo!** En otras palabras, no me llames a menos que usuario presione el botón indicando que desea que se ejecute la acción de ese botón”.

**Primero, el button necesita saber que nos importa**

****

**Segundo, el button necesita una manera de indicarnos cuando un evento ocurrió**

|  |  |
| --- | --- |
|  | 1. ¿Cómo puedes decirle al button que te importan sus eventos? ¿Que estás pendiente de sus eventos? 2. ¿Cómo el button te responderá? Asume que no existe forma de decirle al button el nombre de único método. Entonces, ¿qué otra forma puedes usar para asegurarte que el button sepa que tenemos un método específico a ejecutar cuando el evento sucede? |

Si te importa los eventos del button, implementa una interface que diga “**Estoy atento a tus eventos”.**

Un “listener interface” es el puente entre el listener (tu) y la raíz del evento (el button).

Los componentes GUI de Swing son la raíz de los eventos. En términos de Java, la raíz de evento es un objeto que puede convertir acciones del usuario (presionar un button o una tecla, cerrar una ventana, etc.) en eventos. Y como casi todo en Java, un evento es representado como un objeto. Un objeto de alguna clase de eventos. Si realizas una búsqueda en el package java.awt.event dentro del API, observaras varios clases de eventos (fáciles de distinguir, todas tienen “event” en el nombre). Encontrarás MouseEvent, WindowEvent, ActionEvent y otros tantos.

Una raíz de evento (como un button) crea un event object (objeto de evento) cuando el usuario realiza algo relevante (como el click de un button). La mayoría de código que escribas (y todo el código en este libro) más que crear eventos, va a recibir eventos. En otras palabras, usaras más tú tiempo siendo el que capte los eventos, que como la fuente de esos eventos.

Cada tipo de evento tiene un correspondiente “listener interface”. Si quieres eventos relacionados al mouse, implementa un “MouseListener interface”. ¿Quieres eventos relacionado a la ventana? Implementa un “WindowsListener”. Y así. Recuerda las reglas de las interface, declaras lo que implementas (la clase Perro implementa la interface Mascota) lo cual significa que debes escribir métodos de implementación para cada método en la interface.

Algunas interfaces tienen más de un método por que el evento viene con variaciones. Si implementas MouseListener, por ejemplo, puedes obtener variaciones como “mousePressed”, “mouseReleased”, “mouseMoved”, etc. Cada uno de esos eventos de mouse tiene un método separado en la interface, aunque todos tengan un “MouseEvent”. Si implementas “MouseListener”, el método mousePressed() es invocado cuando el usuario presiona el mouse. Y cuando lo suelta, el método mouseReleased() es invocado. Entonces para los eventos del mouse, existe sólo un “object event”: MouseEvent; a la vez que diferentes métodos de eventos, que representan los diferentes *tipos* de eventos de mouse.

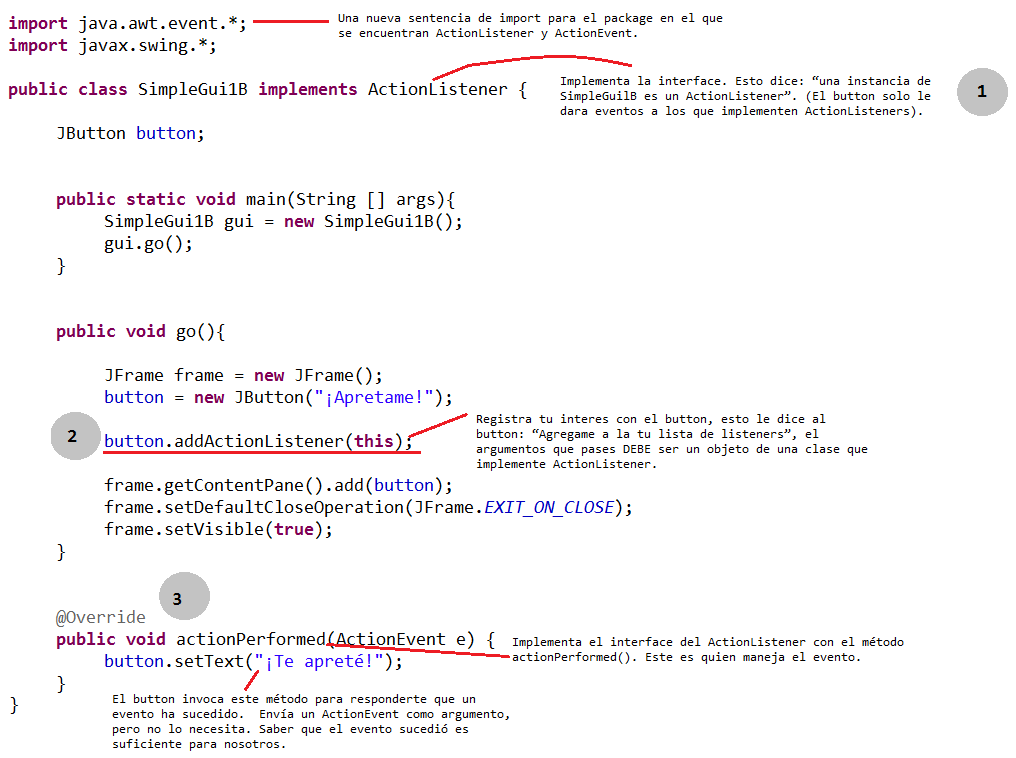
|  |  |
| --- | --- |
| Cuando implementas un listener interface, le das al button una manera de responderte. La interface es donde el método de retorno es declarado. |  |

**Como el listener y la raíz se comunican**

|  |  |
| --- | --- |
|  | |
| **El Listener**  Si tu clase quiere saber acerca de los ActionEvents de un button, implementarás la interface ActionListener. El botón necesita saber que estas interesado, así que registras el button al invocar su método addActionListener(this), a la vez que le pasas una referencia (en este caso, la misma clase es el ActionListener, por lo que le pasas “this”). El button necesita una manera de responderte cuando el evento ocurra, por lo que invoca al método en el listener interface. Ya que eres un ActionListener, tú debes implementar el único método de esta interface, actionPerformed(). El compilador lo verificará. | **El Event Source**  Un button es la raíz de ActionEvents, por lo tanto debe existir una forma de decirle cuales son los objetos que están como Listeners. El button tiene un método addActionListener() para darle a los objetos interesados (listeners) una manera de decirle al button que están interesados. Cuando el método addActionListener() del button se ejecuta (porque un listener lo invocó), el button toma el parámetro (la referencia al objeto listener) y lo almacena en una lista. En el momento que el usuario hace click en el button, éste último dispara el evento al invocar el método actionPerformed() de cada listener en la lista. |

**Creando el ActionEvent de un button**

|  |  |
| --- | --- |
|  | Implementar la interface ActionListener. |
|  | Registrarla con el button (decir si quieres ser un listener de los eventos). |
|  | Definir el método que maneje el evento (implementar el método actionPerformed() de la interface ActionListener). |

**

# Listeners, Sources y Events

Durante la mayor parte de tu carrera en Java, no serás la raíz (source) de los eventos (no importa que tanto creas que eres el centro del universo).

Acéptalo, tu trabajo es un ser un buen listener (lo cual, si haces sinceramente, puede mejorar tu vida social).

|  |  |
| --- | --- |
|  |  |
|  | |

|  |  |
| --- | --- |
| P: ¿Por qué no puedo ser una fuente de eventos?  R: Puedes. Solo hemos dicho que la mayor parte del tiempo serás el receptor, y no el originador del evento (al menos en los primeros días de tu brillante carrera en Java). La mayoría de los eventos que te podrían importar son disparador por algunas clases en la API de Java, y todo lo que tienes que hacer es ser un listener para estas. Podrías sin embargo, diseñar un programa donde necesites un evento personalizado, digamos “StockTiendaEvent” que se disparará cuando la aplicación que monitorea el mercado encuentre algo que sea importante. En este caso, se crearía un objeto StockWatcher, el cual sería el event source; y realizarías las mismas cosas que un button: crear una interface listener para tu evento personalizado, crear un evento de registro que permita a los objetos ser listener (addStockListener), y al invocarlos ponerlos en su lista de listeners. Entonces cuando un evento en el stock suceda, instancia un objeto StockEvent y lo envia a la lista de tus listeners invocando su método stockCambio(StockEvent ev). Y no olvidar que para cada tipo de evento, debe crearse su correspondiente listener en la interface (por tanto, vas a crear una interface StockListener con un método stockChanged()).  P: No veo la importancia de object event que es pasado a los métodos de los eventos. Si alguien invoca mi método mousePressed... ¿Qué otra información necesitaría?  R: Muchas veces, para la mayoría de diseños, no necesitas el object event. Sólo es un pequeño contenedor de información, acerca de la naturaleza del evento. Sin embargo algunas veces podrías necesitar buscar en la información del evento por detalles específicos. Por ejemplo, si tu método mousePressed() es invocado, sabes que el mouse fue presionado. Pero... si quieres saber donde fue presionado, ¿Cómo lo reconocerías? En otras palabras, si quieres saber si fue presionado en las coordinada X,Y de la pantalla correctas. O algunas veces, podrías querer registrar el mismo listener con múltiples objetos. Una calculadora, por ejemplo, tiene diez teclas numéricas y que todas hacen lo mismo; no crearías un listener para cada tecla. En su lugar, podrías registrar un único listener para cada una de las diez teclas y cuando se registre un evento (porque tu método de retorno es invocado), puedes llamara a un método en el evento para descifrar cual fue la verdadera fuente del evento. En otras palabras, saber que tecla envió este evento. | C:\Users\ingas\Desktop\Atencion2.png |

|  |  |
| --- | --- |
| * ¿Como saber si un objeto es fuente de eventos? * Observa en la API. * Bien, ¿Qué busco? * Un método que empiece con “add” acabe con “Listener”, y tome un argumento de tipo interface. Si observas:      * Sabes que una clase con este método es una fuente de eventos “KeyEvents”. Hay un patrón establecido. | C:\Users\ingas\Desktop\Atencion2.png |

# Regresando a los gráficos

Ahora que conocemos un poco de como los eventos funcionan (aprenderemos más después), regresemos a ubicar cosas en la pantalla. Usaremos unos minutos viendo maneras entretenidas de estudiar los gráficos, antes que regresemos al manejo de eventos.

**Tres maneras de ubicar elementos en tu GUI**

|  |  |  |
| --- | --- | --- |
|  | **Poner widgets en un frame**  Agrega buttons, menús, radio buttons, etc.  El package javax.swing tiene más de una docena de tipos de widgets. |  |
|  | **Dibujar gráficos en 2d en un widget**  Usa un objeto gráfico para pintar formas.  Puedes pintar mucho más que cajas y círculos.  El API 2D de java está lleno de entretenidos y sofisticados métodos para los gráficos. |  |
|  | **Ubicar un JPEG en un widget**  Puedes ubicar imágenes en un widget. |  |

# Crea tu propio widget

Si quieres poner tus propios gráficos en la pantalla, tu mejor opción es hacer tu widget personalizado. Lo insertas en el frame, tal como lo haces con un button u otro widget, pero cuando se muestre, mostrará tus imágenes.

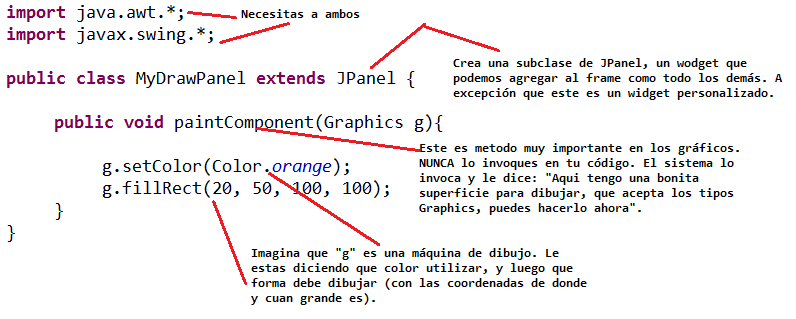
Puedes incluso hacer que esas imágenes se muevan, como en una animación; o cambiar los colores de la pantalla cada vez que se haga click en un button.

¡Es muy fácil!

**Crea una subclase de JPanel y sobrecarga el método paintComponent()**

Todo el código de los gráficos va dentro del método paintComponent(). Piensa en este método como el que es llamado por el sistema para decir: “Hey widget, es hora de pintarte por ti mismo”. Si quieres dibujar un círculo, el método paintComponent() tendrá el código para dibujar un círculo. Cuando el frame que tiene el panel que has dibujado es mostrado, el método paintComponent() es invocado y tu circulo aparece. Si el usuario minimiza la ventana, la JVM ya está enterada que el frame necesita ser “restaurado” cuando sea maximizado, por tanto en ese momento llama nuevamente al método paintComponent(). Recordar que en cualquier momento que la JVM crea que es necesario refrescar la pantalla, el método paintComponent() será llamado.

Algo más, **jamás llames a este método**. El argumento de este método (un object de tipo Graphics) es la imagen misma que estás viendo en la pantalla. Esto no lo puedes usar; puesto que te es entregado por el sistema. Más tarde, sin embargo, puedes pedirle al sistema que refresque la pantalla (método repaint(), quien al final invoca al método paintComponent()).

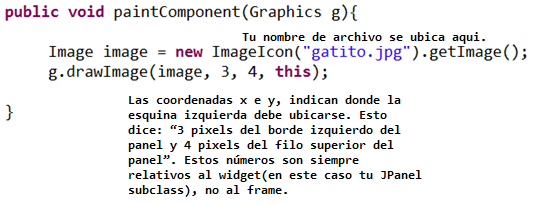


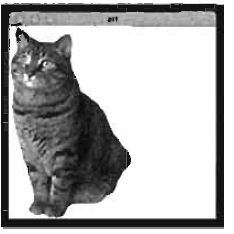
# Cosas divertidas que puedes hacer con el método paintComponent()

Demos una vista a unas pocas cosas extras que puedes hacer en paintComponent(). La más divertida es cuando empiezas a experimentar por ti mismo.

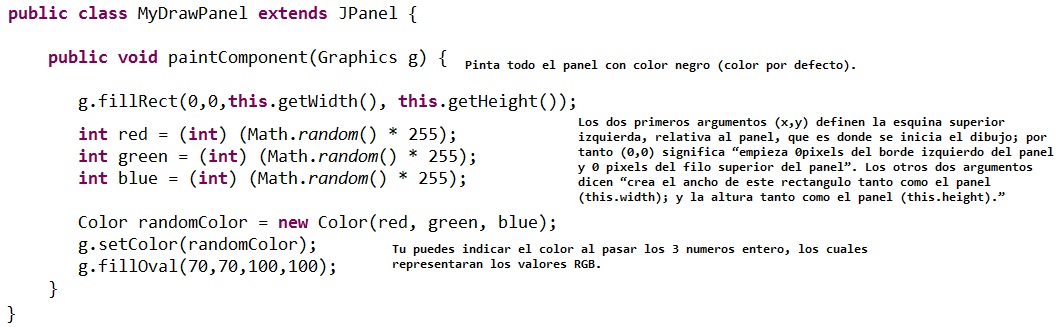
Trata de jugar con los números, revisa la API por clases de tipo Graphics (más tarde veremos que puedes intentar nuevas cosas mas allá de lo que ofrece la clase Graphics).

**Mostrando un JPEG**

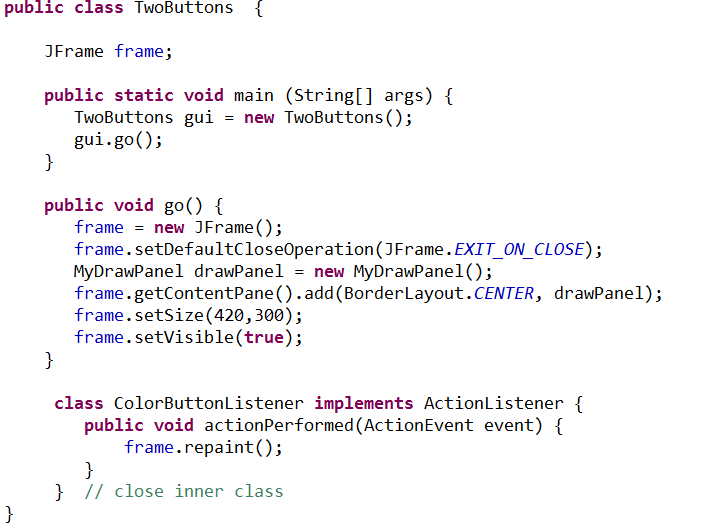
****

****

**Dibujando un círculo de colores aleatorios en un fondo negro**



|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |



**Detrás de cada buen gráfico referente hay un objeto gráfico en 2D**

El argumento al método paintComponent() es declarado de tipo Graphics (java.awt.Graphics).

**public** **void** paintComponent(**Graphics** g)

Por tanto el parámetro ‘g’ es un objeto Graphics. Lo cual quiere decir que podría ser una subclase de Graphics (a causa del polimorfismo). Y de hecho lo es.

**El objeto referenciado por el parámetro ‘g’ es una instancia de la clase Graphics2D**

¿Por qué importa? Porque hay varias cosas que puedes hacer con la referencia a esa clase, que no puedes hacer con la de Graphics. Un objeto de clase Graphics2D es capaz de crear más que un objeto tipo Graphics, y es realmente el responsable detrás de la referencia a Graphics.

Recuerda tu polimorfismo. El compilador decide cuales métodos puedes invocar basado en el tipo de referencia, no el tipo de objeto. Si tienes un objeto Perro referencia por una variable Animal referenciada:

Animal a = **new** Dog();

NO puedes decir:

a.bark();

Aun cuando sabes que realmente es un Perro. El compilador mira a ‘a’, mira que es del tipo Animal, y encuentra que no existe una forma de invocar al metodo bark() en la clase Animal. Sin embargo puedes recuperar el objeto al tipo Dog si lo escribes así:

Dog d = (Dog) a;

d.bark();

Así que la lección con el objeto Graphics es esta:

***Si necesitas usar un método de la clase Graphics2D, no puedes usar el parámetro ‘g’ directamente desde el método, pero lo puedes extraer con una nueva variable de tipo Graphics2d.***

|  |  |
| --- | --- |
| Métodos que puedes invocar en una referencia a Graphics.  drawImage()  drawLine()  drawPolygon()  drawRect()  drawOval()  fillRect()  fillRoundRect()  setColor()  Para extraer el objeto tipo Graphics2d desde la referencia.  Graphics2D g2d = (Graphics2D) g;  Métodos que puedes invocar en una referencia a Graphics2d.  fill3DRect()  draw3DRect()  rotate()  scale()  sheer()  transform()  selRenderingHints()  Esta no es una lista completa, revisa la API para más métodos. | C:\Users\ingas\Desktop\Atencion2.png |

|  |  |
| --- | --- |
| **Eventos:**   * Para crear una GUI, empieza con una ventana, usualmente un Jframe: JFrame frame = **new** JFrame(); * Puedes agregar widgets(buttons, textfields, ect) al Jframe usando: frame.getContentPane(). add(BorderLayout.*CENTER*, drawPanel); * A diferencia de la mayoría de otros componentes, el JFrame no te permite agregar directamente componentes, debes agregar un content pane primero. * Para que la ventana JFrame se muestre, debes darle un tamaño y decirle que sea visible: frame.setSize(300,300); frame.setVisible(true); * Para saber cuando el usuario hace click en un button (o realiza alguna acción en la interfaz) necesitas captar el evento de GUI. * Para captar un evento, debes registrar tu interés en un event source(fuente de eventos). Un event source es el artefacto que dispara un evento basado en la interacción del usuario. * La interface de los listener le da al event source una forma de responder, porque la interface define los métodos que se invocaran cuando el evento suceda. * Para registrar eventos dentro de un source, invoca método de registro. Los métodos de registro siempre toman la forma de “add<EventType>Listener”. Para registrarse en los eventos del button: button.addActionListener(this). * Implemente el interface al implementar todos los metodos event handling de esa interface. Ubica todo el código en el método de respuesta. Para los ActionEvents, el método es:   **public** **void** actionPerformed(ActionEvent e) {  button.setText("¡Te apreté!");  }   * El event Object (objeto de evento) pasado en el método carga la información acerca del evento, incluyendo la fuente del evento. | C:\Users\ingas\Desktop\Atencion2.png |

|  |  |
| --- | --- |
| **Graphics**   * Puedes pintar gráficos 2d directamente a un widget. * Puedes insertar .jpeg .gif al widget. * Para pintar tus propios gráficos (incluyendo un .gif o .jpeg), crea una subclase del JPanel y sobre-escribe el método paintComponent(). * El método paintComponent() es llamado por el sistema GUI. Nunca lo invoques. El argumento del método es un objeto de tipo Graphics que te permite pintar en la pantalla. NO puedes construir ese objeto por ti mismo. * Métodos clásicos para invocar el objeto Graphics son:   graphics.setColor(Color.blue);  g.fillRect(20,SO,lOO,120);   * Para pintar un .jpg construye una imagen usando:   Image image = new ImageIcon("catzilla.jpg").getImage();   * Y para dibujarla:   g.drawImage(image,3,4,this);   * El objeto referenciado por el parámetro Graphics en el método paintComponent() es realmente una instancia de la clase Graphics2D. La clase Graphics2d tiene una variedad de métodos incluyendo: fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform(). * Para invocar los métodos de la clase Graphics2D, debes extraer del parámetro el objeto Graphics2D: Graphics2D g2d = (Graphics2D) g; | C:\Users\ingas\Desktop\Atencion2.png |

**Capturamos eventos.**

**Pintamos gráficos.**

**¿Pero podemos pintar gráficos *cuando* capturamos un evento?**

Vamos a unir un evento con un cambio en nuestro panel de dibujo. Lograremos que el círculo cambie de colores cada vez que haces click en el button. Aquí mostramos el flujo.

|  |  |
| --- | --- |
|  | El frame es construido con dos widgets (tu panel y tu button). Un listener es creado y registrado con el button. Después el frame es mostrado y espera que el usuario haga click. |
|  |  |
|  | El usuario da click al button y el button crea un event object, el cual invoca el event handler del listener. |
|  |  |
|  | El event handler invoca al método repaint() en el frame. El sistema invoca paintComponent() en el panel. |
|  | Listo! Un nuevo color es pintado por el método paintComponent() se ejecuta de nuevo, llenando el círculo con un color aleatorio |
|  |  |

**Capas de la GUI: ubicando más de un widget en el frame**



Cubriremos las capas (layouts) en el siguiente capítulo, pero daremos una lección rápida para empezar. Por defecto, un frame tiene 5 regiones a las que les puedes agregar cosas. Tu solo puedes agregar un elemento a cada región del frame, pero no entres en pánico. Ese único elemento podría ser un panel que contenga 2 elementos o más.... De hecho, nos “equivocamos” al agregar el button de manera directa usando:

frame.getContentPane().add(colorButton);

Esta no es la forma en la que correctamente se hace (método de un solo argumento).

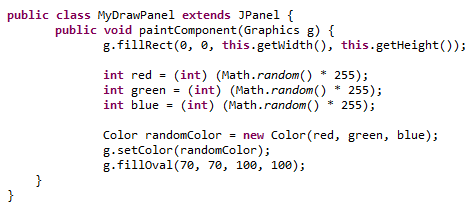
|  |  |
| --- | --- |
|  | Cuando invocas el método que agregar solo un argumento (el que no debemos usar) el widget automáticamente aterrizara en la región central. |

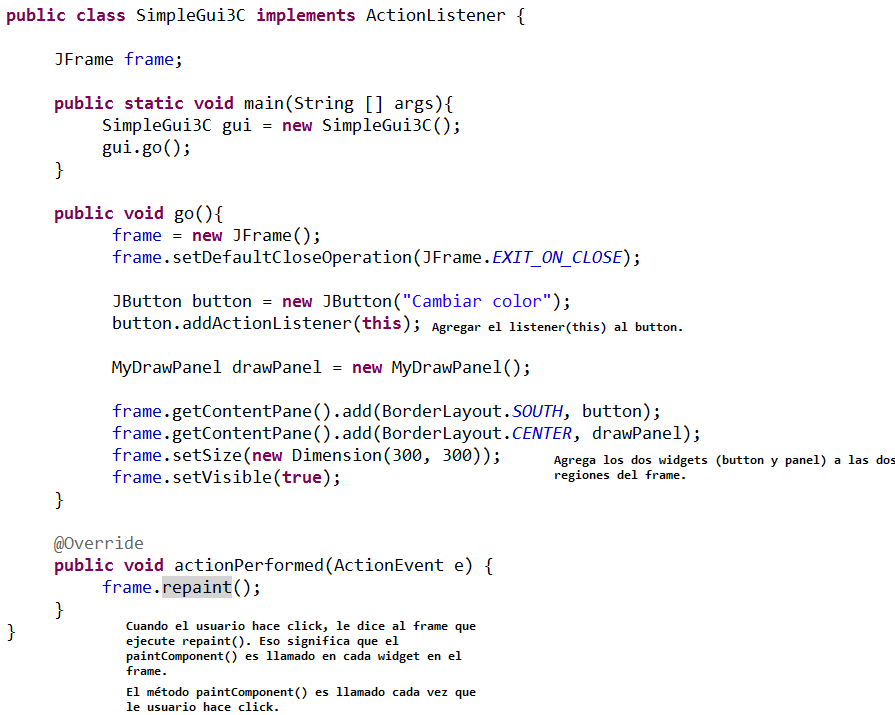
La siguiente es una mejor forma (y usualmente obligatoria) de agregar un content pane por defecto a un frame. Siempre específica donde (la región) quieres que vaya el widget:

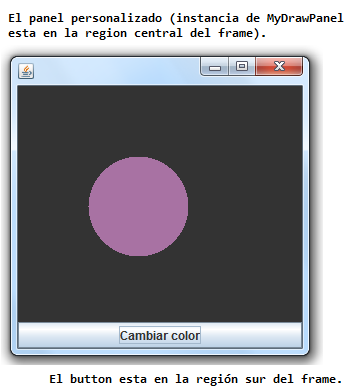
frame.getContentPane().add(BorderLayout.*CENTER*, colorButton);

Invocamos el método de dos argumentos, el cual toma la región (usando una constante) y el widget a agregar.

**El círculo cambia de color cada vez que haces click en el button**



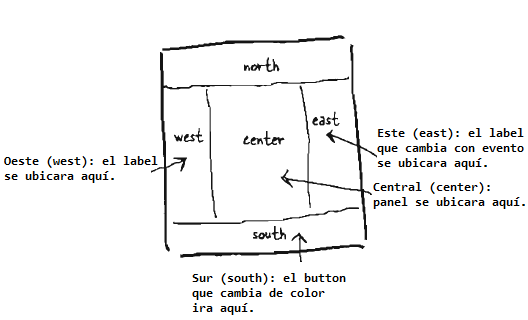




**Intentémoslo con dos buttons**

El button en la región sur actuara como siempre, simplemente repintando en el frame. El segundo button (el cual pondremos en la región este) cambiara el texto en el label (un label es solo texto en la pantalla).

**Por tanto ahora necesitamos cuatro widgets**



**Y ahora necesitamos capturar dos eventos**

¿Es posible? ¿Cómo capturas dos eventos cuando solo tienes un método ActionPerformed()?



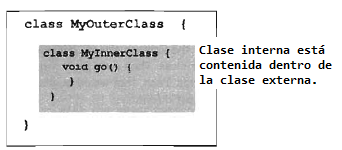
**¿Cómo capturas eventos de dos button cuando cada uno realiza algo diferente?**

|  |  |
| --- | --- |
|  | **Opción Uno**  Implementar dos métodos ActionPerformed |
|  |  |
|  | Falla: ¡No se puede! No puedes implementar el mismo método dos veces en una clase Java. No compilara. E incluso si se permitiera ¿cómo podría saber el event source cual de los dos métodos a llamar? |
|  | **Opción dos**  Registrar el mismo listener con ambos buttons |
|  |  |
|  | Debilidad: Funciona, pero en la mayoría de los casos no es muy buena práctica. Un event handler realizando diferentes cosas significa que tendrás un único método ejecutando diferentes acciones. Si necesitas cambiar como una fuente es manejada, tendrás que modificar todo el resto de los handlers. Algunas veces es buena solución, pero usualmente reduce su mantenimiento y escabilidad |
|  | **Opción tres**  Crear dos clases separadas de ActionListener. |
|  |  |
|  | Falla: Estas clases no tienen acceso a las variables que necesitan modificar, ‘frame’ y ‘label’. Puedes solucionarlo, pero les darías cada clase listener una referencia a la clase principal de la GUI, por tando dentro de los métodos actionPerformed() el listener podría usar la referencia a la clase de la GUI para acceder a sus variables. Pero eso es romper la encapsulacion, es asi que necesitariamos crear métodos get para los GUI widgets (getFrame(), getLabel(), etc). Y probablemente necesitemos agregar un constructor a la clase listener para que pases la referencia de la GUI al listener cuando este es instanciado. Y bueno, se complica y se vuelve confuso.  ***¡Debe existir una mejor forma!*** |



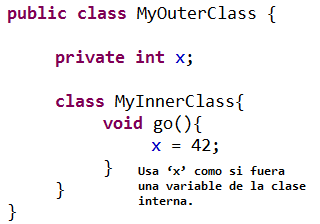
**¡Clases internas al rescate!**

Puedes crear una clase dentro de otra. Es fácil. Solo asegúrate que la definición de la clase interna esta dentro de las llaves del la clase externa.



Una clase interna obtiene un pase especial a los elementos de la clase externa. Incluso sus elementos privados. Y la clase interna puede usar aquellas variables y métodos privados como si fueran definidos dentro de ella. Eso es lo práctico de las clases internas, puesto que tienen todos los beneficios de una clase normal, pero sin derechos especiales de acceso.

**Clase interna usando una variable de la clase externa**



Una clase interna puede usar todos los métodos y variables de la clase externa, incluso los elementos privados.

La clase interna consigue usar aquellas variables y métodos como si hubieran sido declarados dentro de la misma clase interna.

**Una instancia de una clase interna debe estar unida a una instancia de la clase externa.**

Recuerdas cuando hablamos de la clase interna accediendo a elementos de la clase externa, lo que hacemos es que una instancia de la clase interna accediendo a elementos de la clase externa. ¿Pero a cual instancia?

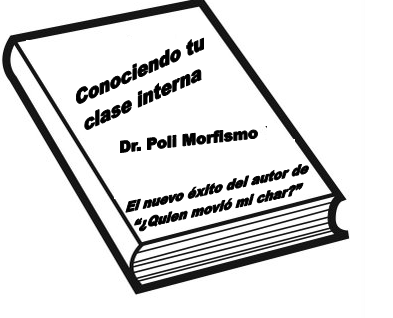
¿Puede arbitrariamente una instancia de la clase interna acceder a los métodos y variables de cualquier instancia de la clase externa? **NO**.

Un objeto interno debe estar unido a un específico objeto exterior.

Un objeto interno comparte una unión especial con un objeto externo.

|  |  |  |
| --- | --- | --- |
|  | Crea una instancia de la clase externa. |  |
|  | Crea una instancia de la clase interna, usando la instancia de la clase externa. |  |
|  | La clase externa e interna ahora están íntimamente unidos. |  |

Estos dos objetos en la carga tienen una unión especial. El interno puedes usar las variables de la clase externa.



Existe una excepción a esto, para un caso especial, una clase interna definida dentro de un método static. Pero no estamos ahí, y es muy probable que nunca lo encuentre en tu vida de desarrollador.

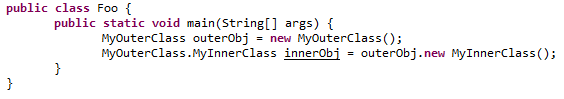
**¿Cómo hacer una instancia de una clase interna?**

Si instancias una clase de interna desde el código de la clase externa, esa instancia será con la que se una la clase interna. Por ejemplo, si el código dentro de un método instancia la clase interna, el objeto interno se unirá a la instancia de aquel método que se está ejecutando.

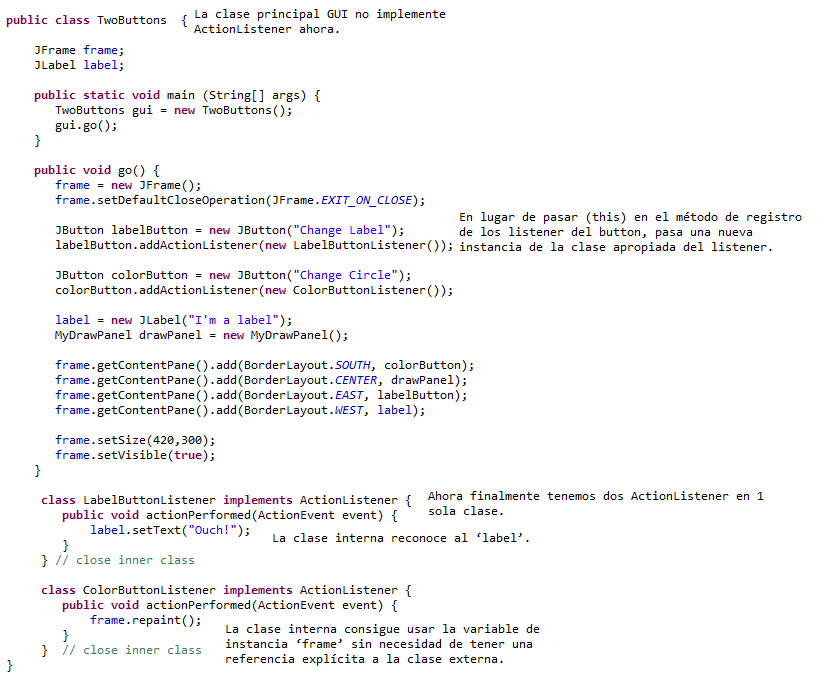
El código en la clase externa puede instanciar una de las clases internas, de la misma forma que instancia cualquier otra clase.

|  |  |
| --- | --- |
|  |  |

Puedes instanciar un elemento instanciado de la clase interna desde la clase externa, pero tienes que usar una sintaxis especial. Es poco probable que lo encuentres en todo tu vida como desarrollador, e incluso usarlo, pero en el caso que estés interesado.



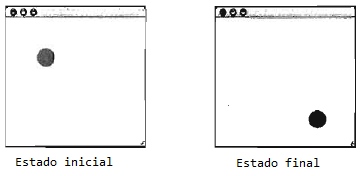
**Finalmente podemos hacer funcionar el código de dos buttons**



**Usando una clase interna para la animación**

Ya vimos por que las clases internas son útiles para los listener event, ya que consigues implementar el mismo método de respuestas más de una vez. Pero ahora veremos que tan útil es una clase interna cuando es una subclase, a la clase externa no aplica ‘extend’. En otras palabras, cuando la clase externa y la clase interna están en diferentes arboles de herencia.

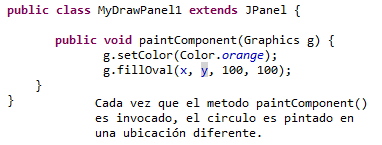
Nuestra meta es crear una simple animación donde el círculo se mueva a través de la pantalla, de la esquina superior izquierda la parte inferior derecha.



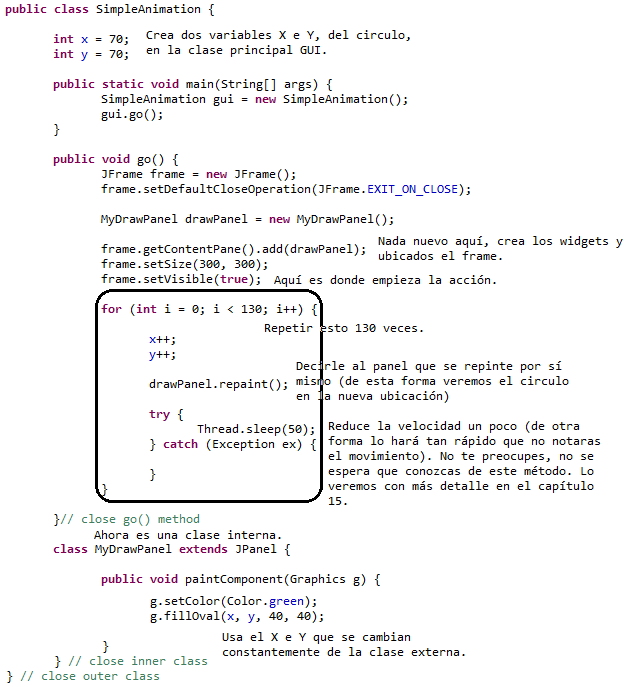
**Como funciona una animación**

|  |  |
| --- | --- |
|  | Pinta un objeto con una coordenada X e Y.  g.fillOval(20, 50, 100, 100);  20 pixels desde la izquierda y 50 pixels desde el alto |
|  | Repinta el objeto en coordenada X diferente y la misma coordenada Y  g.fillOval(25, 55, 100, 100);  25 pixels desde la izquierda, 55 pixels desde el alto (el objeto se moverá un poco hacia abajo y hacia la derecha) |
|  | Repite el paso previo con los valores cambiantes de X e Y, tanto como la animación se supone deba ser. |

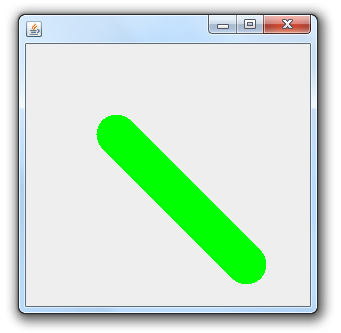
**Lo que realmente queremos es....**

****

**El código resuelto de la animación**



**No se movió... Dejo una mancha...**

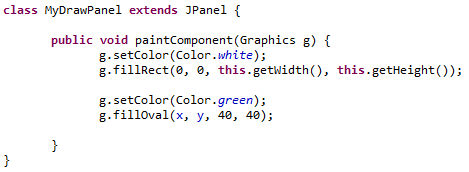


¿Que hicimos mal?

Existe una pequeña falla en el método paintComponent().

**¡Olvidamos borrar lo que estaba! Por tanto dejamos huellas**

Para solucionarlo, todo lo que tenemos que hacer es llenar el panel entero el fondo del color original cada vez, antes de pintar el círculo. Este código agrega dos líneas al inicio del método: uno para poner el color a blanco (el color de fondo del panel), y el otro para llenar el panel del rectángulo con ese color. Dice: “Llena un rectángulo empezando X e Y desde 0 y hazlo tan ancho y alto como el panel actual es”.



**Realicemos un video musical. Usaremos gráficos aleatorios generados por Java que guardaran correspondencia con la música.**

**Junto a ello, registraremos (y escucharemos) una nueva clase de evento no-GUI, disparado por la música en sí.**

Recuerda, esta parte es opcional. Pero pensamos que es bueno para tu formación. Y te gustará, a la vez que puedes usarlo para impresionar a la gente.

(Bueno... seguro solo a la gente que es fácil de impresionar, pero aun así...)

**Capturando un evento no-GUI**

Bien, tal vez no es un video musical, pero haremos que un programa dibuje aleatoriamente gráficos en la pantalla con el tono de la música. En corto, el programa escucha la tonada de la música y dibuja un rectángulo aleatorio con cada tono.

Eso trae nuevos preguntas para nosotros. Hasta ahora, hemos capturado solo eventos GUI, pero ahora necesitamos capturar un evento particular tipo MIDI. Resulta que capturar un evento no-GUI es como capturar uno GUI; implementas un interface listener, registras el listener con un event source; luego esperas que el event source llamar a tu método event handler (el método definido en el listener interface).

La forma más simple de capturar la tonada de la música seria registrar y oír los eventos MIDI que se ejecutan, de esta manera cuando el secuenciador captura el evento, nuestro código lo obtendrá y dibujara el grafico. Pero existe un problema. Un bug, de hecho, que no permitirá que capturemos los eventos MIDI que creemos.

Por tanto tenemos que crear un artificio. Existe otro tipo de evento MIDI que podemos capturar usando el ControllerEvent. Nuestra solución es registrar varios ControllerEvent, y asegurarnos que por cada evento NOTE ON, exista un ControllerEvent disparado en la misma tonada. ¿Cómo nos aseguramos de esto? Pues debemos agregarlo a la pista tal como en otros eventos. En otras palabras, nuestra secuencia de música iría así:

BEAT I - NOTE ON, CONTROLLER EVENT

BEAT 2 - NOTE OFF

BEAT 3 - NOTE ON, CONTROLLER EVENT

BEAT 4 - NOTE OFF

…y así.

Antes que nos sumerjamos de lleno en el programa, aprendamos una manera fácil de crear mensajes/eventos MIDI ya que en este programa usaremos muchos de ellos.

**Lo que el programa musical necesita hacer**

|  |  |
| --- | --- |
|  | Crear una serie de MIDI mensajes/eventos para tocar notas aleatorias en un piano. (O cualquiera será el instrumento que elijas) |
|  | Registrar un listener para los eventos. |
|  | Iniciar la secuencia de reproducción |
|  | Cada vez que el método event handler sea invocado, dibuja un rectángulo aleatorio, dibujarlo en el panel y llamar al método repaint(). |

**Lo construiremos en tres iteraciones**

|  |  |
| --- | --- |
|  | Versión 1: Código que simplifique crear y añadir eventos MIDI, ya que necesitaremos muchos de ellos. |
|  | Versión 2: Registrar y capturar los eventos, pero sin gráficos. Imprimir un mensaje en la línea de comandos en cada tonada. |
|  | Versión 3: Manos a la obra, agregar gráficos al punto 2. |

**Una forma sencilla de crear mensaje/eventos**

En este momento, crear y añadir mensajes y eventos a una pista es tedioso. Para cada mensaje, tenemos que crear una instancia (en este caso ShortMessage), invocar callMessage(), crear un MidiEvent para el mensaje, y agregar el evento a la pista. En el código del último capítulo, realizamos cada paso por cada mensaje. Eso significa ocho líneas de código solo para crear una nota, y luego detener la reproducción. Cuatro líneas para agregar un evento NOTE ON, y cuatro líneas para agregar un evento NOTE OFF.

ShortMessage a = **new** ShortMessage();

a.setMessage(144, 1, note, 100);

MidiEvent noteOn = **new** MidiEvent(a, 1);

track.add(noteOn);

ShortMessage b = **new** ShortMessage();

b.setMessage(128, 1, note, 100);

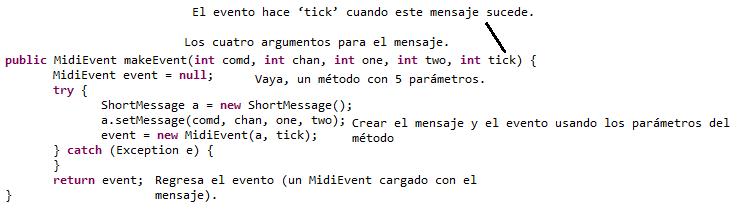
MidiEvent noteOff = **new** MidiEvent(b, 16);

track.add(noteOff);

**Lista de cosas que deben pasar cada evento**

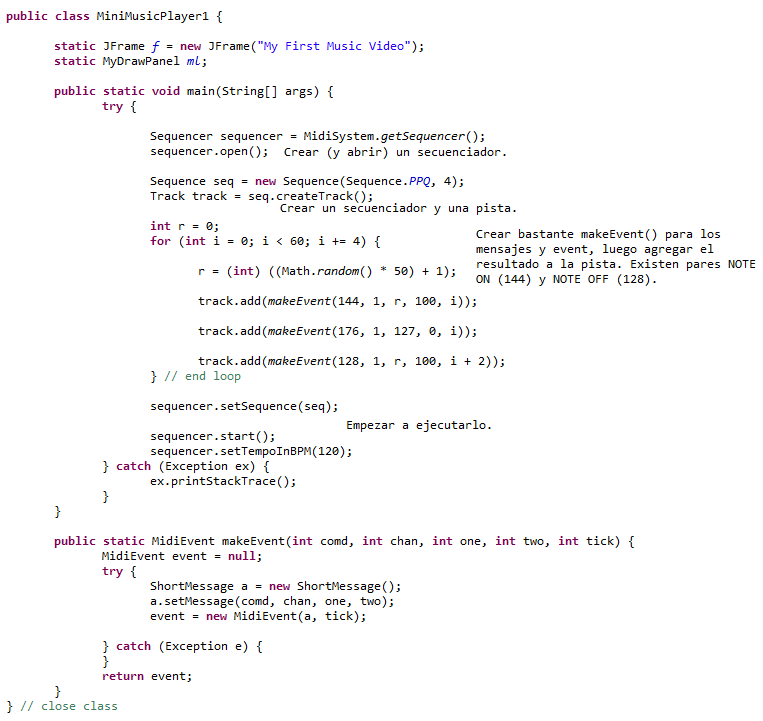
|  |  |
| --- | --- |
|  | Crear una instancia de mensaje  ShortMessage first = **new** ShortMessage(); |
|  | Invocar setMessage() con las instrucciones.  first.setMessage(192, 1, instrument, 0); |
|  | Crear una instancia de MidiEvent para el mensaje  MidiEvent noteOn = **new** MidiEvent(first, 1); |
|  | Agregar el evento a la pista.  track.add(noteOn); |

**Construyamos un método estático de ayuda que crea el mensaje y regrese un MidiEvent**

**

**Ejemplo: cómo usar el nuevo método estático makeEvent()**

No existen event handler o gráficos aquí, solo una secuencia de quince notas que se siguen la escala. El objetivo de este código es simplemente aprender a cómo usar el método que makeEvent() que creamos anteriormente. El código para las dos siguientes versiones es más pequeño y simple gracias a este método.



**Versión dos: Registrar y obtener ControllerEvents**

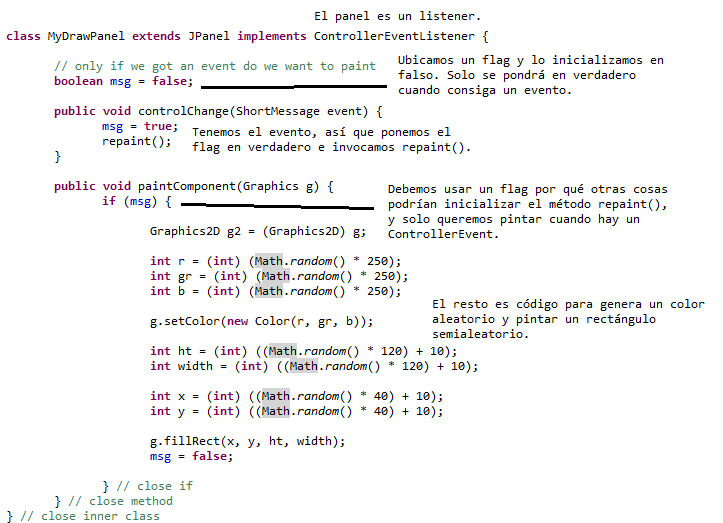
**

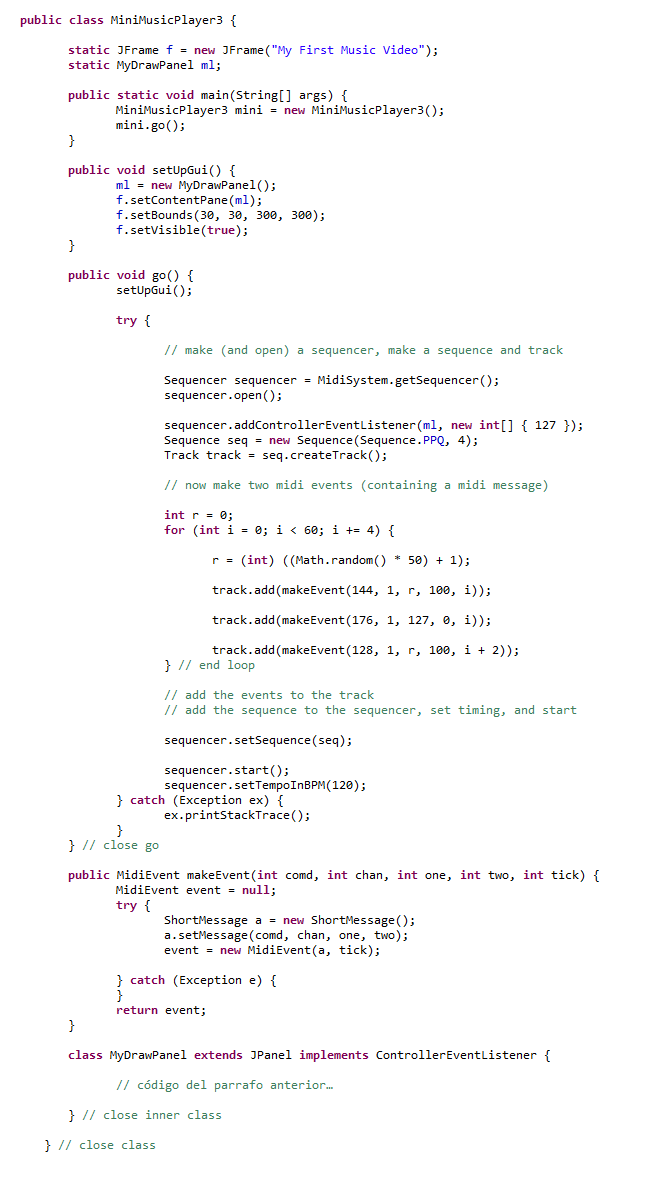
**Versión tres: pintando gráficos a los compas de la música**

Esta versión final se basa en la versión dos, al agregar las partes de GUI. Construimos un frame, agregamos un panel, y cada vez que capturemos un evento, pintamos un nuevo rectángulo y aplicamos repaint() a la pantalla. El único cambio a la versión dos es que las notas se reproducen aleatoriamente como opuestas a simplemente mover la escala.

El cambio más importante al código (mas allá de construir una simple GUI) es que crearemos el panel implementando el ControllerEventListener, que es más importante que el programa mismo. Por tanto cuando el panel (una clase interna) tenga el evento, sepa como pintar el rectángulo. La versión completa del código para esta versión está en la siguiente página.

**La clase interna que usa panel**





Notas:

<http://www.eclipse.org/windowbuilder/>

Para instalar

http://download.eclipse.org/windowbuilder/WB/release/R201109201200/3.7/