9 - Event-Driven Programming

Computer Science Department
California State University, Sacramento

Overview

- **Traditional vs. Event-Driven Programs**
- **Events**
- **Event Listeners:**
  - **CN1 `ActionListener` interface**
  - **Adding listeners to event-generators**
  - **Command design pattern, CN1 `Command` class, key bindings**
  - **Pointer handling**

Traditional vs. Event-Driven

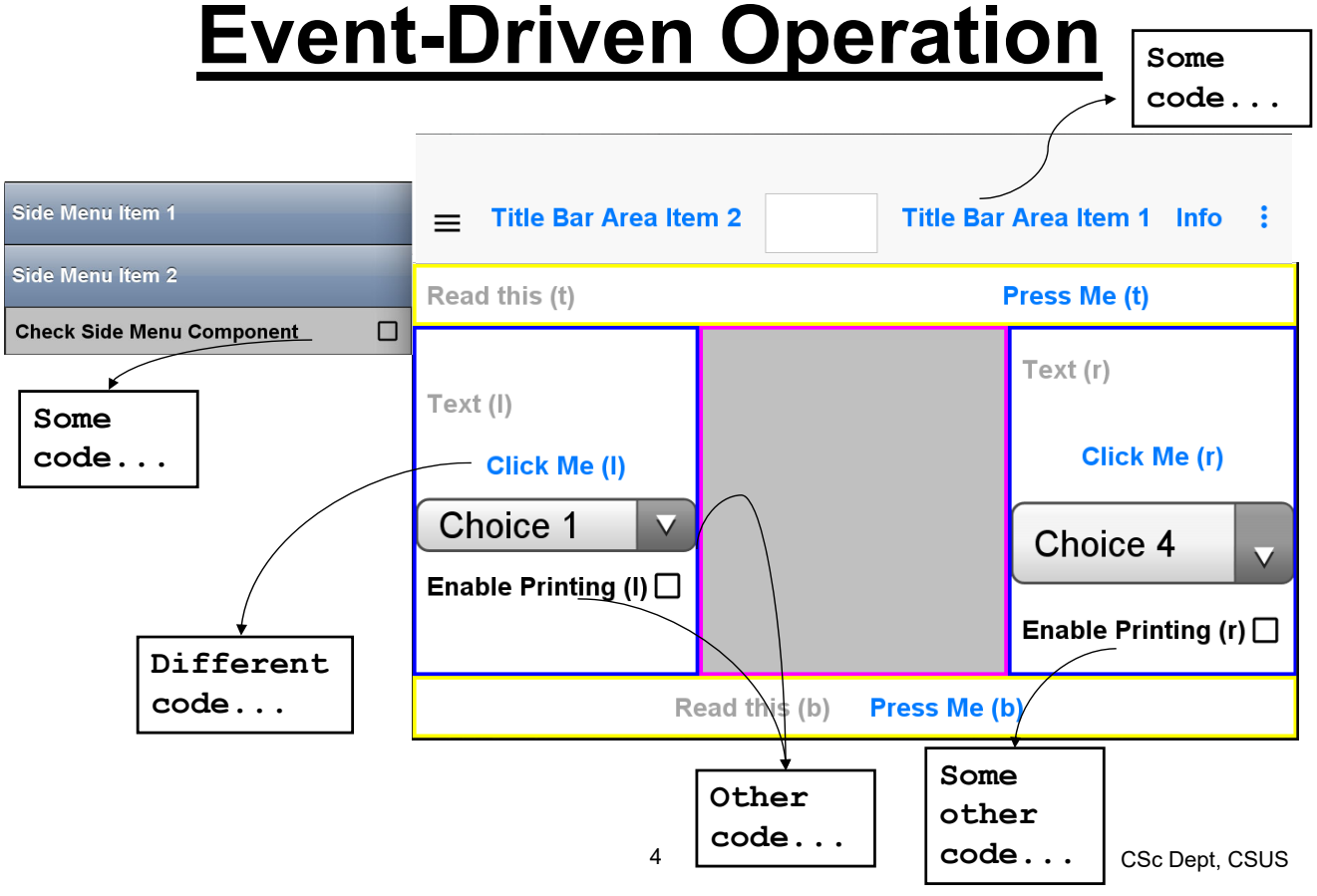
- Traditional program organization:

```
loop {
     get some input ;
     process input ;
     produce output ;
}
until (done);
```

- Event-driven program organization:

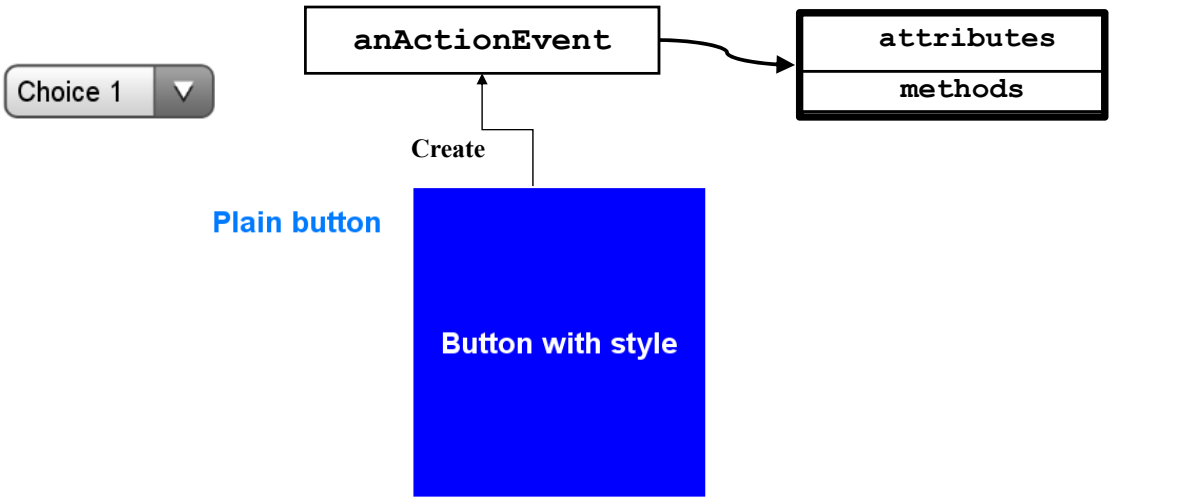
```
create a form ;
create some controls (buttons, etc.) ;
add controls to form ;
make the form visible ;
```


Event-Driven Operation



Event Objects

Activating a component and use of keys and the pointer create an object of type **`ActionEvent`**



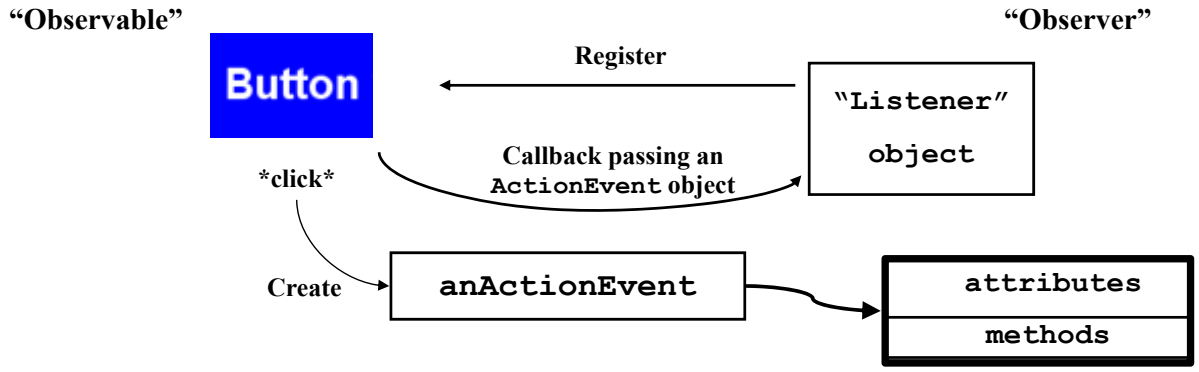
Event Objects (cont.)

- CN1 does not have different type of event objects as in Java (e.g. **`ActionEvent, MouseEvent, KeyEvent,`** etc.)

- Activating a component (e.g., pushing a button), using a key (pressing, releasing), or use of pointer (pressing, releasing, dragging, etc.) ALL produce an object of type **`ActionEvent`**.

Event Listeners

- Event-driven code attaches *listeners* to *event-generators*
- Event-generators make *call-backs* to listeners



ActionListener Interface

- **Listeners must implement interface `ActionListener` (built-in in CN1):**

```
interface ActionListener
{
   public void actionPerformed (ActionEvent e);
}
```

Approaches for Creating a Listener

- (1) Have a class that implements **`ActionListener`**. Two options:
  - (1a) Your listener is different than the class that creates the components
  - (1b) You make the class that creates components (e.g., the class that extends **`Form`**) your listener
- (2) Have a class that extends built-in **`Command`** class. This approach uses the Command design pattern.

Approach (1a)

```
import com.codename1.ui.events.ActionEvent;
import com.codename1.ui.events.ActionListener;

/** This class acts as a listener for ActionEvents.
 *  It was designed to be attached and respond
 *  to button-push events.
 */

public class ButtonListener implements ActionListener{

// Action Listener method:  called from the object being observed
// (e.g. a  button) when it generates an "Action Event"
// (which is what a button-click does)

  public void actionPerformed(ActionEvent evt) {
    // we get here because the object being observed
    // generated an Action Event
   System.out.println ("Button Pushed...");
  }

}
```

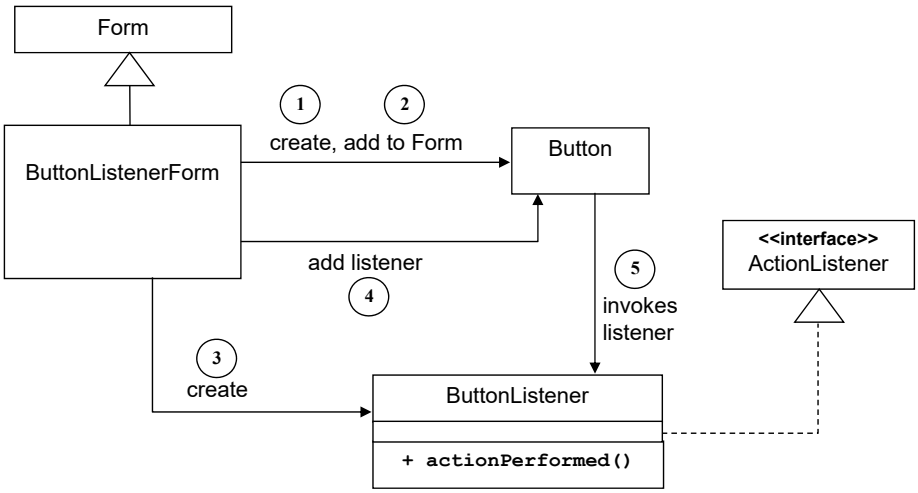
Using the Listener

Inside a class that extends from Form:

```
/** Code for a form ((ButtonListenerForm) with a single Button to which is attached an
 *  ActionListener. The button action listener is invoked whenever the
 *  button is pushed.
 */
//create a button
Button myButton = new Button("Button");
//…[style the button and add it to the form]
//create a separate ActionListener for the button
ButtonListener myButtonListener = new ButtonListener ();
//register the myButtonListener as an Action Listener for
//action events from the button
myButton.addActionListener(myButtonListener);
```

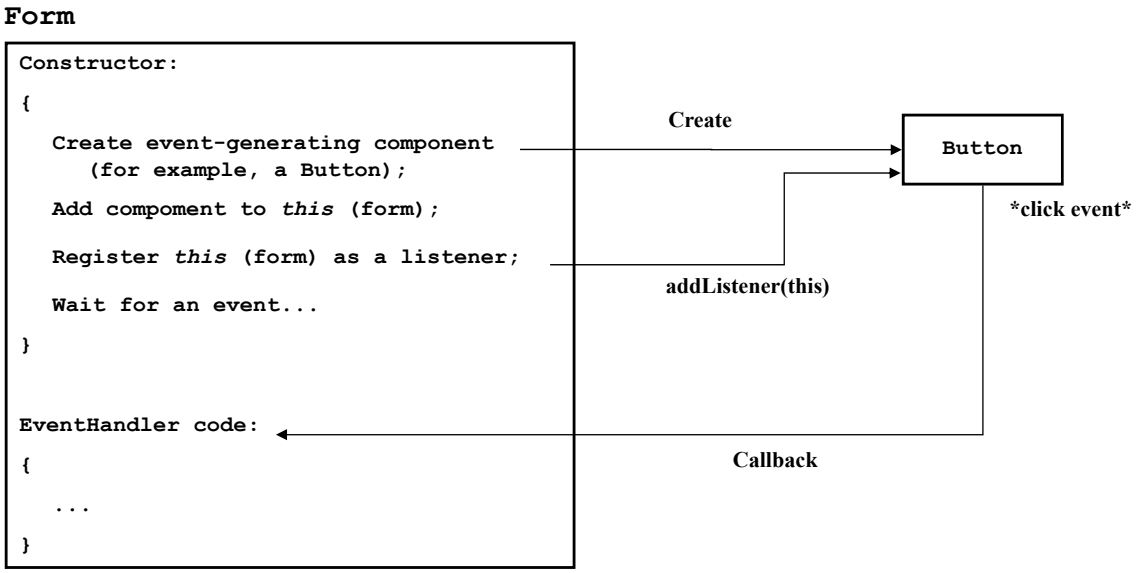

Listener Class Organization

- UML for the previous code:



Approach (1b)

Forms can listen to their own components!

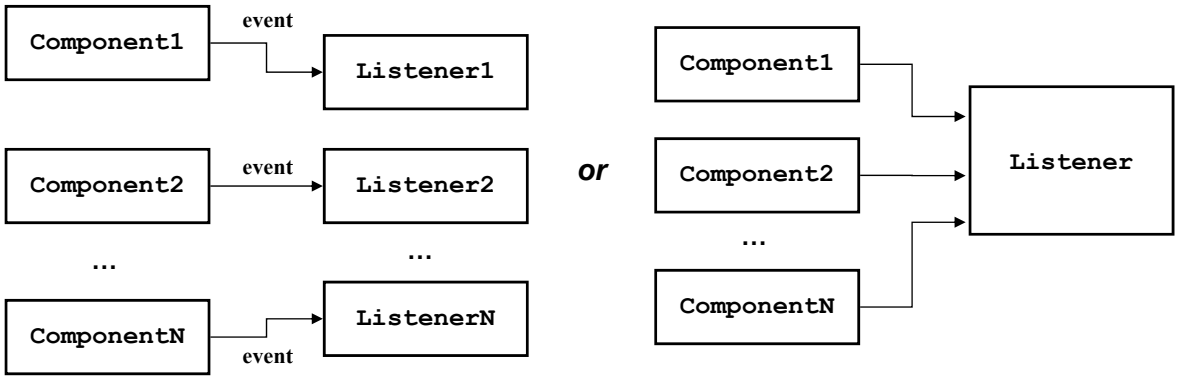


ActionListener Form Example

```
/** Code for a form with a single button which the form listens to. */
public class SelfListenerForm extends Form implements ActionListener {
  public SelfListenerForm () {
     // create a new button
     Button myButton = new Button ("Button");
     // add the button to the content pane of this form
     add(myButton);
     // register THIS object (the form) as an Action Listener for
     //  action events from the button
     myButton.addActionListener(this);
     show();
  }
  // Action Listener method: called from the button because
  // this object -- the form -- is an action listener for the button
  public void actionPerformed (ActionEvent e) {
     System.out.println ("Button Pushed (printed from the form)...");
  }
}
```

Multiple Event Sources

- *Approaches:*
  - (1a) requires multiple separate listeners
  - (1b) requires one listener
    - it would need to be able to *distinguish event source*



Let's consider this second option …

Multiple Component Listener

```
/* Code for a form with multiple buttons which have action handlers in the form */
public class MultipleComponentListener extends Form implements ActionListener{
  private Button buttonOne = new Button("Button One"); //need to make this button a class field
  public MultipleComponentListener() {
    setTitle("Multiple Component Listener");
    Button buttonTwo = new Button("Button Two");
    //...[set styles of the buttons and add them to form]
    buttonOne.addActionListener(this);
    buttonTwo.addActionListener(this);
    show();
   }
  public void actionPerformed(ActionEvent evt) {
    if(evt.getComponent().equals(buttonOne)){ //buttonOne must be a class field
      System.out.println ("Button One Pushed (printed from the form using
                                                          getComponent())...");
    }
    else if(((Button)evt.getComponent()).getText().equals("Button Two")){
      //if we change the label of the button, this code would not work
      System.out.println ("Button Two Pushed (printed from the form using
                                                  getComponent().getText())...");
    }//else if
  }//actionPerformed
}//class
```

Multiple Component Listener (cont.)

- ***actionPerformed()*** would get bigger and bigger… more and more unwieldy as we have more components in the form.
- A better approach is using combination of approaches (1a) and (1b):

Command Design Pattern

which is the Approach (2).

(use one listener for all related components, but you can have multiple listeners for different groups of components)

Anonymous `Command` Sub-Class

We can extend from **`Command`** in a separate .java file and then instantiate an object of this sub-class in a separate .java file.

Or… we generate an object of an anonymous sub-class of **`Command`** in the same .java file.

First option (which is used in the "Command Design Pattern" code example) is recommended…

See the next slide for the second option…But do **NOT use** the second approach (**anonymous sub-classing**) in the assignments!

Anonymous Command Sub-Class (cont.)

```
/* Code for a form that creates an object of anonymous sub-class of the Command */
//create a Toolbar called myToolBar and add it to the form
//create the object (called inforTitleBarAreaItem) of anonymous sub-class of Command
Command infoTitleBarAreaItem = new Command("Info") {
      public void actionPerformed(ActionEvent ev) {
            String Message = "I provide information.";
            Dialog.show("Info", Message, "Ok", null);
       }
};
myToolbar.addCommandToRightBar(infoTitleBarAreaItem);
```


Adding a Command to Side Menu Component

```
/* Code for a form which has a CheckBox as a side menu item*/
public class SideMenuItemCheckForm extends Form{
  private Label checkStatusVal = new Label("OFF");
  private Toolbar myToolbar = new Toolbar();
  public SideMenuItemCheckForm() {
   setToolbar(myToolbar);//...[add some side menu items]
   CheckBox checkSideMenuComponent = new CheckBox("Side Menu Item Check");
   checkSideMenuComponent.getAllStyles().setBgTransparency(255);
   checkSideMenuComponent.getAllStyles().setBgColor(ColorUtil.LTGRAY);
   //create a command object and set it as the command of check box
   Command mySideMenuItemCheck = new SideMenuItemCheck(this);
   checkSideMenuComponent.setCommand(mySideMenuItemCheck);
   //add the CheckBox component as a side menu item
   myToolbar.addComponentToSideMenu(checkSideMenuComponent);
   //add labels to indicate the check box status on the form, have two labels to display
   //text (which does not change) and value (which changes) parts separately,
   //only update the value label when the check box status changes
   Label checkStatusText = new Label("Check Box Status:"); //label for the text
   //label for the value is defined above (checkStatusVal)
   //...[add labels to the form and show the form]
  }
```

continued…

Adding a Command to Side Menu Component

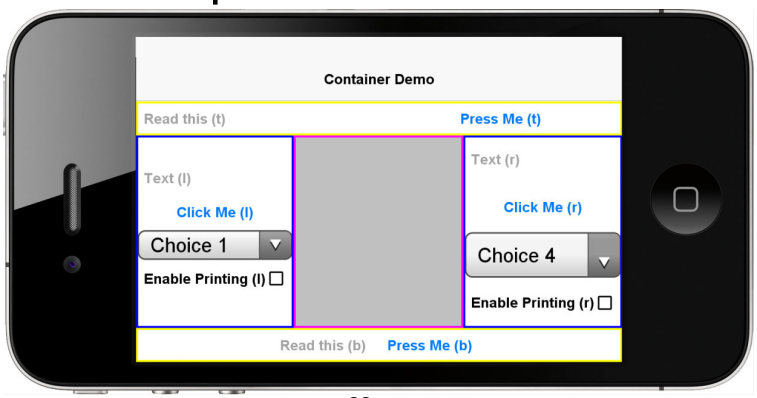
continued…

```
public void setCheckStatusVal(boolean bVal){
  if (bVal)
    checkStatusVal.setText("ON");
  else
   checkStatusVal.setText("OFF");
  revalidate ();} //make sure to call revalidate(), if cannot see updated values properly
}// SideMenuItemCheckForm class   ---------- below is the code for the command class

public class SideMenuItemCheck extends Command {
  private SideMenuItemCheckForm myForm;
  public SideMenuItemCheck (SideMenuItemCheckForm fForm){
    super("Side Menu Item Check"); //do not forget to set the "command name"
    myForm = fForm;}
@Override
public void actionPerformed(ActionEvent evt){
  if (((CheckBox)evt.getComponent()).isSelected())//getComponent() returns the component
                                                  //that generated the event
    myForm.setCheckStatusVal(true);
  else
    myForm.setCheckStatusVal(false);
  myToolbar.closeSideMenu();} //do not forget to close the side menu
}// SideMenuItemCheck class
```


Component Width and Height

- Layout managers automatically place and size the components.
- Hence, we can only get their correct width and height values after calling **`show()`**.
- Remember the "Container Example" from the "GUI Basics" chapter:



Component Width and Height (cont.)

```
public class FormWithMultipleContainers extends Form{
  Container centerContainer;
  public FormWithMultipleContainers(){
    //create the center container and add it to form
   centerContainer = new Container();
    //... [add the centerContainer to the from, create bottomContainer]
    //create a button and add it to bottomContainer
   Button bPressMeB = new Button("Press Me (b)");
   bottomContainer.add(bPressMeB);
   //...[add the bottom Container to the from,
   //create/add other containers and components and style them all]
   //below line prints incorrect values: 0,0
   System.out.println("Center container width/height (printed BEFORE show()):
      " + centerContainer.getWidth() + " " + centerContainer.getHeight());
   show();
    //below line prints correct width and height
   System.out.println("Center container width/height (printed AFTER show()): "
       + centerContainer.getWidth() + " " + centerContainer.getHeight());
   bPressMeB.addActionListener(new Command("Print center"){
         public void actionPerformed(ActionEvent ev){
         //below line also prints correct width and height
         System.out.println("Center container width/height (printed after
                  button click): " + centerContainer.getWidth() + " " +
                  centerContainer.getHeight());
     }//actionperformed(){
    }//new Command(){
   );//addActionListener(
  }//constructor
}//class
```


Pointer Handling

- Components also generate an **`ActionEvent`** when a pointer is pressed/released or dragged on them.
- **`Component`** class provides:

  `addPointerPressedListener( )`

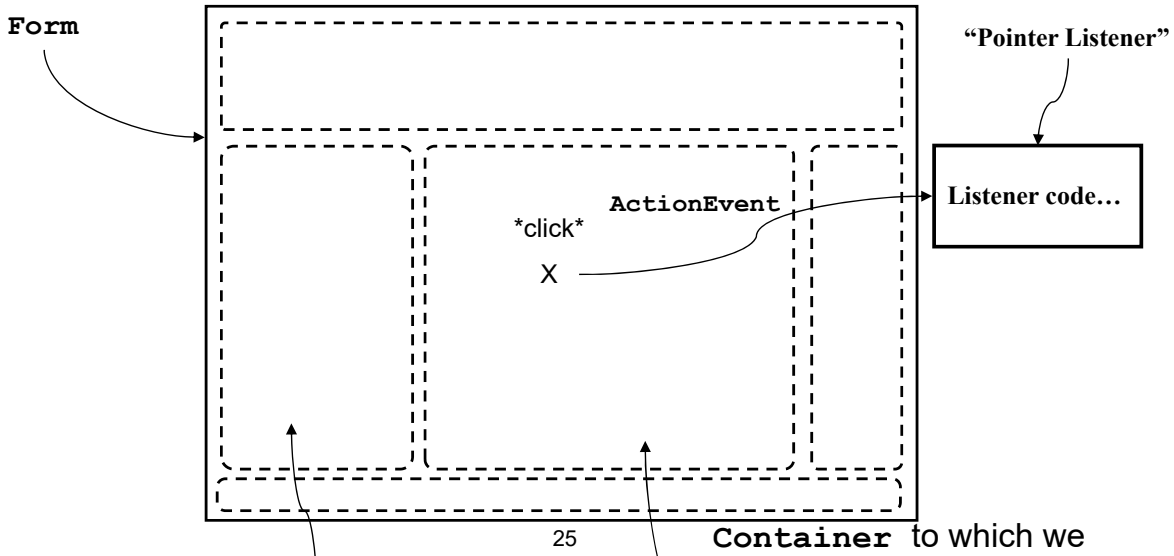
  `addPointerReleasedListener( )`

  `addPointerDraggedListener( )`

…all of which take a parameter of **`ActionListener`** ...

(this means you can attach a **`Command`** and pointer actions can also become a part of Command Design Pattern)

Pointer Handling (cont.)



Container without a listener (clicking on this container would not invoke any action/method)

Container to which we have added a listener (clicking on this container would call **actionPerformed()** of the attached listener)

Pointer Handling (cont.)

- Like action listeners, pointer listeners must also implement **ActionListener** interface:

```
interface ActionListener
{
   public void actionPerformed (ActionEvent e);
}
```

- **ActionEvent** passed to **actionPerformed()** method has **getX()** and **getY()** methods which returns the "screen coordinate" of the pointer location.

Pointer Listener Example

```
/** A Form with a simple pointer-responding container */
public class PointerListenerForm extends Form{
  public PointerListenerForm() {
  //...[set the form layout to borderlayout, generate and style buttons and
  //add them to on north and south containers]
  //have an empty container in the center and add a pointer pressed
  //listener to it
  Container myContainer = new Container();
  PointerListener myPointerListener = new PointerListener ();
  myContainer.addPointerPressedListener(myPointerListener);
  this.add(BorderLayout.CENTER,myContainer);
  //…[add other containers and components to the form]
  }
}
--------
public class PointerListener implements ActionListener {
  public void actionPerformed(ActionEvent evt) {
   System.out.println("Pointer x and y: " + evt.getX() + " " + evt.getY());
  }
}
```


Pointer Listener Example (cont.)

Question:

What happens if I add the listener to the form instead of the container in the form?

```
public class PointerListenerForm extends Form{
  public PointerListenerForm() {
    PointerListener myPointerListener = new PointerListener();
    this.addPointerPressedListener(myPointerListener);
   //…[add containers and components to the form]
 }
}
```

Answer:

Clicking anywhere on the form (including the title bar area) would print out the values…

Adding Listeners for Different Pointer Actions

- There are two approaches:
  - You can add a separate listener for pressed/released/dragged

```
myContainer.addPointerPressedListener(myPressedListener)
myContainer.addPointerReleasedListener(myReleasedListener)
myContainer.addPointerDraggedListener(myDraggedListener)
```

    - This approach requires us to have three separate listener classes.
  - You can have a single listener for all (e.g., self listener) and distinguish between different actions by using **`ActionEvent`**'s **`getEventType()`** method.
    - You need to have if-then-else structure which can get unwieldy if the form is also listening for other event types

Adding Pointer Listener vs Overriding Pointer Methods

- **`Component`** class also has following methods:

  `pointerPressed()`

  `pointerReleased()`

  `pointerDragged()`

  ….all of which gets the parameters which indicate screen location of the pointer…

- If you are extending from a **`Component`** (e.g. **`Form`, `Container`**), you can override these functions. This is the recommended approach since it is easier than adding a listener for each separate pointer action.

Overriding Pointer Methods

```
/* Center container of the form is a PointerContainer which extends from Container */
public class PointerListenerForm extends Form{
  public PointerListenerForm() {
    PointerContainer myPointerContainer = new PointerContainer();
    this.add(BorderLayout.CENTER,myPointerContainer);
    //…[add other containers and components to the form]    }
}
------
/* We can override the pointer methods in the Container */
public class PointerContainer extends Container{
  @Override
  public void pointerPressed(int x,int y){
    System.out.println("Pointer PRESSED x and y: " + x + " " + y);    }
  @Override
  public void pointerReleased(int x,int y){
    System.out.println("Pointer RELEASED x and y: " + x + " " + y);  }
  @Override
   public void pointerDragged(int x,int y){
     System.out.println("Pointer DRAGGED x and y: " + x + " " + y);  }
}
```