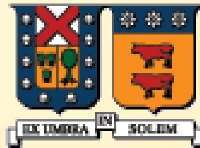


UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Programación “Conducida por eventos” Event-driven programming

Agustín J. González

ELO329/ELO330

Introducción

- Este tema tiene su aparición en la programación de interfaces gráficas de usuarios.
- Los programas de consolas típicos siguen un flujo secuencial en el que típicamente se tienen ciclos:
  - entrada->procesamiento->salida
- Cuando programamos una Interfaz Gráfica de Usuario (GUI: Graphics User Interface) debemos tomar en cuenta la variedad de posibles interacciones con el usuario.
- En lugar de un único flujo de entrada de datos por consola, las GUIs permiten muchas más acciones del usuario.
  - Por ejemplo: es posible presionar botones gráficos, escribir texto en un campo de texto, o mover alguna scrollbar.
  - ¿Cómo podemos estar atento a tanta cosa al mismo tiempo?
- La GUI del programa debe responder bien a todos estos eventos.

Modelo

- Una forma de manejar todo tipo de posibles interacciones de usuarios es el uso de interrupciones.
- De esta manera la CPU no pierde tiempo “mirando” los posibles eventos de usuarios, sino simplemente responde al evento y reanuda su procesamiento normal (otras tareas).
- Comúnmente, lenguajes de programación no dan acceso directo a eventos asincrónicos.
- Lenguajes como Java nos permiten definir y manejar interrupciones o eventos por software.
- La API de Java permite a los programadores crear clases de objetos, llamados **listeners**, que responden a interrupciones causadas por la GUI.
- La API de Java tiene **interfaces** que deben ser implementadas por las clases listener.
- Los métodos de la interfaz (“equivale a la rutina de servicio de interrupción”) son llamados cuando un evento específico ocurre.

Esquema para la Atención de Eventos

- Las componentes gráficas y los datos del evento son objetos de las clases definidas en la API de Java.
- Los listener son objetos de las clases definidas por el usuario y que implementan la interfaz definida por la API.

Ejemplo

- Veamos el caso de una ventana de nivel superior en Java (aquellas que se pueden mover libremente en el desktop) CloseableFrame.java

```
import java.awt.event.*;
import javax.swing.*;

class CloseableFrame extends JFrame {
 public CloseableFrame() {
  setTitle("My Closeable Frame");
  setSize( 300, 200);
  // cause window events to be sent
  // to window listener object
  addWindowListener( new MyWindowListener());
 }
}
```

Ejemplo

```
class MyWindowListener implements WindowListener {
  // Do nothing methods required by interface
  public void windowActivated( WindowEvent e) {}
  public void windowDeactivated( WindowEvent e) {}
  public void windowIconified( WindowEvent e) {}
  public void windowDeiconified( WindowEvent e) {}
  public void windowOpened( WindowEvent e) {}
  public void windowClosed( WindowEvent e) {}

  // override windowClosing method to exit program
  public void windowClosing( WindowEvent e) {
    System.exit( 0); // normal exit
  }
}
class Main {
  public static void main( String[] args) {
    CloseableFrame f = new CloseableFrame();
    f.show();         // makes the frame visible
  }
}
```

Relación de clases (generada por Jprasp)

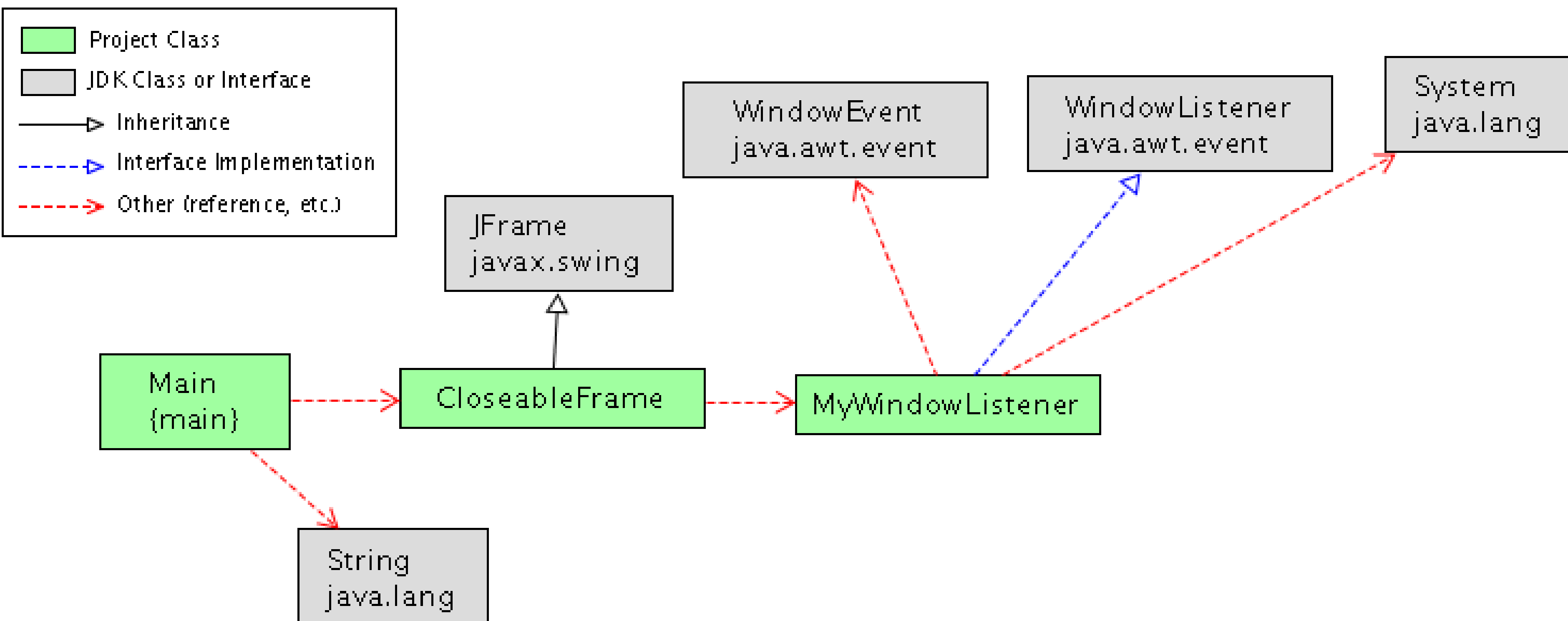
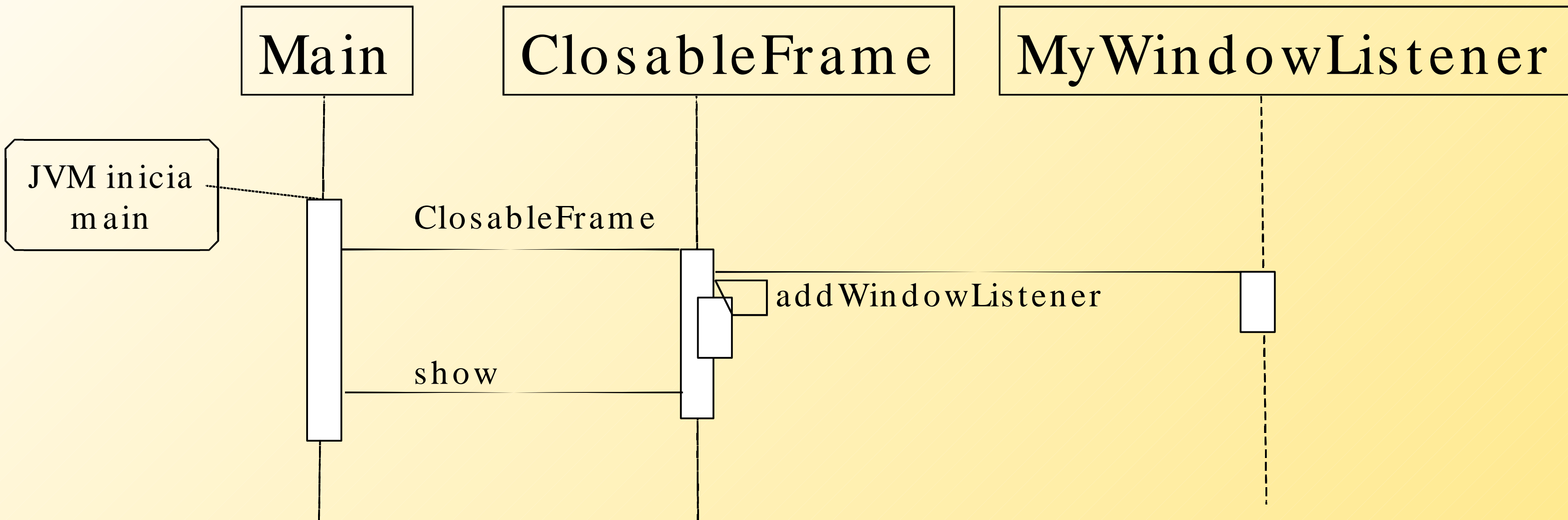


Diagrama de secuencia para creación de ventana



Explicación

- Objetos en la clase ClosableFrame causan que una ventana aparezca en la pantalla del usuario. La aplicación puede crear tantas ventanas como lo desee creando múltiples objetos ClosableFrame.
- Un objeto MyWindowListener es registrado con addWindowLister (es como configurar quien atenderá las interrupciones). Cuando el evento ocurre, la máquina virtual Java llama automáticamente al método apropiado de la interfaz WindowListener.
- La interfaz WindowListener es implementada por la clase MyWindowListener, así su instancia puede responder a los eventos de la ventana en la cual él fue registrado.
- Hay siete métodos en la Interfaz WindowListener (Ver API) Aquí sólo nos interesamos por el evento cierre de ventana.
- La mayoría de las otras interfaces para eventos no difieren mucho. Veremos otro ejemplo.

Entrada en Campo de texto

- Supongamos que queremos leer lo ingresado en un campo de texto y luego copiarlo en un label (rótulo).
- Mimic.java

Diagrama de Clases de Mimic.java

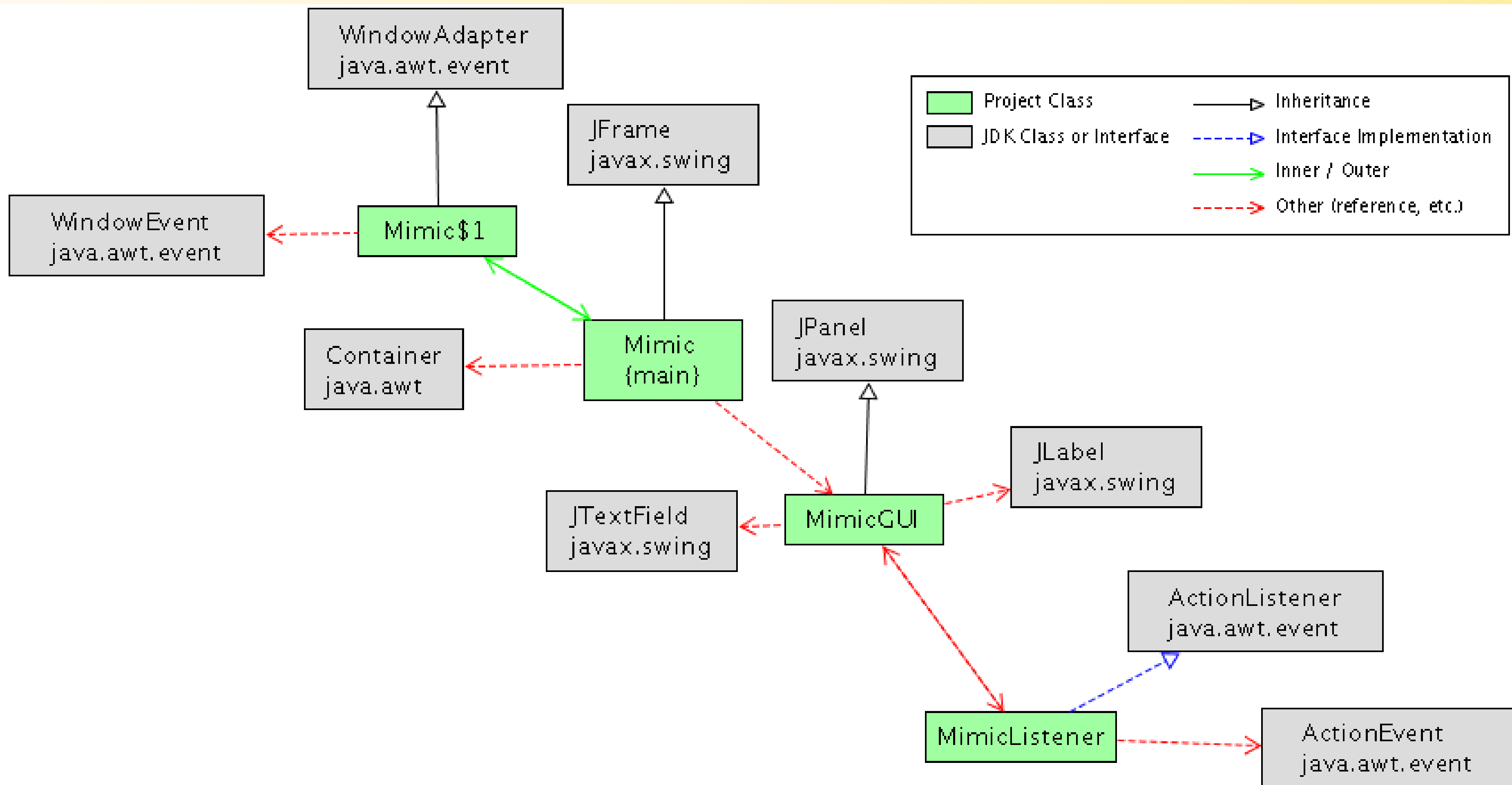
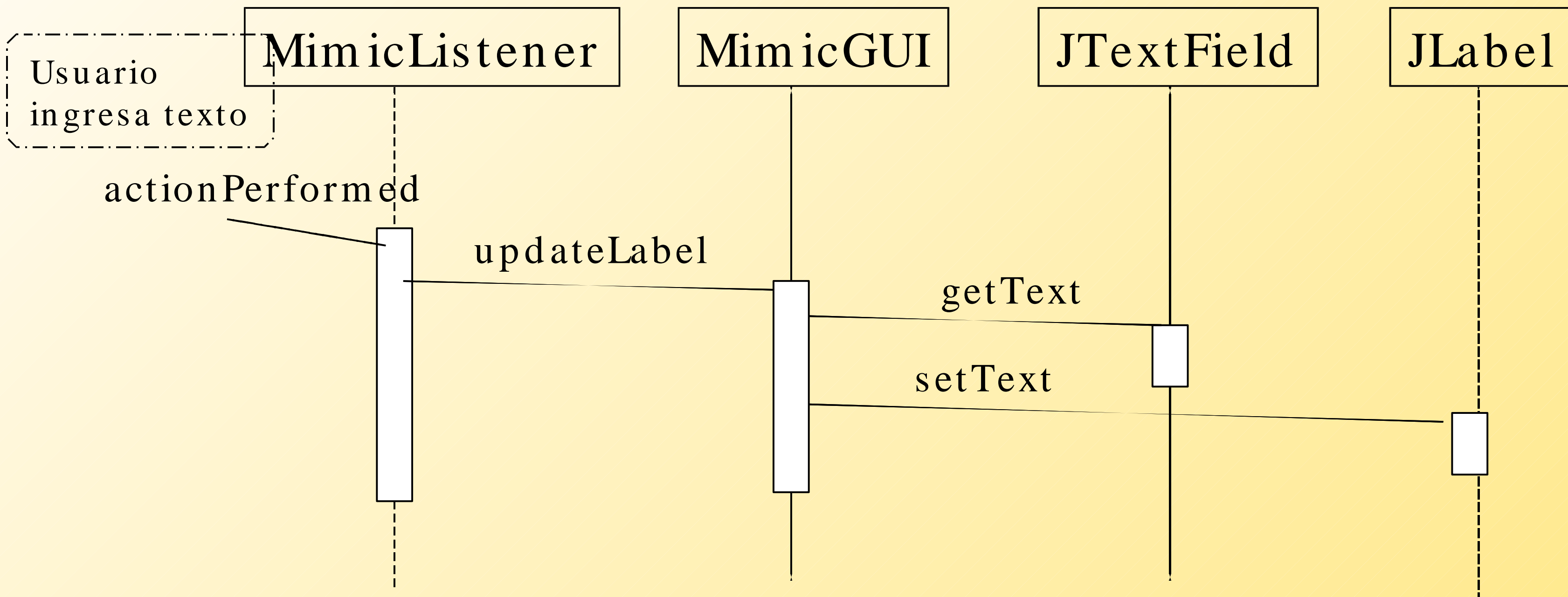


Diagrama de secuencia:



- Es recomendable tener bien clara esta relación, y ayuda hacer este dibujo antes de hacer el código. Sirve además como documentación.
- JtextField y Jlabel pertenecen al API, luego sólo hay que escribir el código para las otras tres clases.

Explicación del ejemplo

- El listener es registrado con el objeto quote – *comillas* - de la clase JTextField en el constructor MimicGUI() al ejecutar quote.addActionListener(listener). Cuando listener es registrado, éste es agregado a una lista interna que mantiene los objetos que deben ser notificados cuando un evento ocurre. Podemos tener más de un listener registrado por evento.
- Cuando el evento ocurre (por ejemplo, presionamos return en la ventana), el método listener.actionPerformed(ActionEvent event) es llamado por la JVM. Notar que los datos sobre el evento son pasados en el objeto pasado como parámetro al método.
- El código en el método del listener maneja el evento llamando a gui.updateLabel() , el cual hace eco del contenido del campo texto en el rótulo puesto en la ventana.

Algunas recomendaciones

- No es estrictamente necesario poner la descripción de las componentes de la GUI en una clase separada, pero es buena idea. Incluso puede ser conveniente ponerla en una archivo separado para así separar la presentación e interacción con el usuario del procesamiento o cálculo interno.
- La clase JTextField es incluso más completa. Nos permite recibir una notificación cada vez que un caracter es ingresado.
- Los cambios requeridos para ello se muestras en MimicCharbyChar.java