CSC 133
Object-Oriented Computer Graphics Programming

Event I

Dr. Kin Chung Kwan

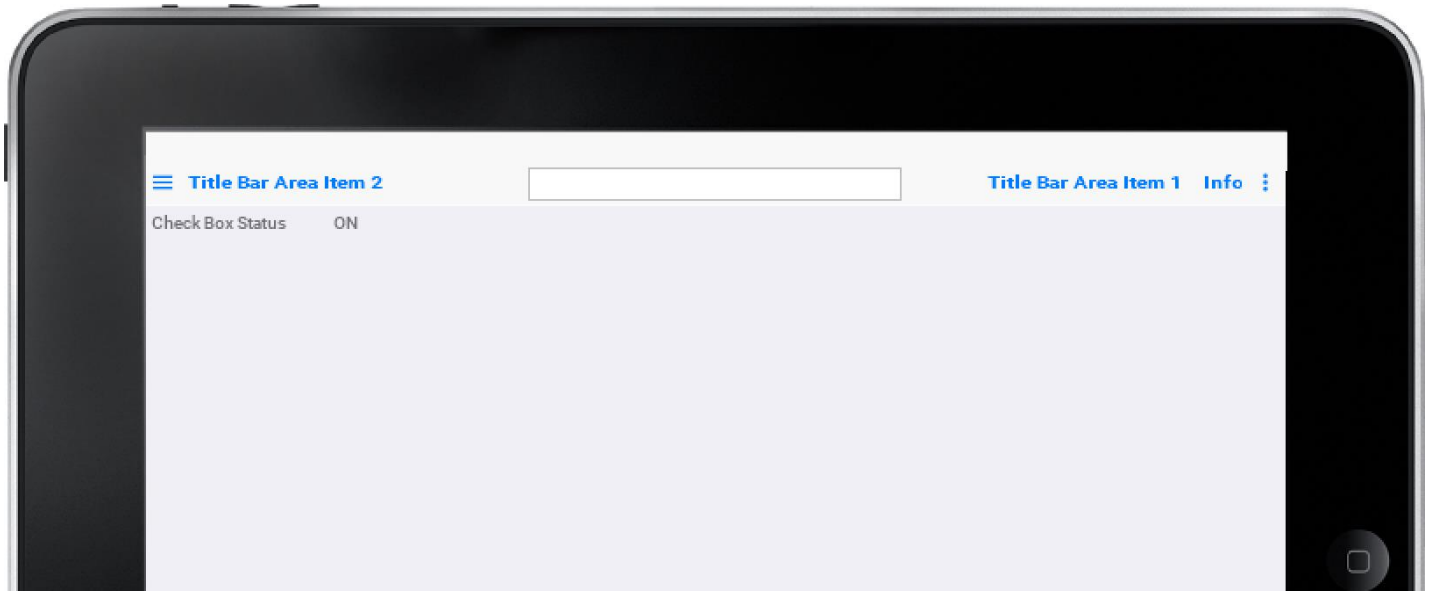
Spring 2023

Computer Science Department
California State University, Sacramento

SACRAMENTO STATE

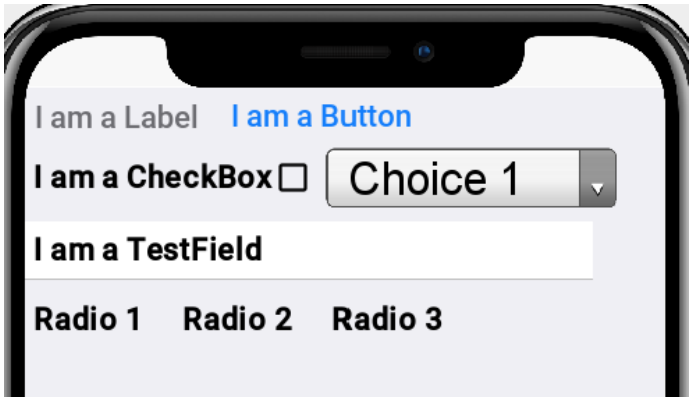
UI

- We built the UI in Codename One
  - `New Component()`
  - `Add()`



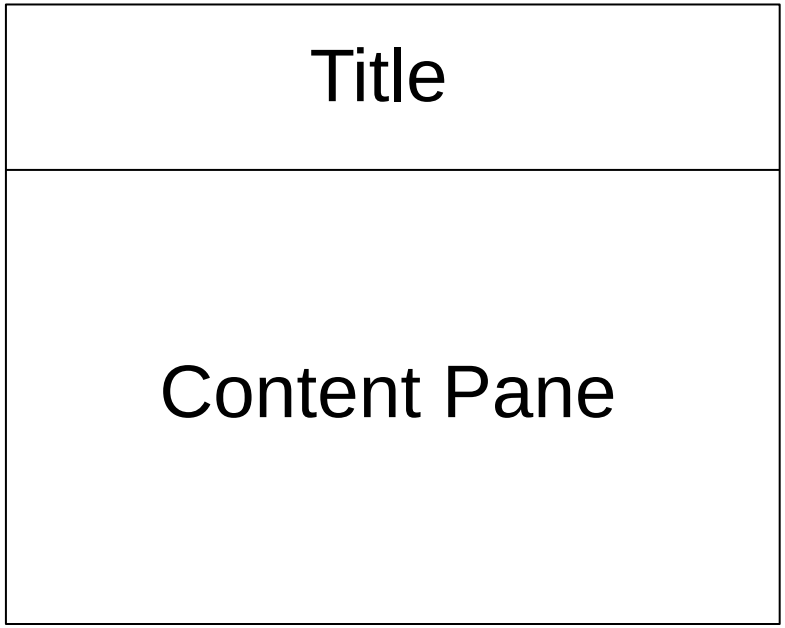
Components

- Label
- Button
- CheckBox
- ComboBox
- TextField
- RadioButton

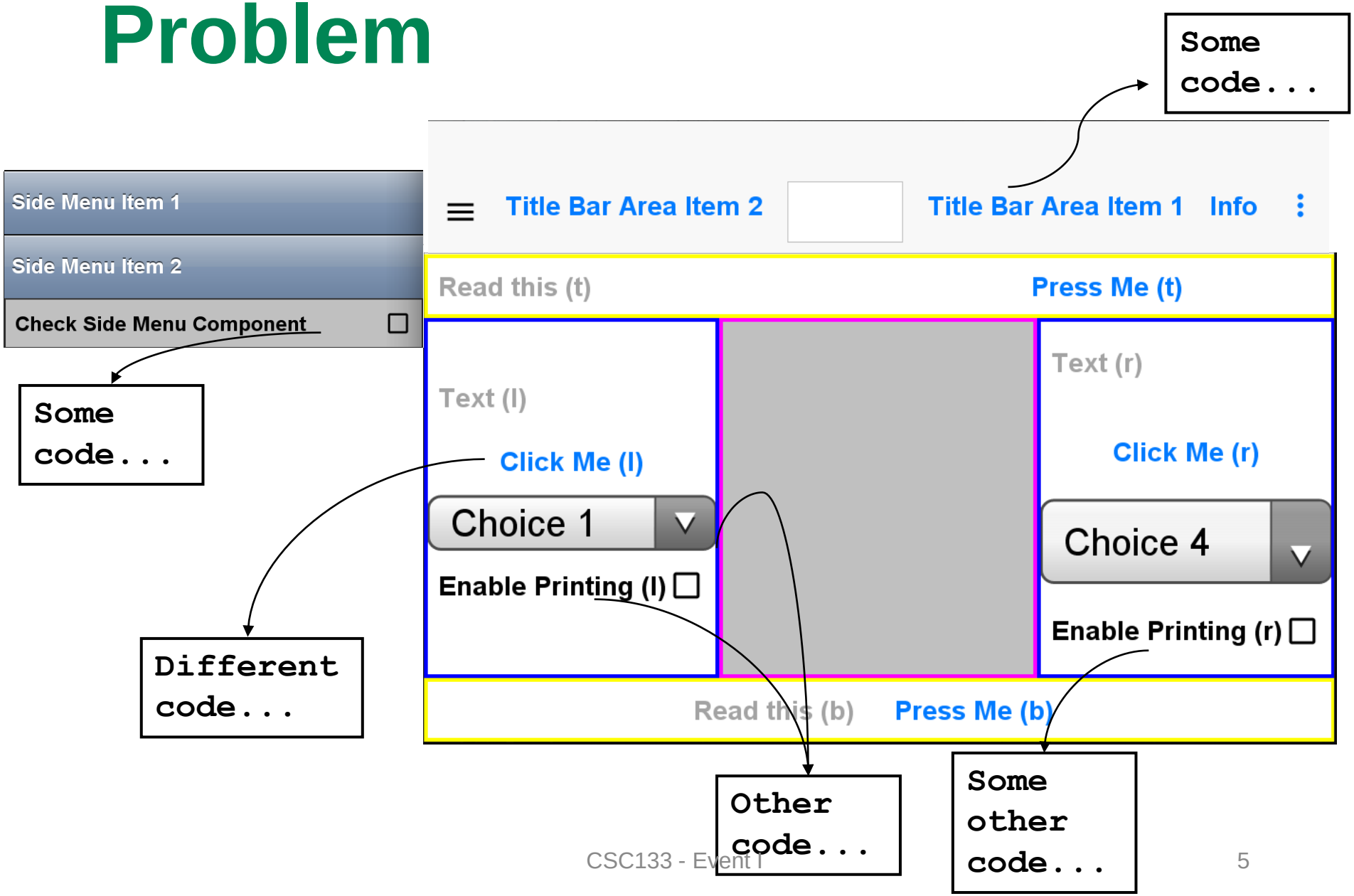


Form

- The top-level container
  - The main display



Problem


Some code...
Side Menu Item 1
Side Menu Item 2
Check Side Menu Component
Some code...
Title Bar Area Item 2
Title Bar Area Item 1
Info
Read this (t)
Press Me (t)
Text (l)
Click Me (l)
Choice 1
Enable Printing (l)
Text (r)
Click Me (r)
Choice 4
Enable Printing (r)
Different code...
Read this (b)
Press Me (b)
Other code...
Some other code...

How to make them work?

Traditional Way

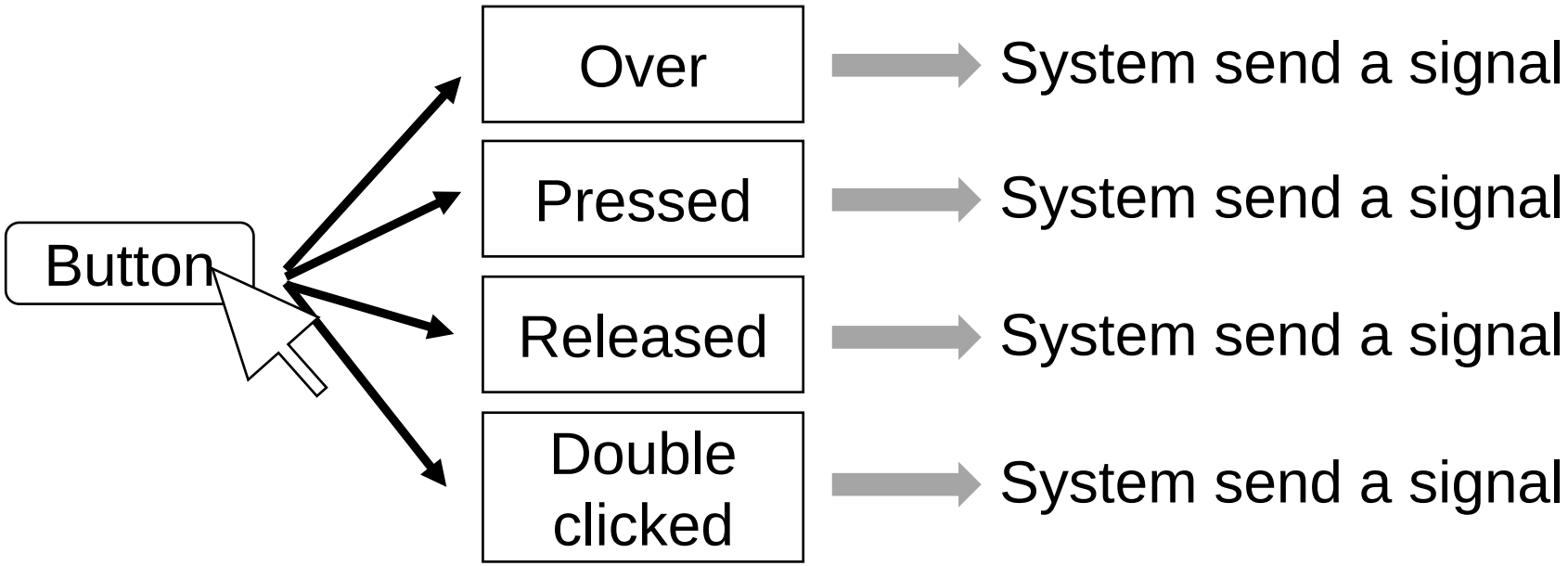
- Traditional program organization:

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

- In fact, someone did that already
  - Why not reuse it?

Event

- When you did sometime on the UI, event happen.



Event Object Types in Java

- Something happened
  - e.g., pushing a button,
  - **`ActionEvent`**
- User used the mouse
  - e.g., left clicked, right clicked, dragging
  - **`MouseEvent`**
- User used the keyboard
  - e.g., pressed, released
  - **`KeyEvent`**

Codename one

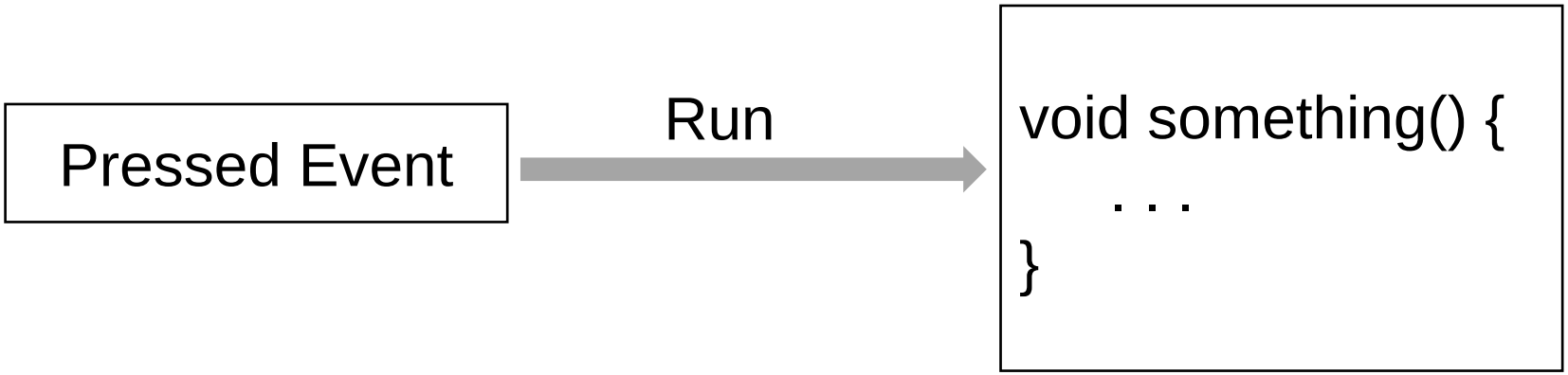
- CN1 does not have different type of event objects

- ALL produce an object of type **`ActionEvent`**
  - using a key: pressing, releasing
  - using of touch: pressing, releasing, dragging

- Note: but have different **`add`** method

Event-Driven

- You provide methods/codes
  - Callback function
- Run when the event happened



Traditional vs. Event-Driven

- Traditional program organization:

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

- Event-driven program organization:

```
add controls to ui {
    process input;
    process output;
}
```

Event Loop

- In fact, it is still a Loop inside event

```
loop {
   read event signal buffer;
   if (read)
      call your controls ();
}
until (done);
```

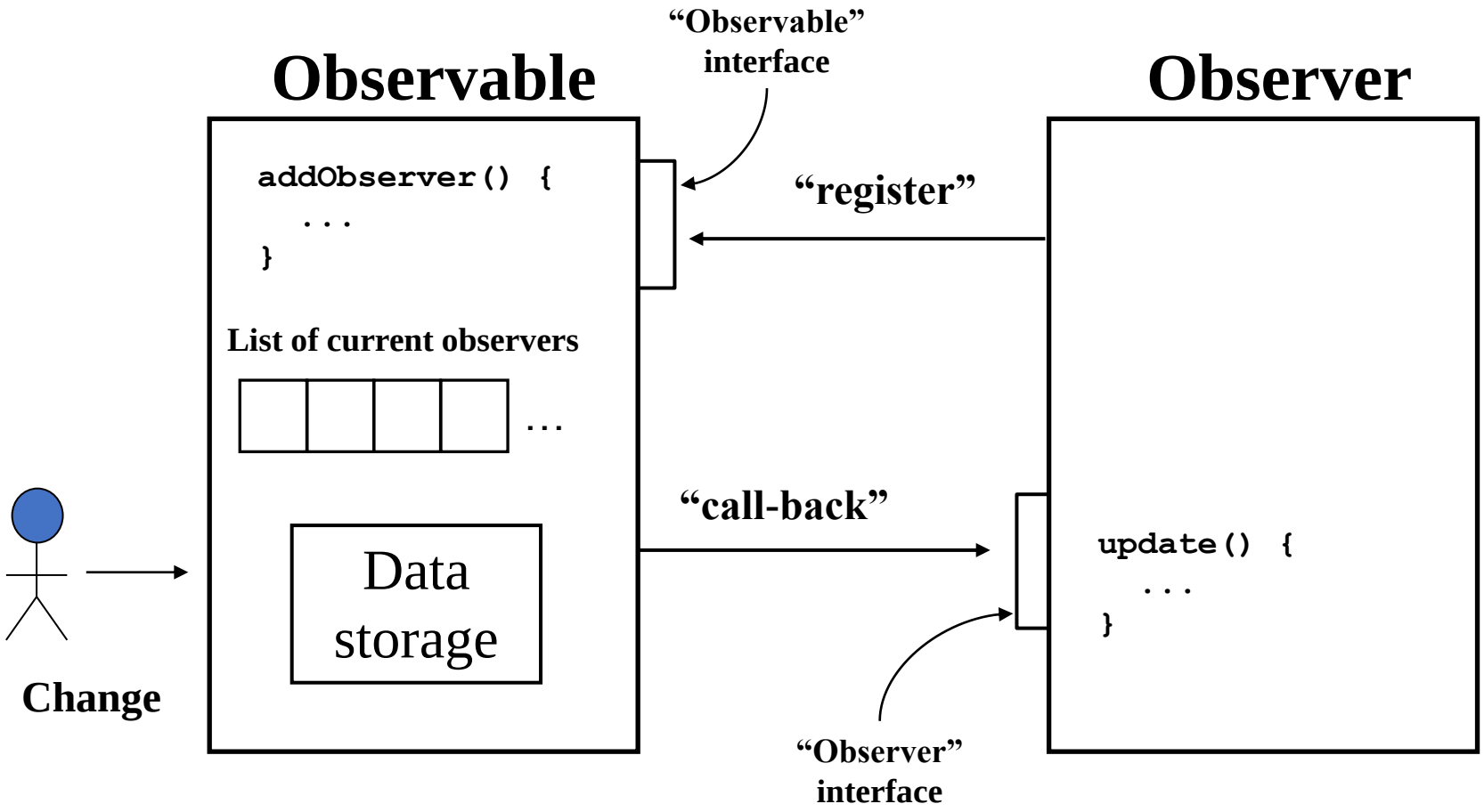
But someone did that for you

Event-Driven

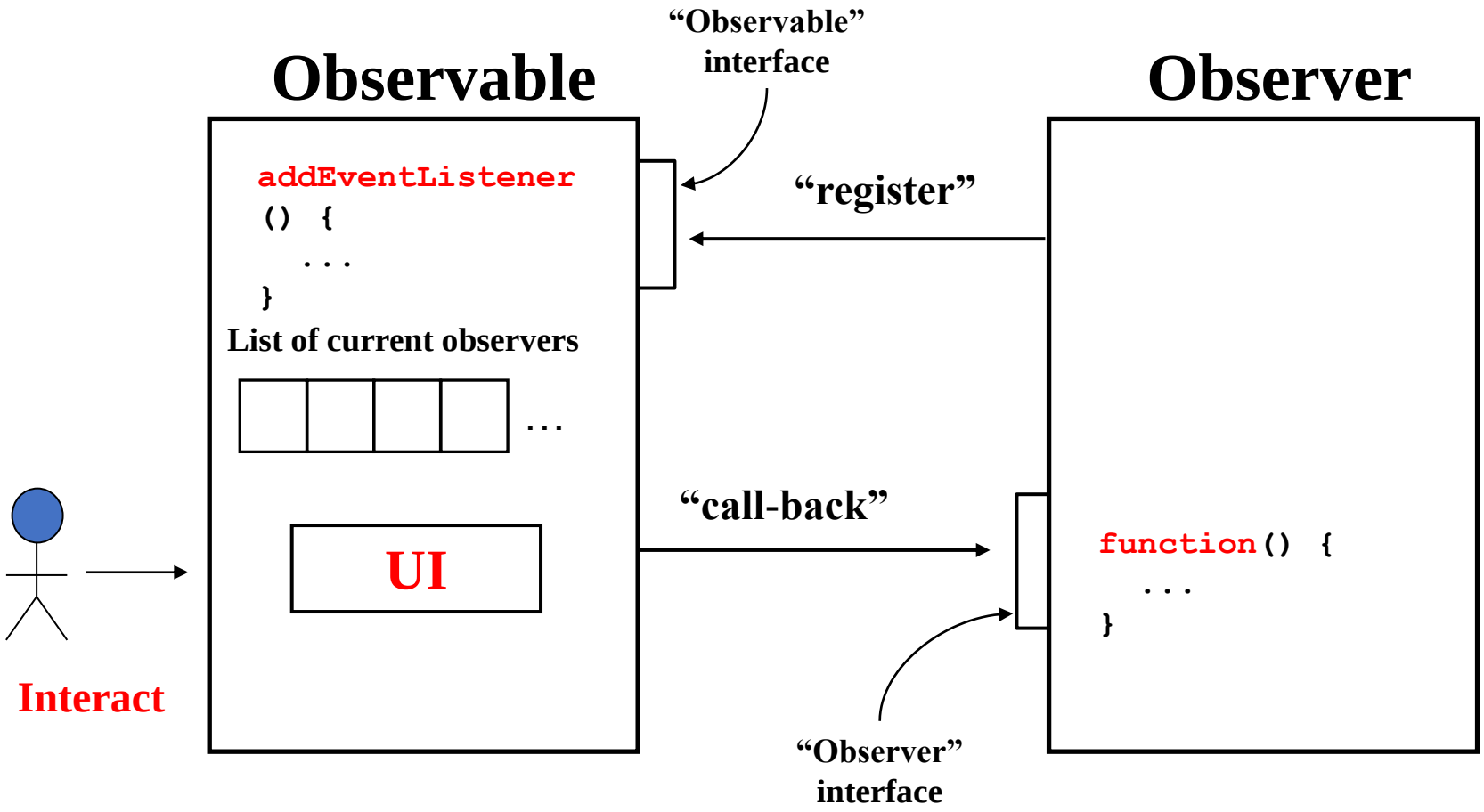
- To run a code when event happen

- Keep tracking the event
  - Observer Design Pattern!

Observer Design Pattern

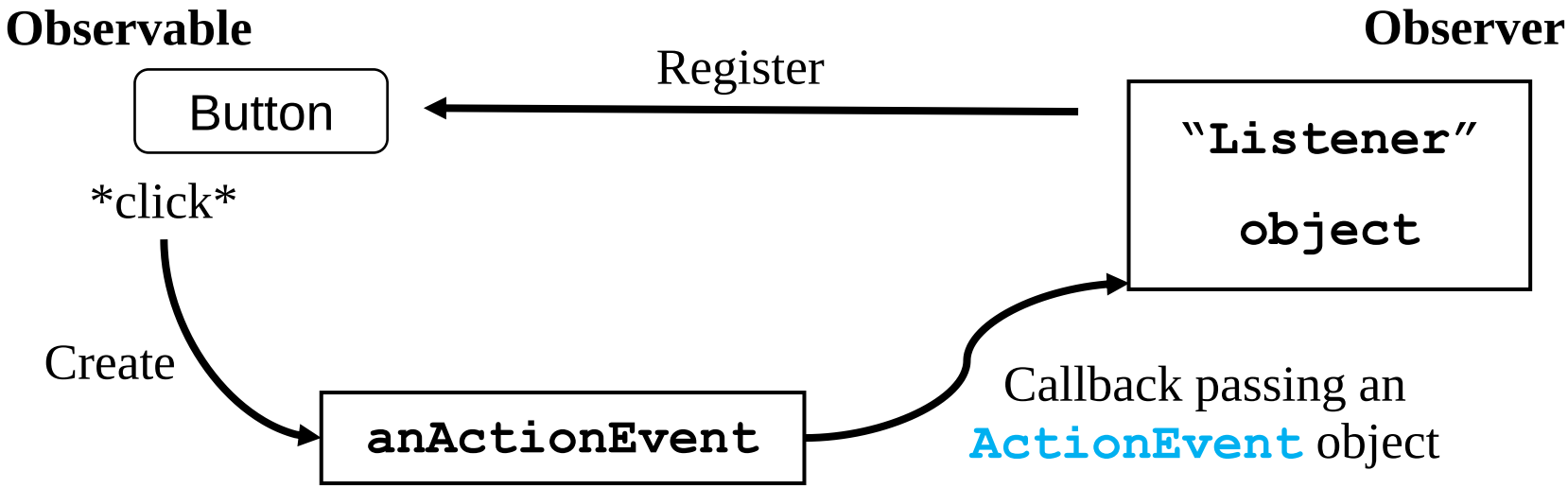


Same Idea


"Observable" interface
Observable
Observer
addEventListener () { . . . }
List of current observers
. . .
"register"
"call-back"
UI
function() { . . . }
Interact
"Observer" interface

Event Listeners

- Event-driven code attaches **listeners** to components
- Event callbacks to listeners



What to do?

- Implement the event listener
  - Handle the code
  - How?

- Add the listener to components
  - `addActionListener()`

ActionListener Interface

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

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

Creating a ActionListener

Two approaches:

1. Have a class that implements `ActionListener`

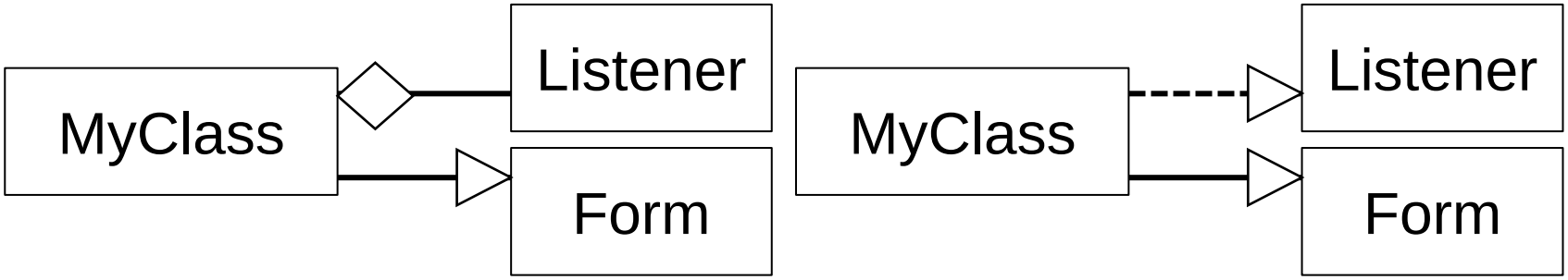
2. Have a class that extends built-in `Command` class

Approaches 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



Approach (1a)

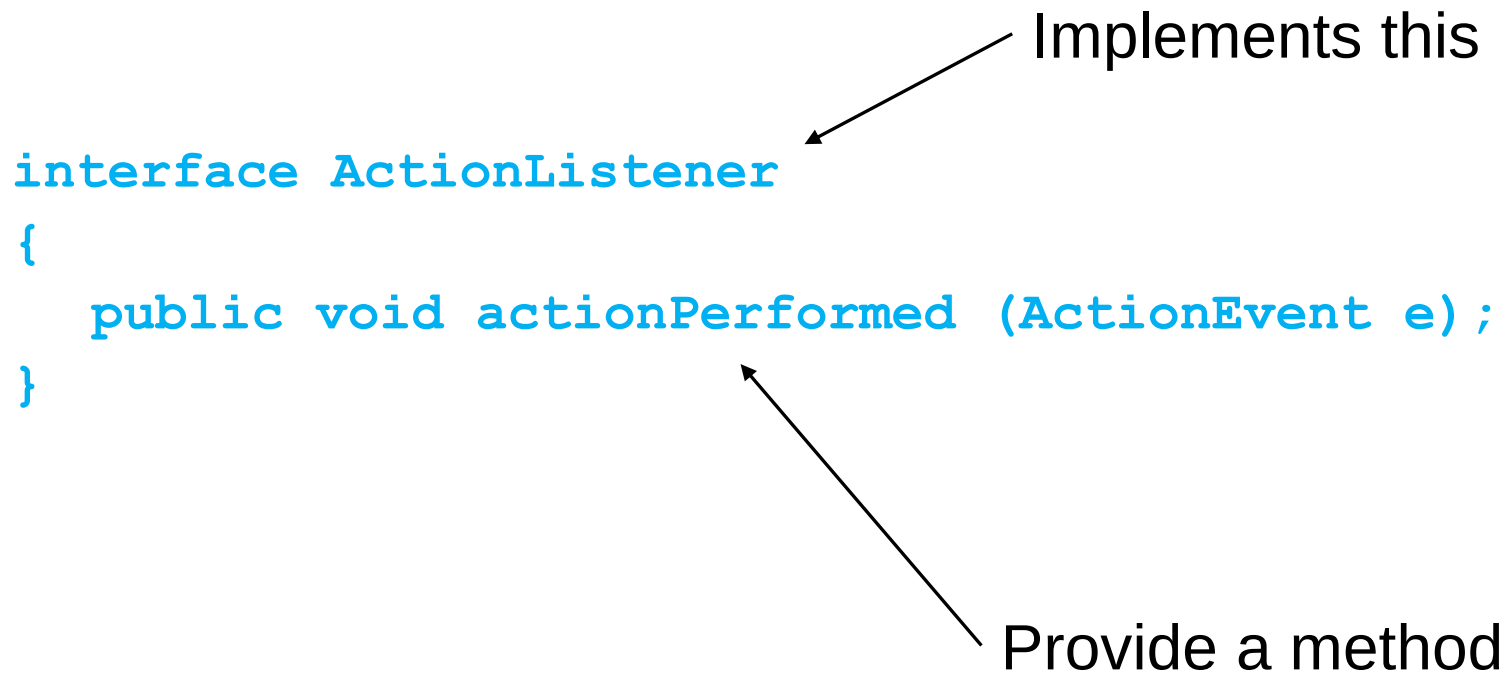
Your listener is different than the class that creates the components

You need:

- Implement a new class which
- `Implements ActionListener`

Remember?

- interface **`ActionListener`**



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

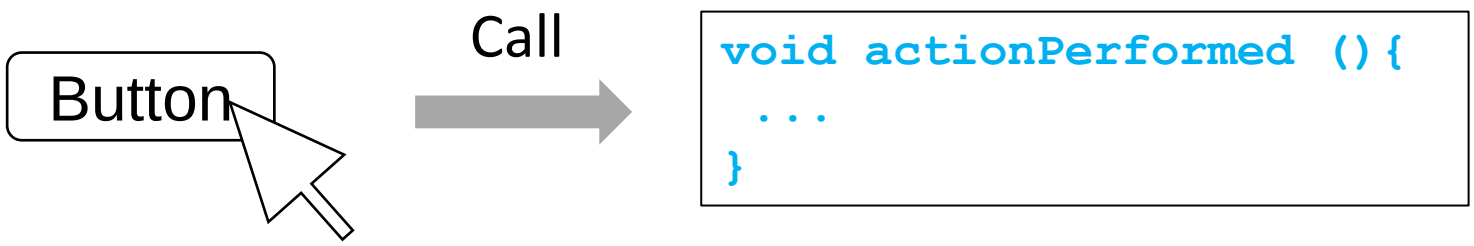
Inside a class that **extends from Form**:

```
/** 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");
//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);
```

Done

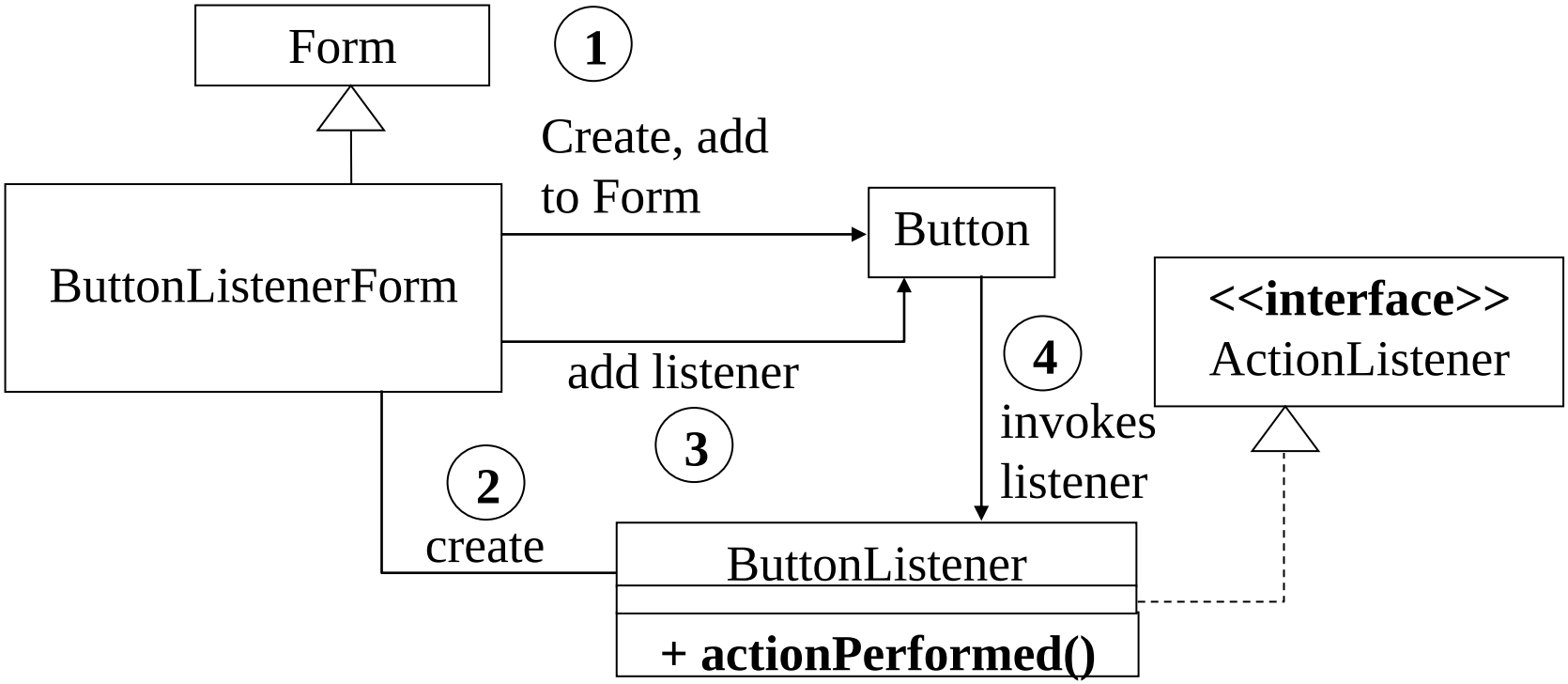
- After clicking the button

- `actionPerformed` in `ButtonListener` will be called



Listener Class Organization

UML for the previous code:



Remember in Asg1?

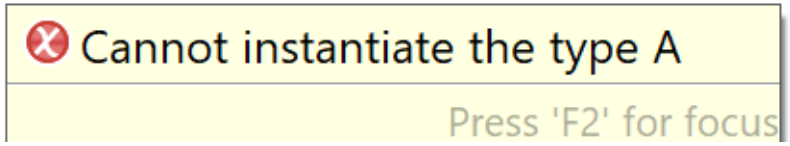
```
myTextField.addActionListener(
   new ActionListener(){
     public void actionPerformed
                          (ActionEvent evt) {
       String sCommand =
          myTextField.getText().toString();
       myTextField.clear();
       switch (sCommand.charAt(0)) {
         case 'x': gw.exit(); break;
       }
     }
   }
);
```

New interface?

```
interface A{
    void go();
}
```

Interface cannot be “new”

```
A a = new A();
```



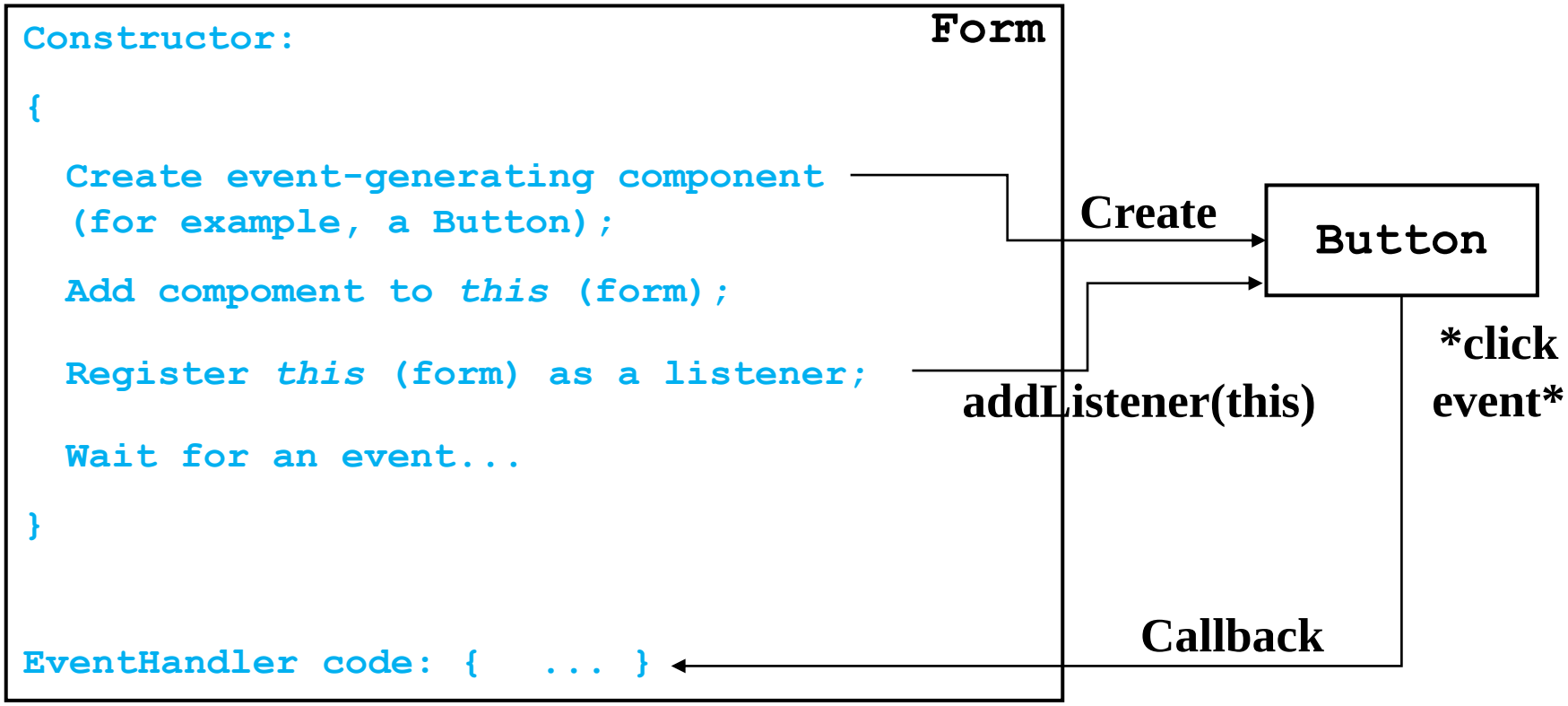
Shortcut

- A temporary class implements the interface
  - With auto-generated class name
  - “new” directly

```
A a = new A(){
    public void go(){
        System.out.print(“A”);
    }
};
```

Approach (1b)

Forms can listen to their own components!



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)...");
  }
}
```

Touch Input

```
public class MyForm extends Form implements ActionListener {
  public MyForm() {
      addPointerPressedListener(this);
      show();
  }
  public void actionPerformed(ActionEvent evt) {
      System.out.println(evt.getEventType());
  }
}
```

Three Types of Touch Event

- `addPointerDraggedListener(this);`

- `addPointerPressedListener(this);`

- `addPointerReleasedListener(this);`

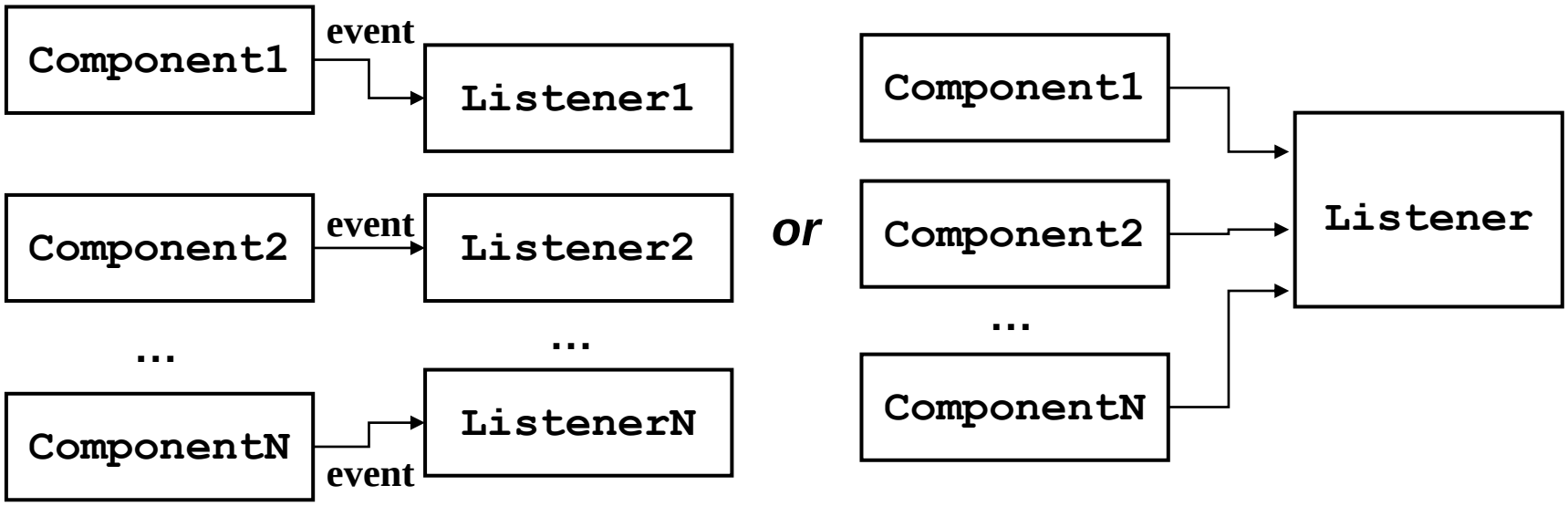
Keyboard Input

- Use addKeyListener in Form
  - With a keycode
- Event happens when clicking that key

- Code:
  - `this.addKeyListener('a',this);`

Multiple Event Sources

- Approaches:
  - (1a) requires multiple separate listeners
  - (1b) requires one listener

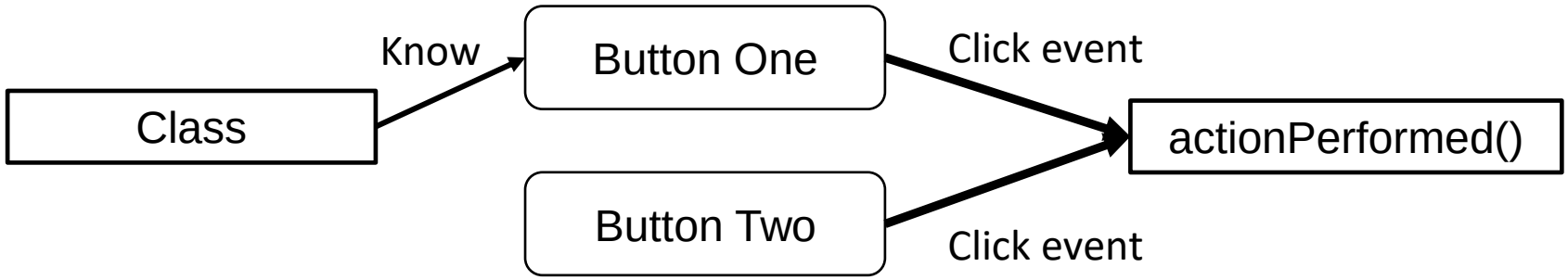


Single Listener

- Multiple events to one listener
- No need to create many class
- Need to be able to distinguish event source

Consider this

- Built 2 buttons
  - Button one in instance variable
  - Button two in local variable
- Clicking them will call `actionPerformed`



Code of Constructor

```
/* 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");
     add(buttonOne).add(buttonTwo);
     buttonOne.addActionListener(this);
     buttonTwo.addActionListener(this);
     show();
   }
   public void actionPerformed(ActionEvent evt) {
     //...See next page
   }
}
```

Code of method

```
public void actionPerformed(ActionEvent evt) {
    if(evt.getComponent().equals(buttonOne)){
      //buttonOne must be a class field
       ...
    }
    else if(((Button)evt.getComponent()).getText().equals
            ("Button Two")){
      //this does not work if label is changed
       ...
    }
  }
```

Multiple Component Listener

- `actionPerformed()` get bigger and bigger when more components

- Better approach is
  - Combination of (1a) and (1b): The approach (2)
  - Command Design Pattern!

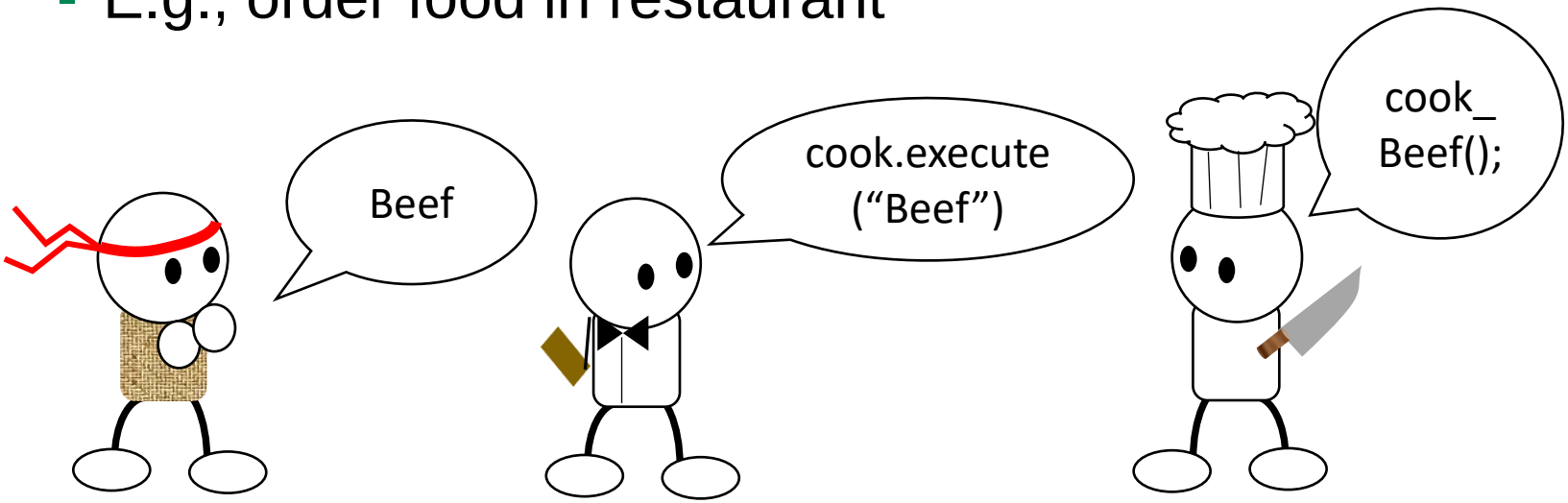
Approach (2)

- Use single listener
  - for all related components

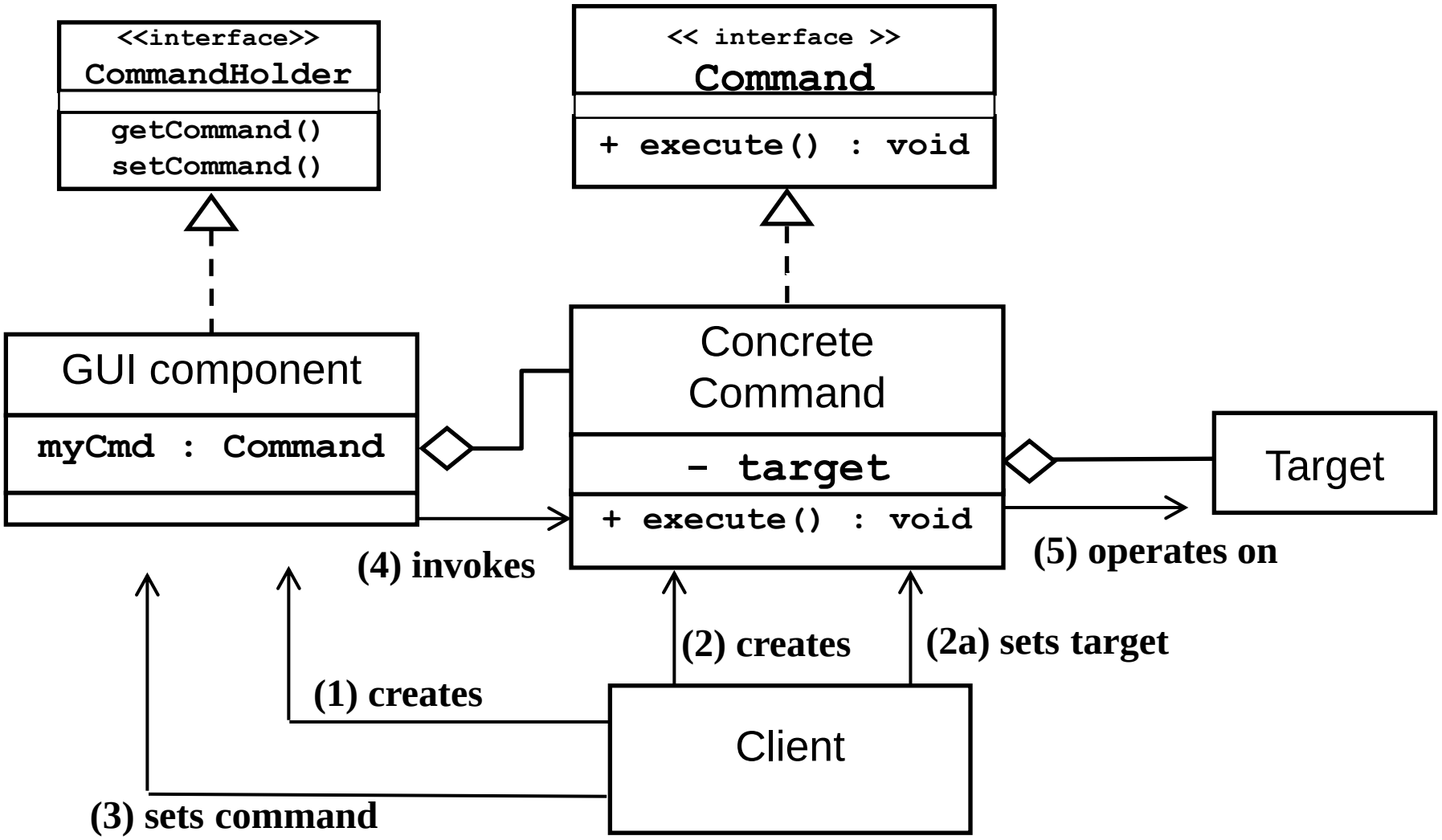
- But multiple listeners
  - for different groups of components

Command Pattern

- Behavioral
- Set up a list of command for `execute()`, only receiver know how to do it.
  - E.g., order food in restaurant

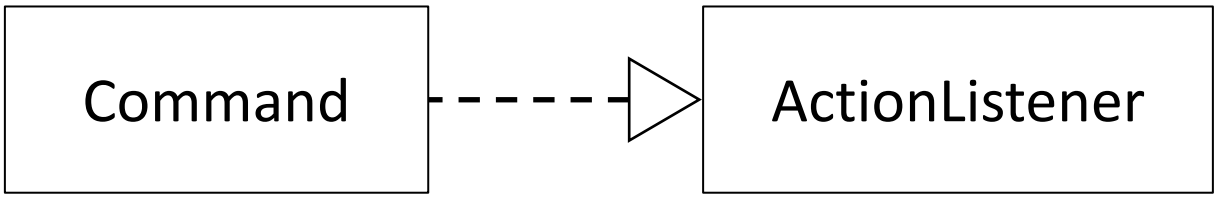


Command Pattern Organization



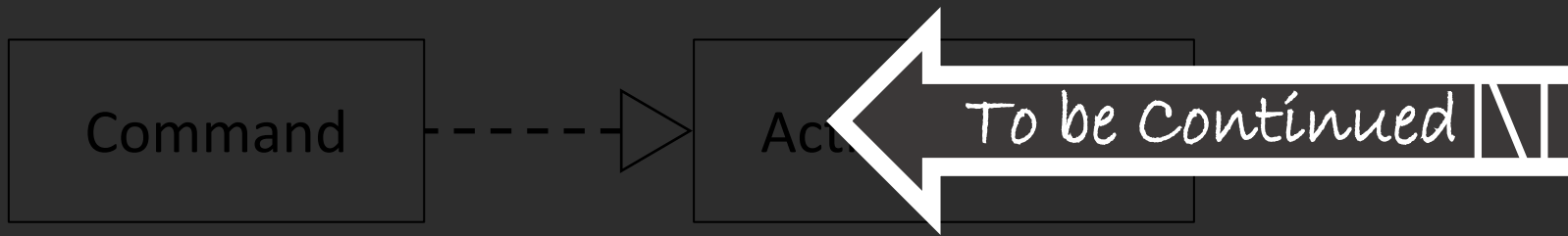
CN1 `Command` Class

- Build-in class that implements `ActionListener` interface.
  - Provides empty body implementation for: `actionPerformed() == "execute()"`



CN1 `Command` Class

- Build-in class that implements `ActionListener` interface.
  - Provides empty body implementation for: `actionPerformed() == "execute()"`



Any Questions?