View in MVC

Dr. Youna Jung

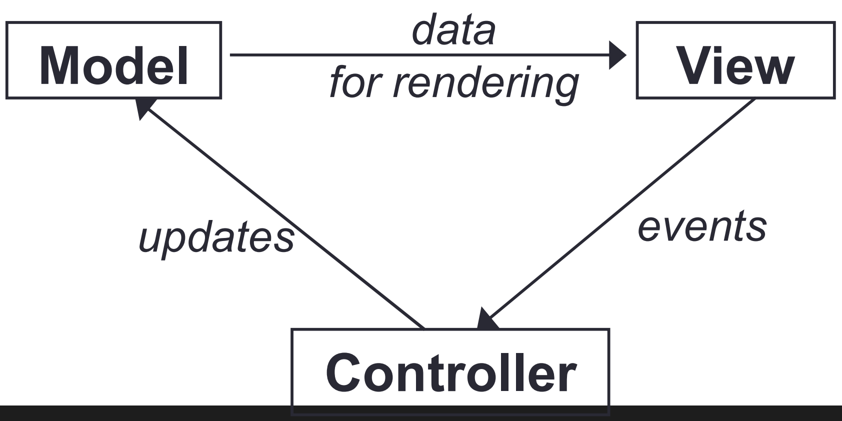
Northeastern University

yo.jung@northeastern.edu

MVC PATTERNS

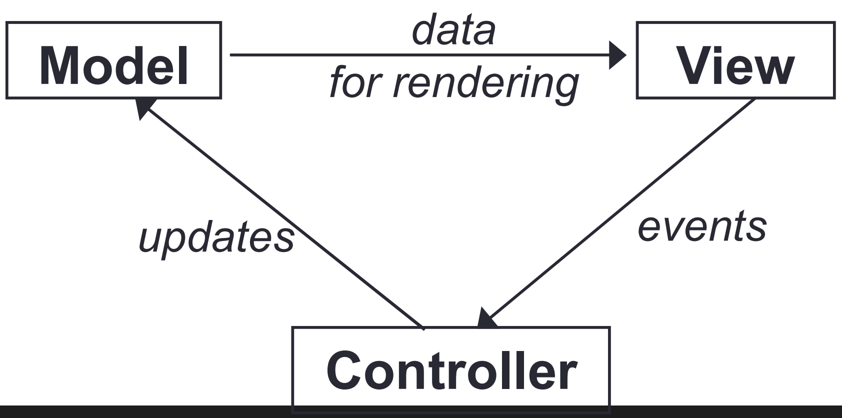
Model-View-Controller Pattern

- **Model**
  - ✓ **Classes** in your system that are related to the internal representation of **data** and **state** of the **system**
    - often part of the model is connected to **file**(s) or **database**(s)
    - Ex) **Card** game - **Card**, **Deck**, **Player**
    - EX) **Bank** system - **Account**, **User**, **UserList**
  - ✓ What it does
    - implements all the **functionality**
  - ✓ Does not do
    - does not care about **which functionality** is used **when**, **how results** are **shown** to the user



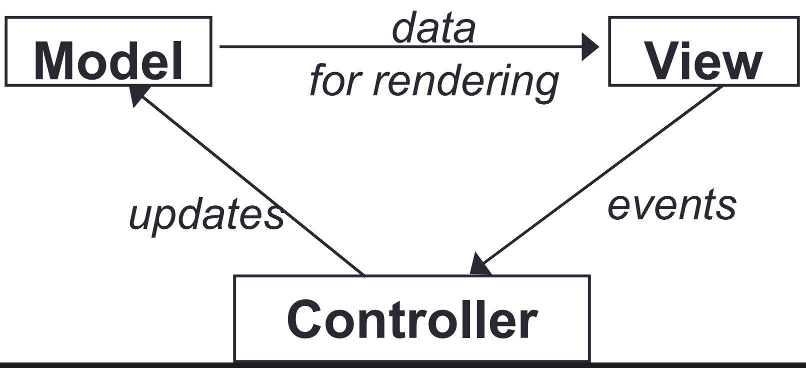
MVC Pattern

- **Controller**
  - ✓ Classes that connect **model** and **view**
    - defines how user interface reacts to user input (events)
    - receives messages from **view** (where events come from)
    - sends messages to **model** (tells what data to display)
  - ✓ What it does
    - **Takes** user inputs, tells **model** what to **do** and **view** what to **display**
  - ✓ Does not do
    - does **not care how model implements** functionality, screen layout to **display results**



MVC Pattern

- **View**
  - ✓ **Classes** in your system that **display** the state of the model to the user
    - generally, this is your **GUI** (could also be a **text UI**)
    - should not contain crucial application data
    - **Different views** can represent the same data in different ways
      - ➢Ex) Bar chart vs. pie chart
  - ✓ What it does
    - **display** results to user
  - ✓ Does not do
    - does not care **how** the **results** were **produced**, **when** to **respond** to user action



VIEW

MVC Pattern

VIEW

- The primary **responsibility** is to display information to the user
  - ✓ ← **View** gets the **data** from the **model** (directly or indirectly) but does not have the ability to directly ~~change the data~~ inside the model.
  - ✓ 2 Ways to display
    - **Console-based** output
    - **Graphical** outputs using a GUI
- **Design** of View
  - ✓ **(Traditionally) Controller** acts as the **client** for both the **model** and the **view** ➔ **View** should offer an **interface** that lets the **controller** call **operations** on the **view**.
    - **Determine what** the **controller** needs to be able to **tell** the **view to do**. This may include providing it with relevant **data** to **display**, telling the **view** to take specific **actions** at **specific times**, etc.

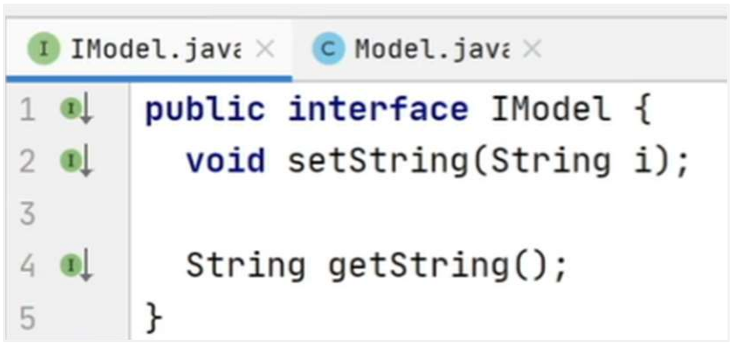
VIEW

- **Complication** on GUI-based View
  - ✓ **GUI** as being able to both
    - **display** information to the **user** **AND**
    - offer ways for the **user** to specify **input**
      - ➢ Taking inputs is the **Controller's responsibility**

⇩

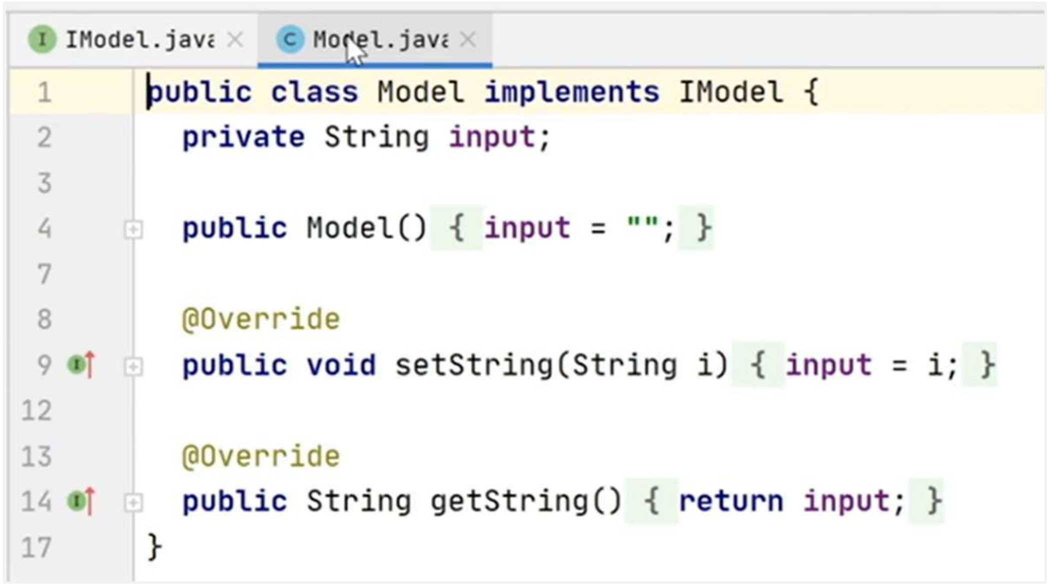
The **challenge** is to implement the **view** and **its communication** with the **controller** so that responsibilities are divided appropriately.

EX) Read a string and display: Model



IModel.java × Model.java ×

```java
public interface IModel {
  void setString(String i);

  String getString();
}
```

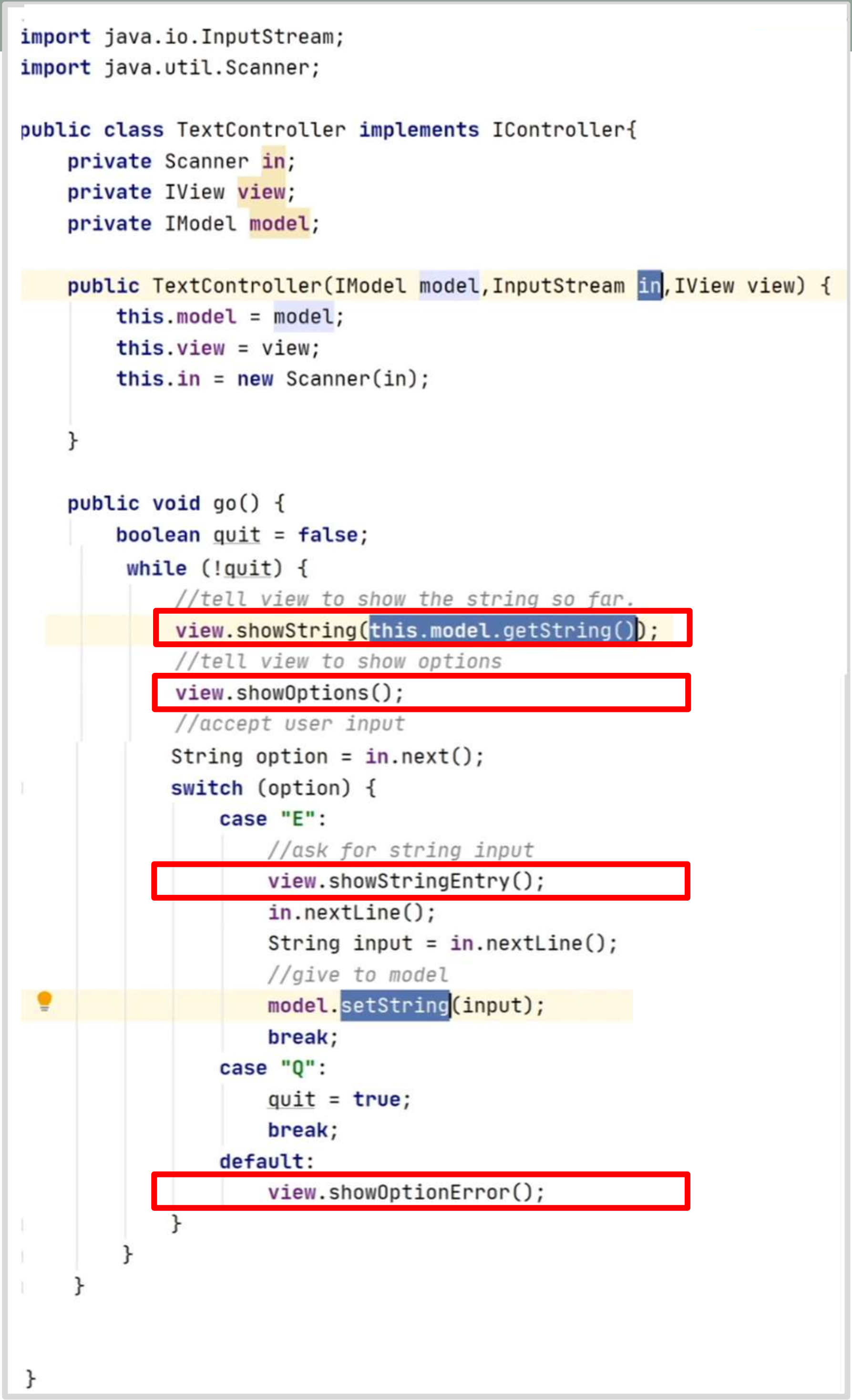


IModel.java × Model.java ×

```java
public class Model implements IModel {
  private String input;

  public Model() { input = ""; }

  @Override
  public void setString(String i) { input = i; }

  @Override
  public String getString() { return input; }
}
```

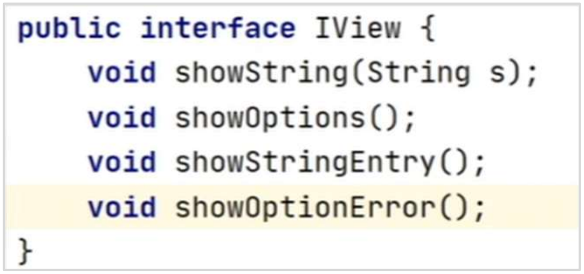
Controller

1) Tells the **View** to **show** the currently entered string
2) Tells the **View** to **show** the options to the user
3) Asks the **user** to enter an option
4) The user entered an option
    1) If a user select to quit, terminate the program
5) Tells the **View** to **show** a **message** to **input** a string
6) Takes the string as an input from the user
7) **Gives** the string to **Model**
8) Repeat

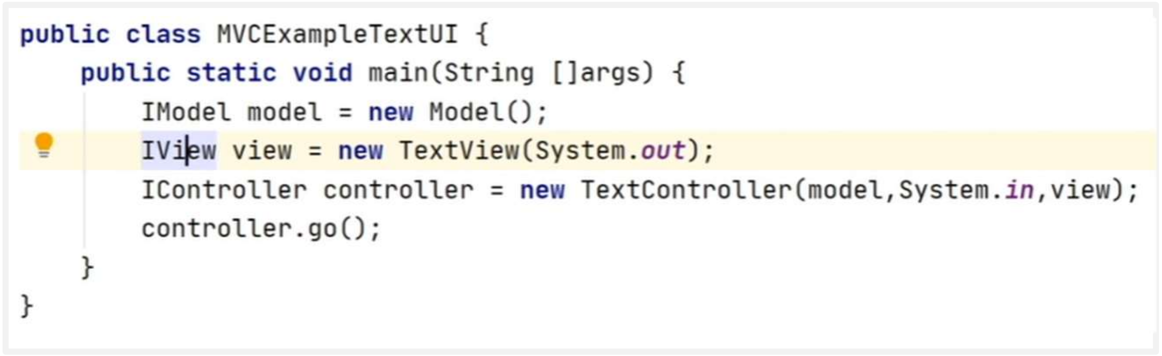


```
import java.io.InputStream;
import java.util.Scanner;

public class TextController implements IController{
    private Scanner in;
    private IView view;
    private IModel model;

    public TextController(IModel model,InputStream in,IView view) {
        this.model = model;
        this.view = view;
        this.in = new Scanner(in);

    }

    public void go() {
        boolean quit = false;
         while (!quit) {
            //tell view to show the string so far.
            view.showString(this.model.getString());
            //tell view to show options
            view.showOptions();
            //accept user input
            String option = in.next();
            switch (option) {
                case "E":
                    //ask for string input
                    view.showStringEntry();
                    in.nextLine();
                    String input = in.nextLine();
                    //give to model
                    model.setString(input);
                    break;
                case "Q":
                    quit = true;
                    break;
                default:
                    view.showOptionError();
            }
        }
    }

}
```

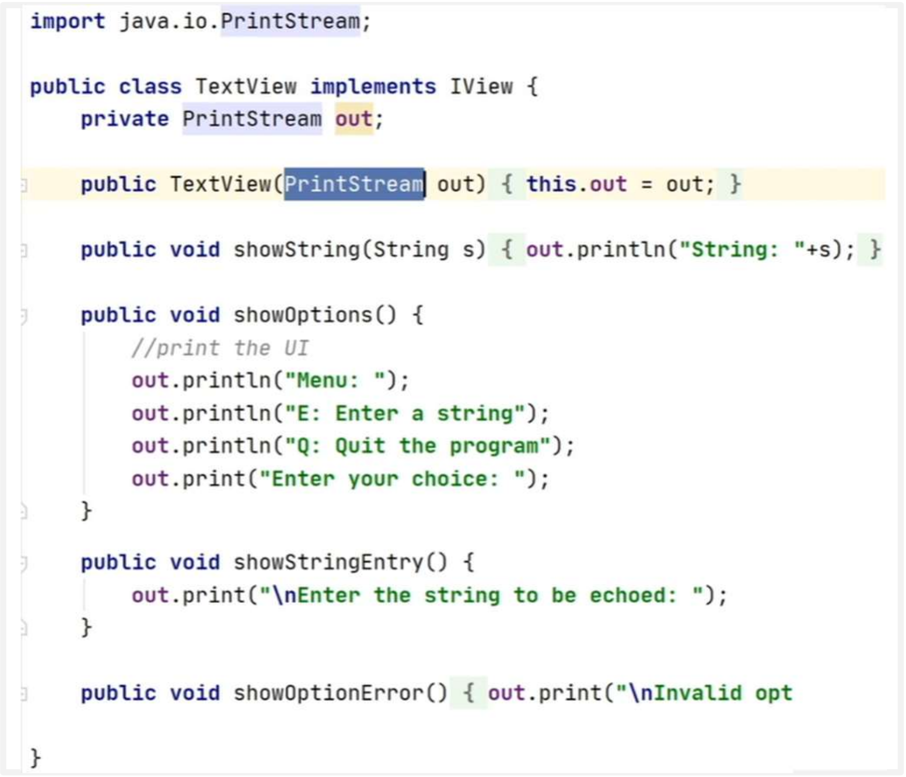
Text-based View



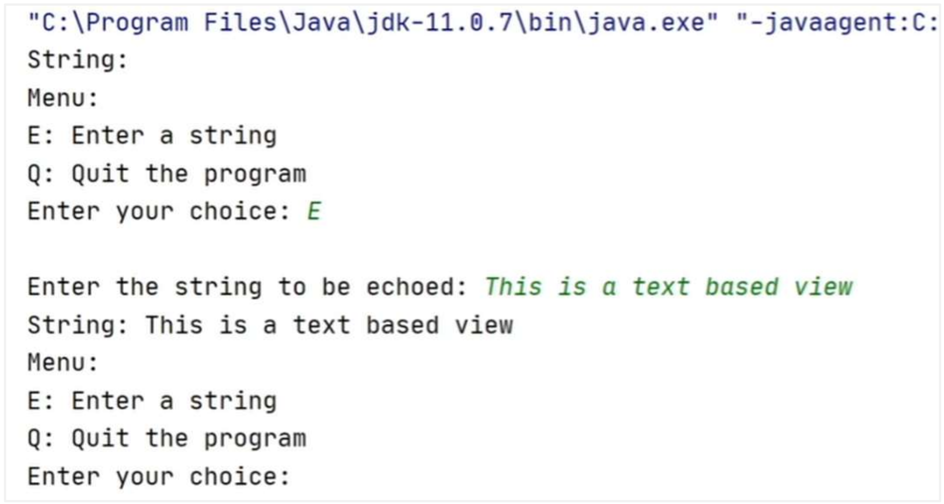
```
public interface IView {
    void showString(String s);
    void showOptions();
    void showStringEntry();
    void showOptionError();
}
```



```
public class MVCExampleTextUI {
    public static void main(String []args) {
        IModel model = new Model();
        IView view = new TextView(System.out);
        IController controller = new TextController(model,System.in,view);
        controller.go();
    }
}
```



```
import java.io.PrintStream;

public class TextView implements IView {
    private PrintStream out;

    public TextView(PrintStream out) { this.out = out; }

    public void showString(String s) { out.println("String: "+s); }

    public void showOptions() {
        //print the UI
        out.println("Menu: ");
        out.println("E: Enter a string");
        out.println("Q: Quit the program");
        out.print("Enter your choice: ");
    }

    public void showStringEntry() {
        out.print("\nEnter the string to be echoed: ");
    }

    public void showOptionError() { out.print("\nInvalid opt

}
```



```
"C:\Program Files\Java\jdk-11.0.7\bin\java.exe" "-javaagent:C:
String:
Menu:
E: Enter a string
Q: Quit the program
Enter your choice: E

Enter the string to be echoed: This is a text based view
String: This is a text based view
Menu:
E: Enter a string
Q: Quit the program
Enter your choice:
```

Event-driven programming

- How it works
  - ✓ When a program is started → setup procedures and then **waits**
  - ✓ When the user interacts with the program (**Event**)→ the program springs into action → **performs** certain **tasks**
  - ✓ Then goes back to **waiting**

- **Event**
  - ✓ Each **action** by the **user**
    - – E.g.) clicking a button, selecting a menu item, typing text, etc
  - ✓ could be generated without any user actions as well
    - – E.g.) the program **auto-saves a file** after a fixed time interval, the program **auto-checks email** periodically, etc.

Event-driven programming

- Design and Implementation
  1) **STEP 1** - **Identifying which events** the program should **react** to, and **what** that **reaction** should be
     - In GUIs, create the GUI layout and determine which user interaction the program should react to
  2) **STEP 2** - **Write** the **reactions** as code
     - Reactions can be written as **functions**
     - In GUIs, provide **callbacks** for the button clicks, menu item selections, etc.
       - **Callback** function – "*When the program is run, and you generate an event, call this function*"

- Programs with a GUI are all **event-driven** (Reactive)

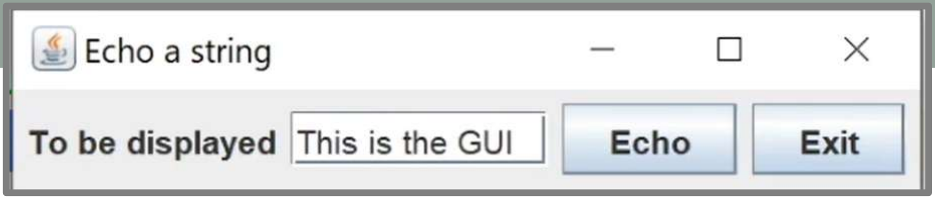
GUI Terms

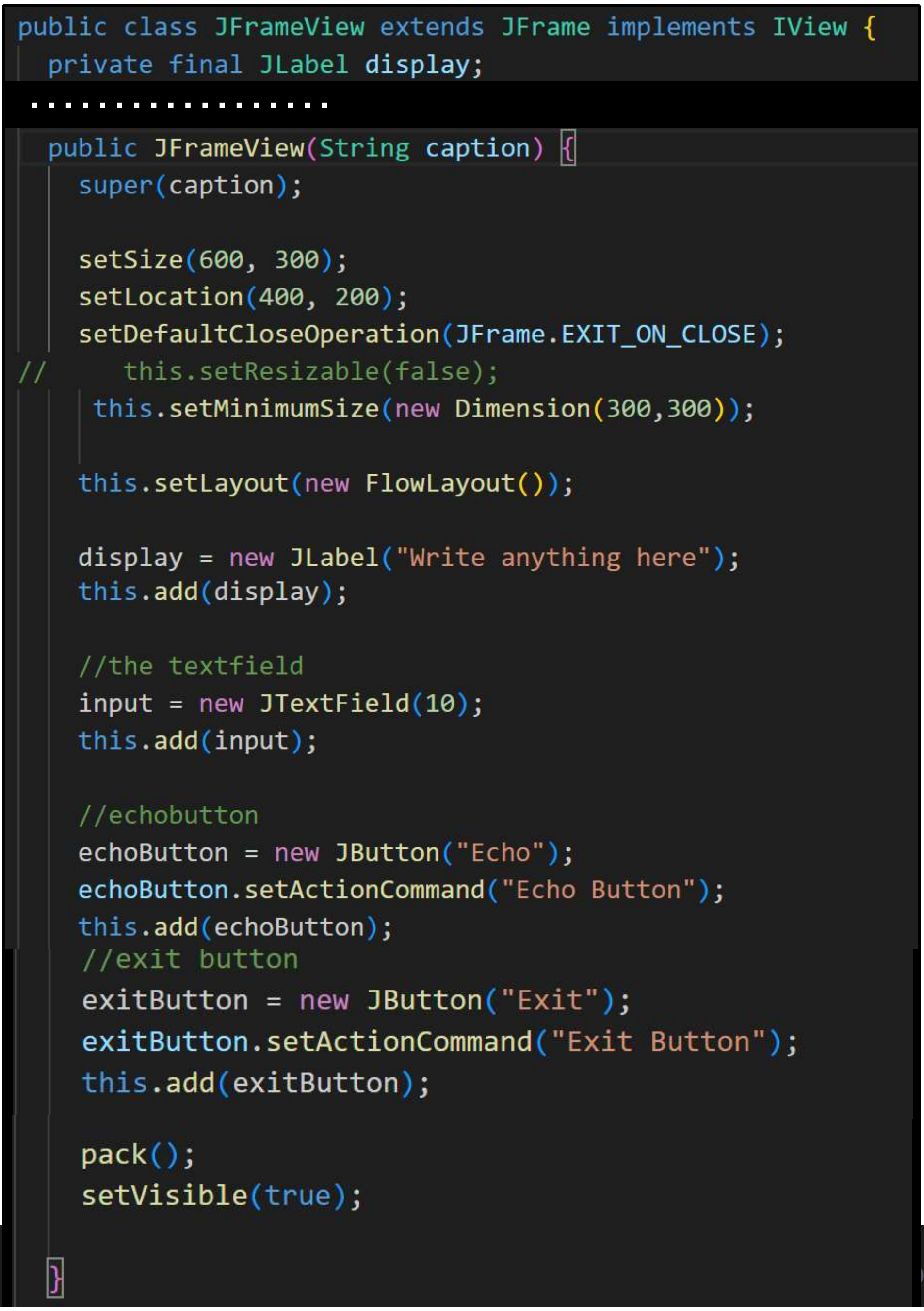
- **Frame**
  - ✓ A "window". It comes with a title bar along with the three buttons to minimize, maximize and close the window respectively.
- **Pane**
  - ✓ The area inside the borders of the frame. The pane usually contains all the components of the GUI.
- **Panel**
  - ✓ The pane may be divided into smaller regions, referred to as panels.
- **Container**
  - ✓ A general-purpose entity that is capable of containing other things within it
    - Ex) A button can contain text or an icon
    - EX) A menu contains menu items
- **Component**
  - ✓ General-purpose entity that provides some functionality (and usually fires events).
    - The same item (e.g. button) can function as both a container and a component).

STEP 1: GUI Design

- To **create** a **window** with a **frame** in Java Swing, we use the `JFrame` class.
  - ✓ to **customize** the frame → A **subclass** of `Jframe` → `JFrameView`
- `JFrameView()` **constructs** the **visual layout** of the **window**
    - – Always call the constructor of `JFrame` → `super()`
  - ✓ `setSize()` → `setSize(600,300)`
    - – **creates** a **frame** with a specific **size**
  - ✓ `setDefaultCloseOperation()`
    - – determines the **behavior** when the "**close-window**" **button** is clicked
    - – → `setDefaultCloseOperation(Jframe.EXIT_ON_CLOSE)`
  - ✓ `setLayout()`
    - – Determines the layout of the window → `setLayout(new FlowLayout())`
      - ➢ `FlowLayout()` arranges components in a directional flow
        - ✓ LEFT_TO_RIGHT or RIGHT_TO_LEFT
  - ✓ `setVisible()`
    - – sets the **window** to be **visible**.
  - ✓ `Pack()`
    - – packs the **window** to tightly enclose its **contents**

STEP 1



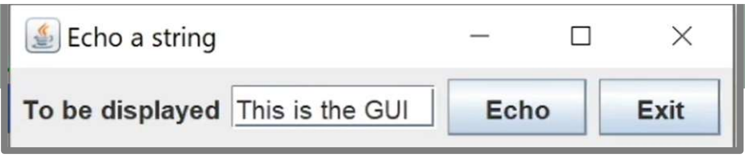


```
public class JFrameView extends JFrame implements IView {
  private final JLabel display;
 ..................
  public JFrameView(String caption) {
    super(caption);

    setSize(600, 300);
    setLocation(400, 200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//     this.setResizable(false);
     this.setMinimumSize(new Dimension(300,300));

    this.setLayout(new FlowLayout());

    display = new JLabel("Write anything here");
    this.add(display);

    //the textfield
    input = new JTextField(10);
    this.add(input);

    //echobutton
    echoButton = new JButton("Echo");
    echoButton.setActionCommand("Echo Button");
    this.add(echoButton);
    //exit button
    exitButton = new JButton("Exit");
    exitButton.setActionCommand("Exit Button");
    this.add(exitButton);

    pack();
    setVisible(true);

  }
```

STEP 2: Implementation of Reactions

- To represent a **text-label**, **text field**, and **button** for input respectively
  - ✓ Use the `JLabel` , `JTextField` and `JButton` classes

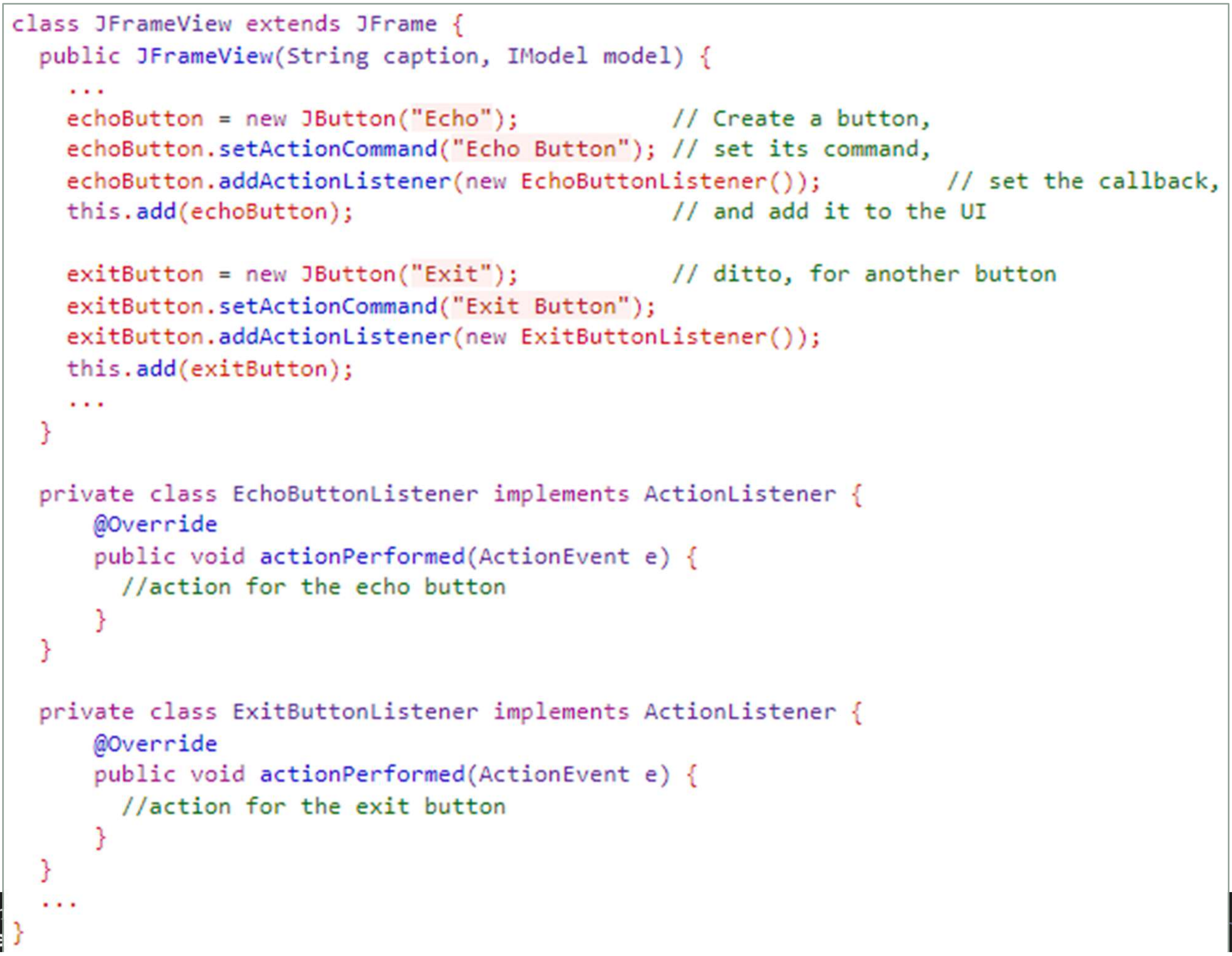
- `JButton` Class
  - ✓ Each button a unique name (called an ***action command***)
  - ✓ When a button is clicked
    - – `JButton` object generates an `ActionEvent`-type event
    - – The corresponding **callback** is in the form of an object that implements the `ActionListener` interface
    - – The actual callback function is implemented in `actionPerformed()`



1) View class with one `ActionListener`

```
class JFrameView extends JFrame implements ActionListener {
  public JFrameView(String caption, IModel model) {
    ...
    echoButton = new JButton("Echo");                  // Create a button,
    echoButton.setActionCommand("Echo Button"); // set its command,
    echoButton.addActionListener(this);             // set the callback,
    this.add(echoButton);                               // and add it to the UI

    exitButton = new JButton("Exit");                  // ditto, for another button
    exitButton.setActionCommand("Exit Button");
    exitButton.addActionListener(this);
    this.add(exitButton);
    ...
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    switch (e.getActionCommand()) {
    case "Echo Button": ...
    case "Exit Button": ...
    }
  }

  ...
}
```

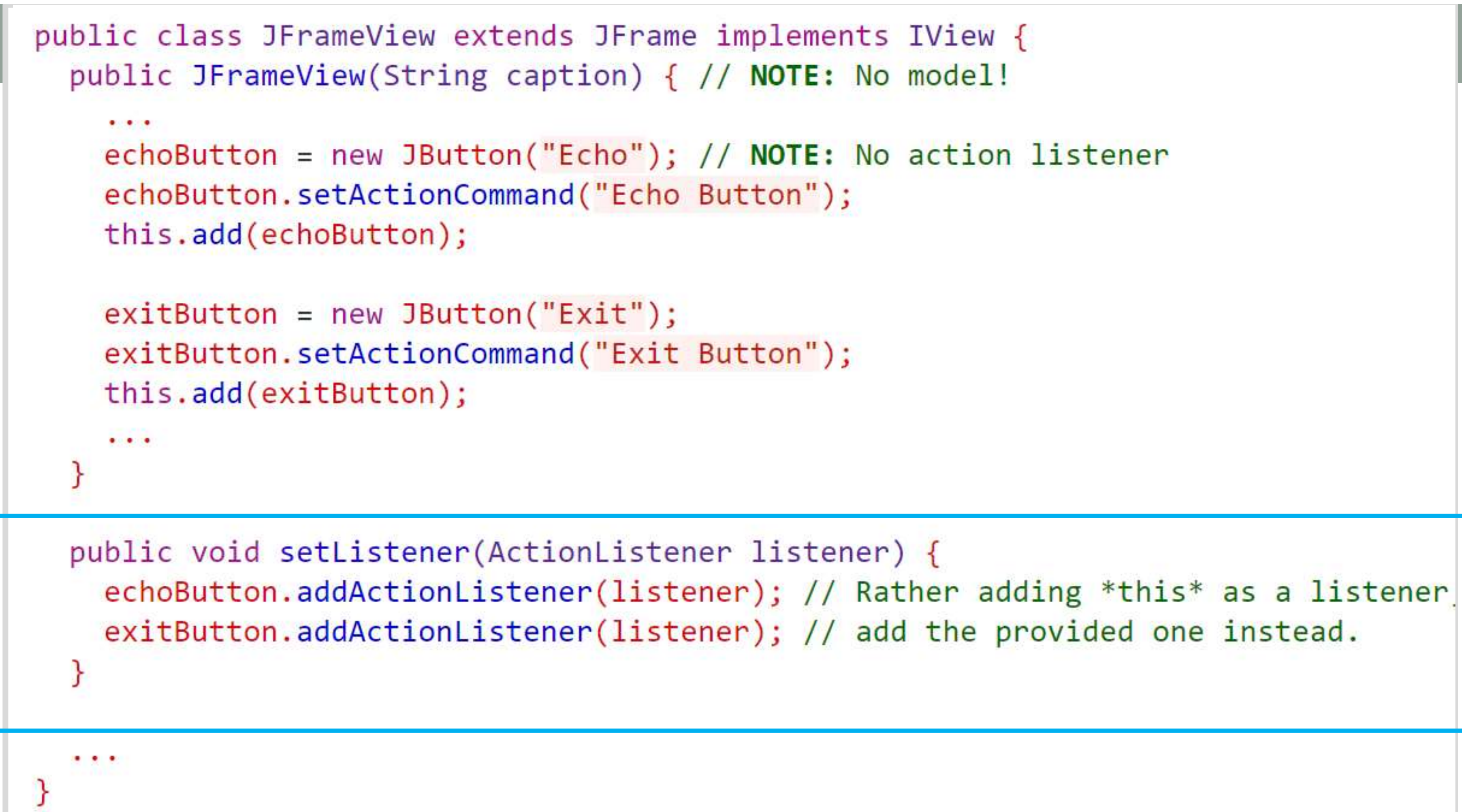
2) View class with two `ActionListener` objects



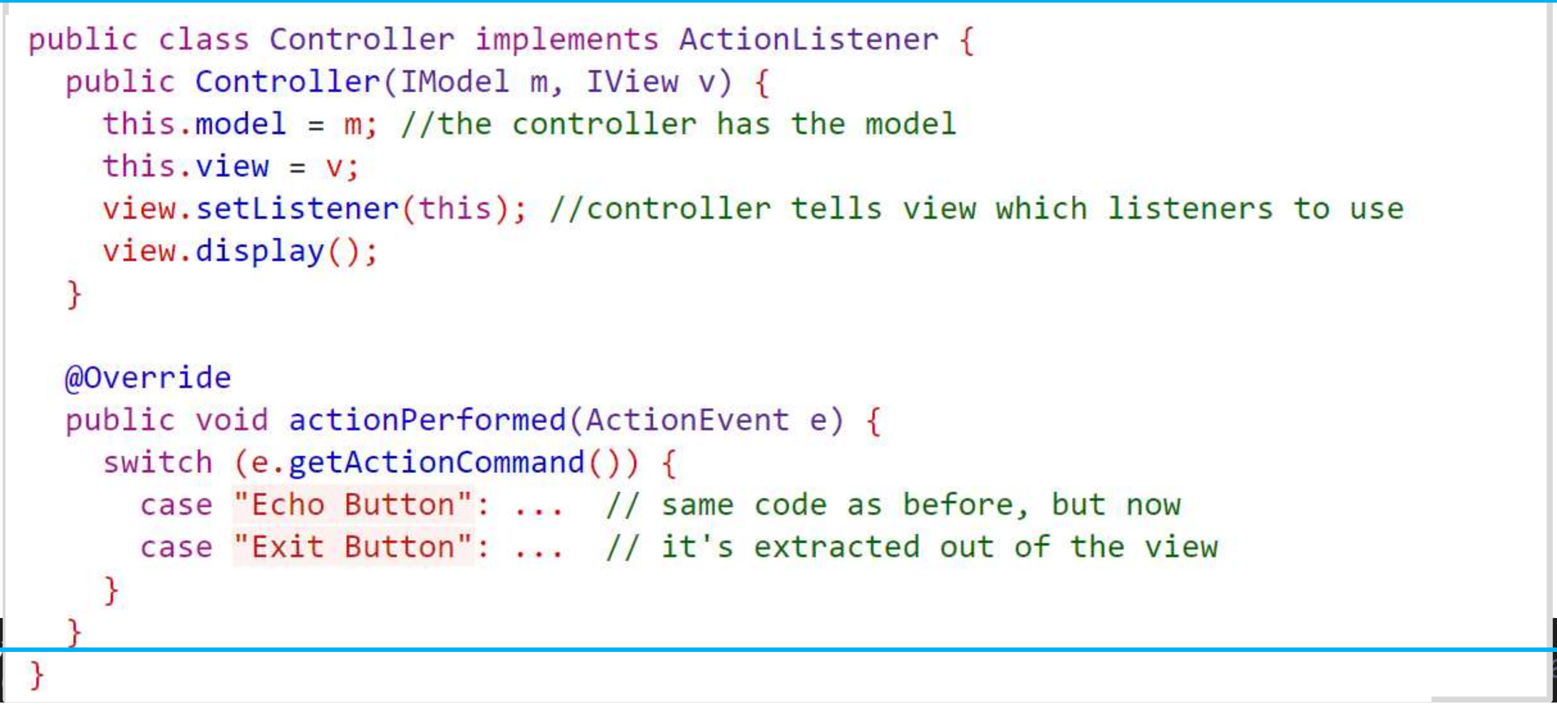
```
class JFrameView extends JFrame {
  public JFrameView(String caption, IModel model) {
    ...
    echoButton = new JButton("Echo");              // Create a button,
    echoButton.setActionCommand("Echo Button"); // set its command,
    echoButton.addActionListener(new EchoButtonListener());          // set the callback,
    this.add(echoButton);                          // and add it to the UI

    exitButton = new JButton("Exit");              // ditto, for another button
    exitButton.setActionCommand("Exit Button");
    exitButton.addActionListener(new ExitButtonListener());
    this.add(exitButton);
    ...
  }

  private class EchoButtonListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent e) {
        //action for the echo button
      }
  }

  private class ExitButtonListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent e) {
        //action for the exit button
      }
  }
  ...
}
```

MVC-Compliant Design

- refactors the application to follow **MVC** design
  - ✓ contains a controller that effectively acts as the **callback** for the buttons in the view.
  - ✓ `main()` **creates** the model, view and controller → **passes** M & V to C (V no longer has direct access to the model)
  - ✓ During initialization, **C passes** itself as the **listener** for all the **V's buttons**
    - When the **button** is **clicked** ➔ A **method inside** the **controller** is called ➔ **C gets control** over **what to do next**



```java
public class JFrameView extends JFrame implements IView {
  public JFrameView(String caption) { // NOTE: No model!
    ...
    echoButton = new JButton("Echo"); // NOTE: No action listener
    echoButton.setActionCommand("Echo Button");
    this.add(echoButton);

    exitButton = new JButton("Exit");
    exitButton.setActionCommand("Exit Button");
    this.add(exitButton);
    ...
  }

  public void setListener(ActionListener listener) {
    echoButton.addActionListener(listener); // Rather adding *this* as a listener
    exitButton.addActionListener(listener); // add the provided one instead.
  }

  ...
}
```



```java
public class Controller implements ActionListener {
  public Controller(IModel m, IView v) {
    this.model = m; //the controller has the model
    this.view = v;
    view.setListener(this); //controller tells view which listeners to use
    view.display();
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    switch (e.getActionCommand()) {
      case "Echo Button": ...  // same code as before, but now
      case "Exit Button": ...  // it's extracted out of the view
    }
  }
}
```

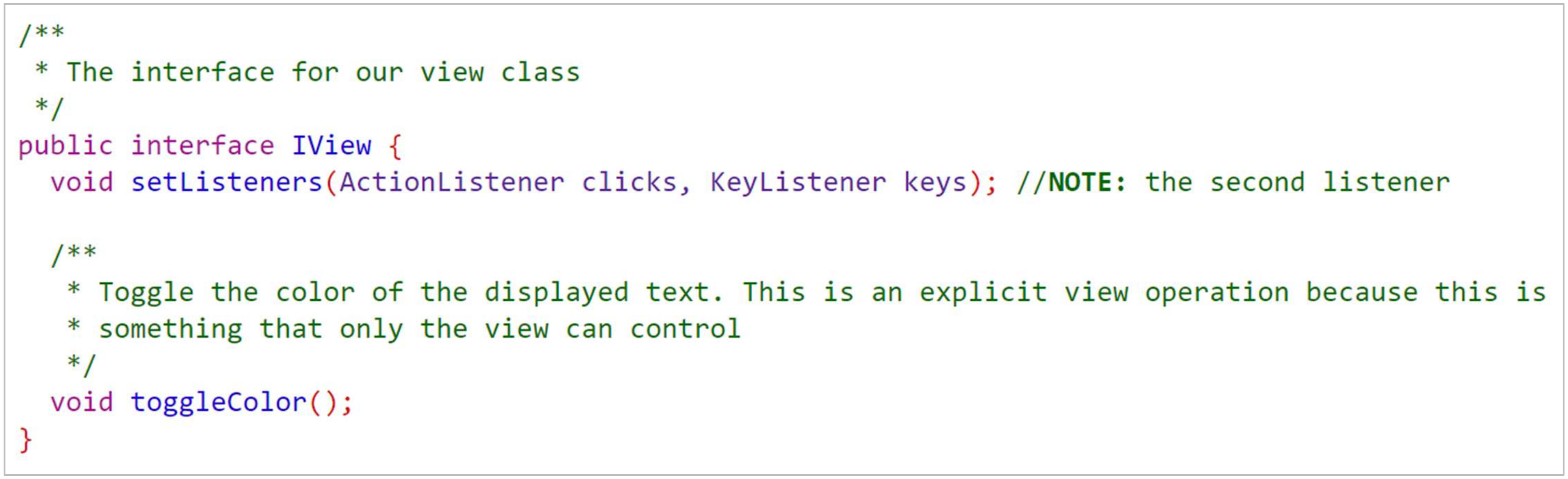
Adding **Keyboard Inputs**

- ❑ Let us enhance our application by adding the following features
  1) **Pressing** the **'D' key** toggles the color of the echoed text between **black** and red.
  2) **Holding down** the **'C' key** makes the echoed text upper-case, and **releasing** it reverts back to the original case.

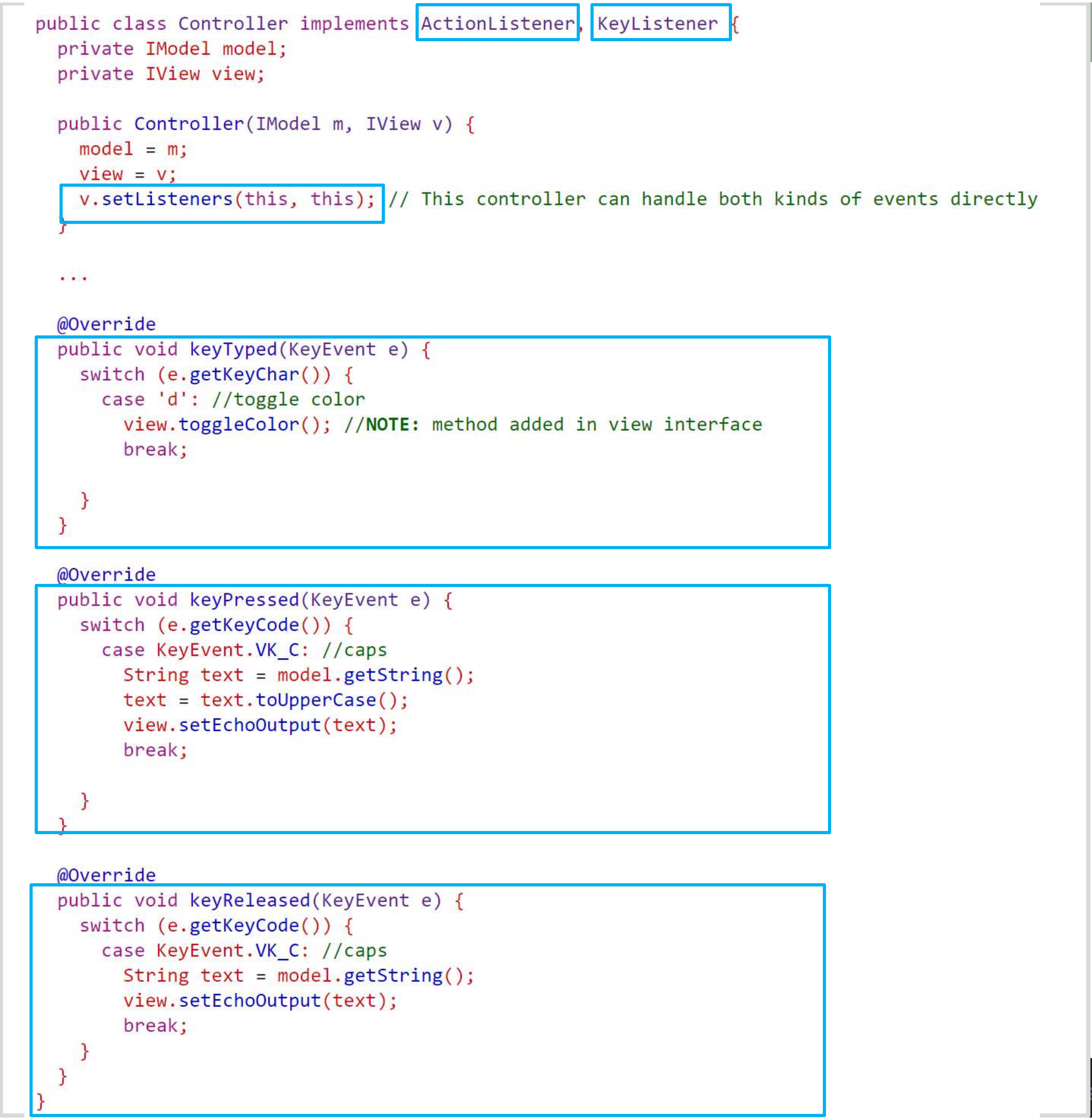
- ❑ Need to make our GUI to **respond** to **keyboard events**
  - ✓ Identify an **object** that **listens** to **key events**.
  - ✓ **Add** it to the **GUI** somewhere so that it always listens for key events Implement the above functionality **for** the **specific key** presses.

Adding keyboard inputs

- **KeyEvent**
  - ✓ Ex) **KEY_PRESSED**, **KEY_RELEASED**, **KEY_TYPED** (Pressed + Released)
  - ✓ Need **callbacks** for these events in the **`KeyListener`** interface
    - Must implement a **`KeyListener`** object → Connect it to a part of the **GUI** so that it listens to these events



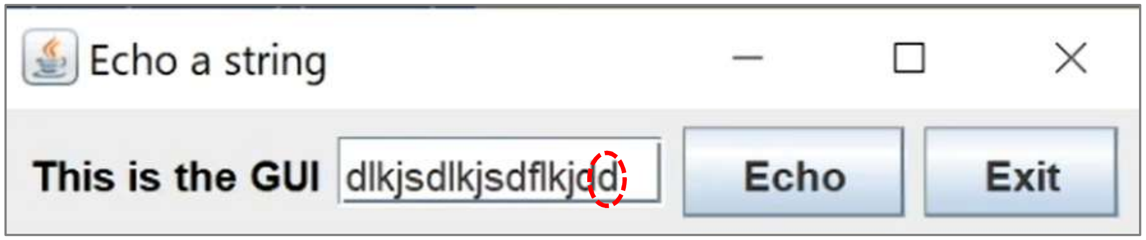
```java
/**
 * The interface for our view class
 */
public interface IView {
  void setListeners(ActionListener clicks, KeyListener keys); //NOTE: the second listener

  /**
   * Toggle the color of the displayed text. This is an explicit view operation because this is
   * something that only the view can control
   */
  void toggleColor();
}
```

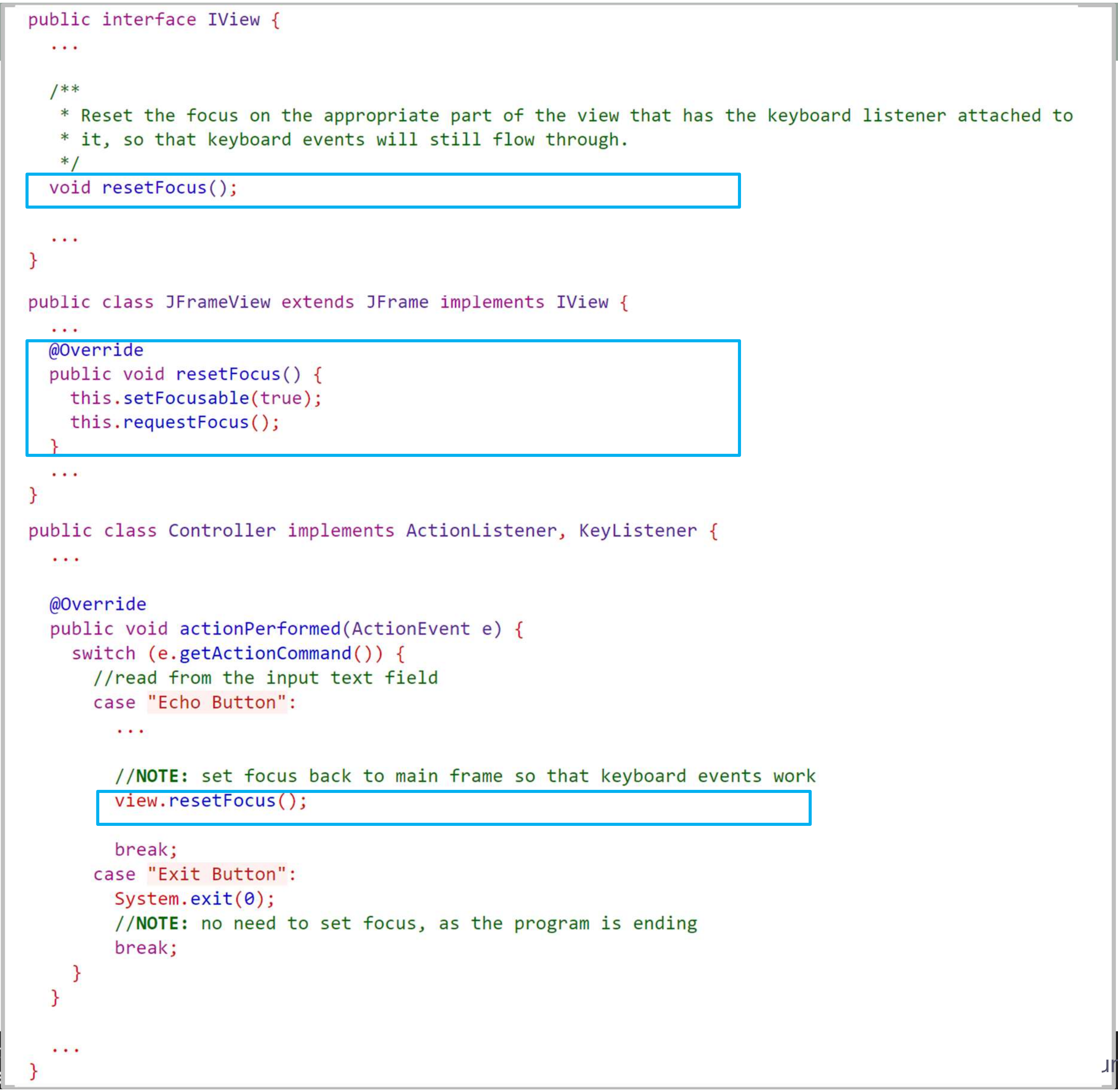


```java
public class Controller implements ActionListener, KeyListener {
  private IModel model;
  private IView view;

  public Controller(IModel m, IView v) {
    model = m;
    view = v;
    v.setListeners(this, this); // This controller can handle both kinds of events directly
  }

  ...

  @Override
  public void keyTyped(KeyEvent e) {
    switch (e.getKeyChar()) {
      case 'd': //toggle color
        view.toggleColor(); //NOTE: method added in view interface
        break;

    }
  }

  @Override
  public void keyPressed(KeyEvent e) {
    switch (e.getKeyCode()) {
      case KeyEvent.VK_C: //caps
        String text = model.getString();
        text = text.toUpperCase();
        view.setEchoOutput(text);
        break;

    }
  }

  @Override
  public void keyReleased(KeyEvent e) {
    switch (e.getKeyCode()) {
      case KeyEvent.VK_C: //caps
        String text = model.getString();
        view.setEchoOutput(text);
        break;
    }
  }
}
```

focusable

- `KeyEvents` are **captured only when** the UI part that the key listener is added to is **in focus**
  - ✓ → Need to make it **`focusable`** and calling its **`requestFocus()`** method.
  - ✓ → To **restore focus** from the controller, add a new method in the V → call it whenever a button is pressed.





```java
public interface IView {
  ...

  /**
   * Reset the focus on the appropriate part of the view that has the keyboard listener attached to
   * it, so that keyboard events will still flow through.
   */
  void resetFocus();

  ...
}

public class JFrameView extends JFrame implements IView {
  ...
  @Override
  public void resetFocus() {
    this.setFocusable(true);
    this.requestFocus();
  }
  ...
}

public class Controller implements ActionListener, KeyListener {
  ...

  @Override
  public void actionPerformed(ActionEvent e) {
    switch (e.getActionCommand()) {
      //read from the input text field
      case "Echo Button":
        ...

        //NOTE: set focus back to main frame so that keyboard events work
        view.resetFocus();

        break;
      case "Exit Button":
        System.exit(0);
        //NOTE: no need to set focus, as the program is ending
        break;
    }
  }

  ...
}
```

Configuring Keyboard Shortcuts

- Drawbacks
  - ✓ As **more key** shortcuts are supported, **Callback methods grow** quickly
  - ✓ There is **no easy way** to **change** the keyboard shortcuts while still offering

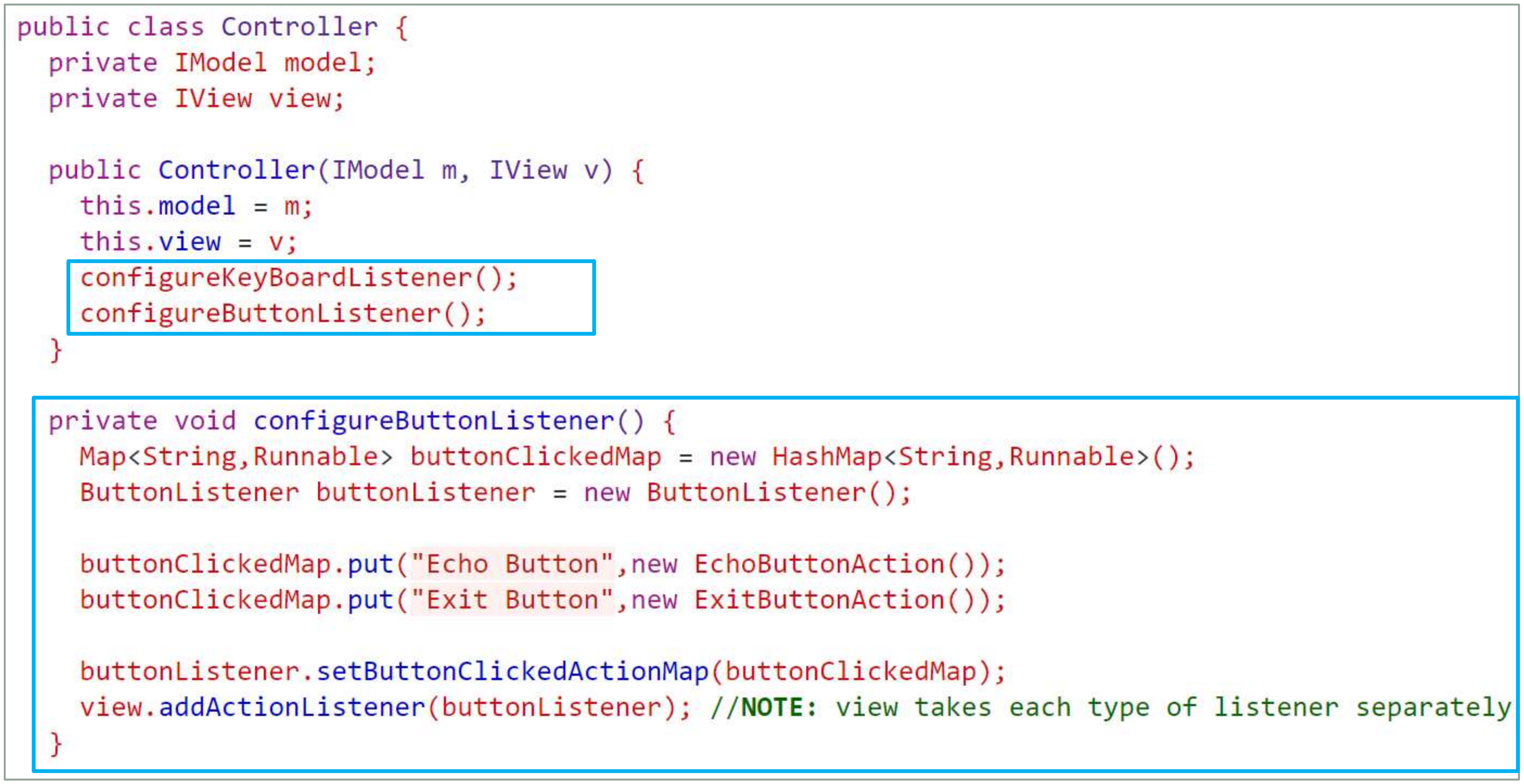
Use **`Map`** object

  - ✓ Unify **all** such **methods** as **`Map` objects** of **`Runnable`** interface
  - ✓ **1 `Map` object** for **each** type of **key event** with corresponding runnable-to-be-executed method
    - **keyPressedMap**, **keyReleasedMap**, **keyTypedMap**

```java
public class KeyboardListener implements KeyListener {
  private Map<Character, Runnable> keyTypedMap;
  private Map<Integer, Runnable> keyPressedMap, keyReleasedMap;

  /**
   * Set the map for key typed events. Key typed events in Java Swing are characters
   */

  public void setKeyTypedMap(Map<Character, Runnable> map) {
    keyTypedMap = map;
  }

  ...

  /**
   * This is called when the view detects that a key has been typed. Find if anything has been
   * mapped to this key character and if so, execute it
   */

  @Override
  public void keyTyped(KeyEvent e) {
    if (keyTypedMap.containsKey(e.getKeyChar()))
      keyTypedMap.get(e.getKeyChar()).run();
  }

  ...
}
```

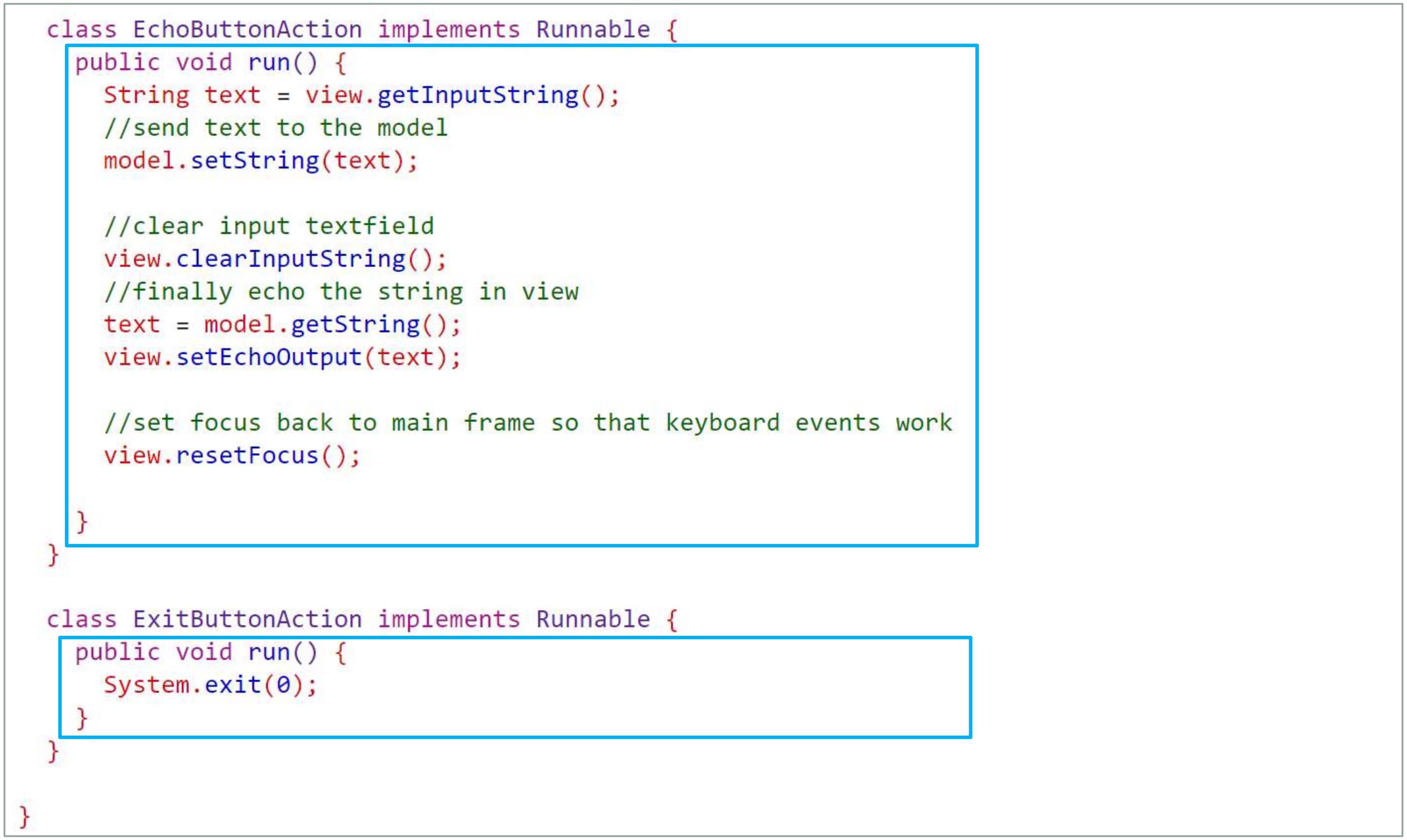
```java
public class Controller {
  private IModel model;
  private IView view;

  public Controller(IModel m, IView v) {
    this.model = m;
    this.view = v;
    configureKeyBoardListener();
    ...
  }

  /**
   * Creates and sets a keyboard listener for the view. In effect it creates snippets of
   * code as a Runnable object, one for each time a key is typed, pressed and released, only
   * for those that the program needs.
   *
   * Last we create our KeyboardListener object, set all its maps and then give it to the view.
   */
  private void configureKeyBoardListener() {
    Map<Character, Runnable> keyTypes = new HashMap<>();
    Map<Integer, Runnable> keyPresses = new HashMap<>();
    Map<Integer, Runnable> keyReleases = new HashMap<>();

    keyPresses.put(KeyEvent.VK_C, new MakeCaps()); //NOTE: see below
    keyReleases.put(KeyEvent.VK_C, new MakeOriginalCase()); //NOTE: see below
    ...

    KeyboardListener kbd = new KeyboardListener();
    kbd.setKeyTypedMap(keyTypes);
    kbd.setKeyPressedMap(keyPresses);
    kbd.setKeyReleasedMap(keyReleases);

    view.addKeyListener(kbd); //NOTE: view takes each type of listener separately

  }
  class MakeCaps implements Runnable {
    public void run() {
      String text = model.getString();
      text = text.toUpperCase();
      view.setEchoOutput(text);
    }
  }

  class MakeOriginalCase implements Runnable {
    public void run() {
      String text = model.getString();
      view.setEchoOutput(text);
    }
  }

  class ExitButtonAction implements Runnable {
    public void run() {
      System.exit(0);
    }
  }

}
```

Same Idea to **Action Listeners (Buttons)**



```
public class Controller {
  private IModel model;
  private IView view;

  public Controller(IModel m, IView v) {
    this.model = m;
    this.view = v;
    configureKeyBoardListener();
    configureButtonListener();
  }

  private void configureButtonListener() {
    Map<String,Runnable> buttonClickedMap = new HashMap<String,Runnable>();
    ButtonListener buttonListener = new ButtonListener();

    buttonClickedMap.put("Echo Button",new EchoButtonAction());
    buttonClickedMap.put("Exit Button",new ExitButtonAction());

    buttonListener.setButtonClickedActionMap(buttonClickedMap);
    view.addActionListener(buttonListener); //NOTE: view takes each type of listener separately
  }
```

Same Idea to Action Listeners (Buttons)



```
  class EchoButtonAction implements Runnable {
    public void run() {
      String text = view.getInputString();
      //send text to the model
      model.setString(text);

      //clear input textfield
      view.clearInputString();
      //finally echo the string in view
      text = model.getString();
      view.setEchoOutput(text);

      //set focus back to main frame so that keyboard events work
      view.resetFocus();

    }
  }

  class ExitButtonAction implements Runnable {
    public void run() {
      System.exit(0);
    }
  }

}
```

Decoupling of Controller and View

```
public class Controller implements ActionListener, KeyListener {
```

- **Limitations**
  - ✓ **Controller depends** on **View-specific** interfaces, **ActionListener** or **KeyListener**
    - ➔ **View-specific** details **leak out**
    - ➔ **Changing** the **view** implementation may cause **changes** in the **Controller**

- High-level **capabilities** of View
  - ✓ Echo on (some part of) the view a string
  - ✓ Toggle the color of the text shown by (some part of) the view
  - ✓ Display the text shown by (some part of) the view in upper case
  - ✓ Restore the case of the text displayed by (some part of) the view
  - ✓ Exit the program

⇩

Encapsulate each as a **callback** function in a **common interface**

```java
public interface Features {
  void echoOutput(String typed);
  void toggleColor();
  void makeUppercase();
  void restoreLowercase();
  void exitProgram();
}
```

```java
public class Controller implements Features {
    private IModel model;
    private IView view;

    public Controller(IModel m) {
        model = m;
    }

    public void setView(IView v) {
        view = v;
        //provide view with all the callbacks
        view.addFeatures(this);
    }
    ...
}

/**
 * The interface for our view class
 */
public interface IView {
    ...

    void addFeatures(Features features); //NOTE: this replaces addListeners(..)
}
```

```java
@Override
public void addFeatures(Features features) {
  //connect echoOutput callback to the clicking of the echo button
  echoButton.addActionListener(l->features.echoOutput(input.getText()));

  //NOTE: connect exitProgram to the clicking of the exit button
  exitButton.addActionListener(l->features.exitProgram());
  this.addKeyListener(
          new KeyListener() {

            @Override
            public void keyTyped(KeyEvent e) {
              if (e.getKeyChar()=='d') {
                //NOTE: connect toggleColor callback to typing 'd'

                features.toggleColor();
              }
            }

            @Override
            public void keyPressed(KeyEvent e) {
              if (e.getKeyCode()==KeyEvent.VK_C) {
                //NOTE: connect makeUppercase callback to pressing 'c'

                features.makeUppercase();
              }
            }

            @Override
            public void keyReleased(KeyEvent e) {
              if (e.getKeyCode()==KeyEvent.VK_C) {
                //NOTE: connect restoreLowercase callback to releasing 'c'

                features.restoreLowercase();
              }
            }
          }
  );
}
```