

# G51PRG: Introduction to Programming Second semester GUI continued

Natasha Alechina  
School of Computer Science & IT  
nza@cs.nott.ac.uk

## Previous lecture (before Easter)

- AWT and Swing
- Some simple components
- Layout managers

Hopefully, can now create a window with buttons and text fields:



GUI continued

2

## Plan of today's lecture

- How to make GUI interact with the user - respond to events?
- Java Event Model
- Nested classes and anonymous classes
- Some hints for the Browser exercise

GUI continued

3

## Java Event Model

- GUI is an *event driven* application
- *source* of an event can be a button, a window, a text field ...
- *event* is an object generated by the source
- the source sends events which it generates to *listeners* which are registered with the source
- the listeners deal with the event

GUI continued

4

## Difference from Visual Basic

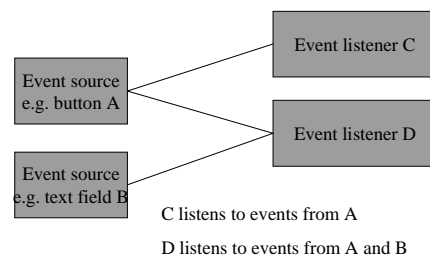
- In many GUI programming languages, like VB, buttons have a method to respond to events, e.g.  

```
public void buttonPressed() {  
    // what to do if this happens  
}
```
- **not so in Java:** components themselves do not do event handling; they may register other objects with them, so called *event listeners*, which have methods to respond to events.

GUI continued

5

## Java Event Model



GUI continued

6

## How to make it work

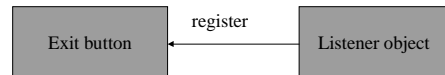
- **Write a class which defines the event listener.** If you need a listener for an Action Event, like a button click, implement `ActionListener`. If you need a listener for window events (e.g. closing a window), implement `WindowListener`. If you need a listener for mouse events, implement `MouseListener` or `MouseMotionListener`. **This involves implementing a method which responds to the event**, e.g. `actionPerformed()` of the `ActionListener`.
- Create an instance of that class. Add it to the component which needs an event listener.

GUI continued

7

## Example

- Suppose we want to exit if the user clicks "Exit" button.
- We need to register an event listener with the "Exit" button.

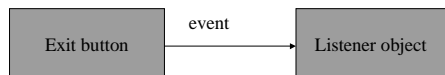


GUI continued

8

## Example

- The listener object should be an instance of a class which implements `ActionListener` interface.
- When it gets an `ActionEvent` from the button, it executes the `actionPerformed()` method which calls `System.exit()`.



GUI continued

9

## Writing a listener class

- Write a new class which implements the action listener (so it has `actionPerformed(ActionEvent e)` method)
- `actionPerformed(ActionEvent e)` calls `System.exit(1)`
- Create an instance `x` of this class
- Register this instance `x` with the exit button
- `actionPerformed()` will be invoked by `x` when the exit button generates an event (is clicked).

GUI continued

10

## Example

```
class NewListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(1);
    }
}
```

And then add these two lines in the constructor of `SimpleGUI`:

```
NewListener x = new NewListener();
exitB.addActionListener(x);
or simply
exitB.addActionListener(new NewListener());
```

GUI continued

11

## Alternatively

- Make the main window class (extending `JFrame`) to double as an event listener
- Get it to implement the action listener
- Define `actionPerformed()` method which when an action event is generated calls `System.exit(1)`.
- The current instance of the frame is registered with the exit button as an event listener (so it listens to events generated by the exit button).

GUI continued

12

## Example: frame is the listener

```
class SimpleGUI extends JFrame implements
    ActionListener {
    // buttons and text fields omitted

    public SimpleGUI(){
        // creation of buttons, text fields and panels
        // is omitted here
        exitB.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        System.exit(1);
    }
}
```

GUI continued

13

## Distinguishing events from different sources

- If one listener is registered with several event sources
- How can it react in one way if exit button is pressed, and in another way if some other button is pressed?

GUI continued

14

## EventObject

The **Event** class is the abstract root class from which all event state objects shall be derived (GUI event from **AWTEvent**).

Field:

- **Object** source

Methods:

- **Object** getSource()
- **String** toString()

GUI continued

15

## Listening on several event sources

```
public void actionPerformed(
    ActionEvent e){
    if(e.getSource().equals(exitB)) {
        System.exit(1);
    }
    if(e.getSource().equals(okB)) {
        label.setText("Hello,"+tf.getText());
    }
}
```

GUI continued

16

## Adapter Classes

Same function as Listener Interfaces, but classes which provide empty implementation of methods.

Extending an **Adapter** is sometimes more convenient than implementing an interface as one does not have to provide implementation for all its methods.

GUI continued

17

## Example: WindowAdapter

- **WindowAdapter** is a class which implements a **WindowListener** interface providing an empty implementation for all its methods.
- If you only need to implement one method of the **WindowListener**, for example **windowClosing()**, you would have to provide a dummy implementation for all other methods.
- Instead, you could extend **WindowAdapter** and overwrite just one method.

GUI continued

18

## Example

```
class MyClass extends WindowAdapter {  
  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(1);  
    }  
  
}
```

GUI continued

19

## Problems with examples before

- Have to keep inventing new class names for all those listeners
- New classes litter the program (make it long and unreadable, or scatter around in the working directory)
- If you need to change other components (not just to exit) then they should be “visible” from the listener class, so can’t be declared private in the main class.

GUI continued

20

## Answer: inner classes

- Recall that classes can be declared inside other classes, and inside methods and blocks in other classes.
- All those classes are called *inner classes* (in Java Gently; in other sources you may meet *nested classes* as a generic term).

```
class TopLevelClass {  
    // some code  
    class NestedClass {  
        // some code for the NestedClass  
    } // end of NestedClass  
} // end of TopLevelClass
```

GUI continued

21

## When compiled

- `TopLevelClass.class`
- `TopLevelClass$NestedClass.class`

GUI continued

22

## Types of inner classes

- An inner class can be declared static: then it basically has the same status as a top-level file and is just grouped inside another file for convenience. Java Gently calls such classes *nested classes*.
- If an inner class is not declared static, it has a subordinate status. Java Gently calls it a *member class*. It can access all members of the encompassing class, including private members.
- *Local class* is a member of a method or a block.
- *Anonymous class* is a local class without a name.

GUI continued

23

## Example of anonymous class

```
f.addWindowListener (new WindowAdapter ()  
// class definition follows:  
{  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(1);  
    }  
});
```

Here, the class definition is in italics; the class is not given a name and its definition is nested in a method call.

GUI continued

24

## What is it equivalent to

```
class X extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(1);
    }
}

f.addWindowListener (new X());
```

GUI continued

25

## Event thread

When Swing is used to create a window, a separate thread of execution is created, which enters an infinite loop waiting for events to happen.

Unlike AWT, Swing is not thread safe --- the access to the contents of the window is not synchronised. You should take care that after the window is displayed, all changes in the appearance of the window are done within the event thread and no other thread has access to them as well.

GUI continued

26

## Hint for the Browser exercise

- The previous example shows how to get text (a **String**) from a text field in response to a button click.
- You can use the resulting **String** to create a **URL** object.
- Then you can get the contents of the page at the **URL** as a **String** (see lecture on networking).
- If you just wanted to display that **String** as plain text, you could use **setText()** method of **JEditorPane** to do this, as we did with a label in the previous example.
- Unfortunately, displaying html is not that simple (read **setText()** method description in **JEditorPane** API description).

GUI continued

27

## Hint for the Browser exercise cont.

- I don't want to go into details of **JTextComponents** and their document models, which are needed to do this properly and adjust the document model for every new html page.
- Instead, here is an ugly but simple method for refreshing pages.

GUI continued

28

## Hint for the Browser exercise cont.

Suppose your **JEditorPane** object is called **jpane**. Every time you need to display a new html **String s** in **jpane**, do:

```
jpane.setVisible(false);
jpane = new JEditorPane("html/text", s);
this.getContentPane().add("Center", jpane);
jpane.setVisible(true);
```

GUI continued

29

## Hint for the Browser exercise cont.

Finally, you may use

```
jpane = new JEditorPane(url);
```

where **url** is a **URL** object. This would always work with simple web pages but you *may* get problems with web pages using style sheets, like the School web page. Again there are fixes for this but I'd like to keep things simple.

GUI continued

30

## Summary

The main points are:

- Java Event Model
- Communication by sending objects
- Listeners register with event sources and handle events.
- Anonymous classes are used to write compact code for event listeners.

For more examples see *Java Gently*, Chapter 11 or Sun Java tutorial.