

Lecture 6

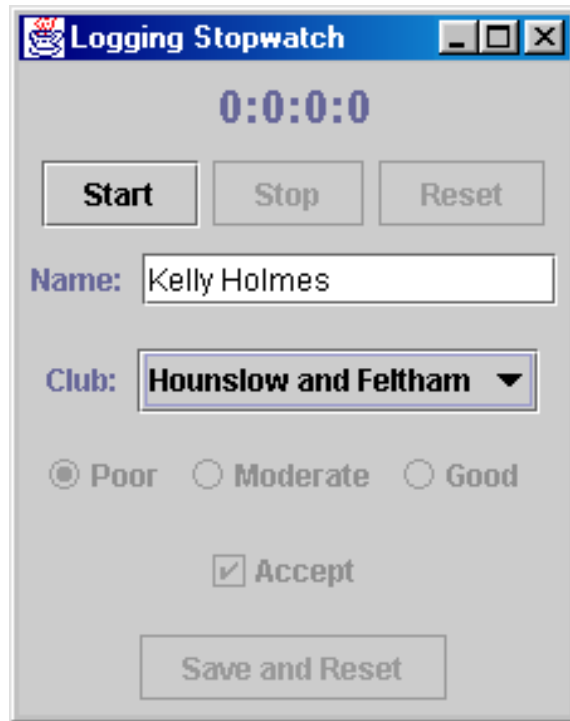
The Logging Stopwatch artefact

- selection components, JTextField

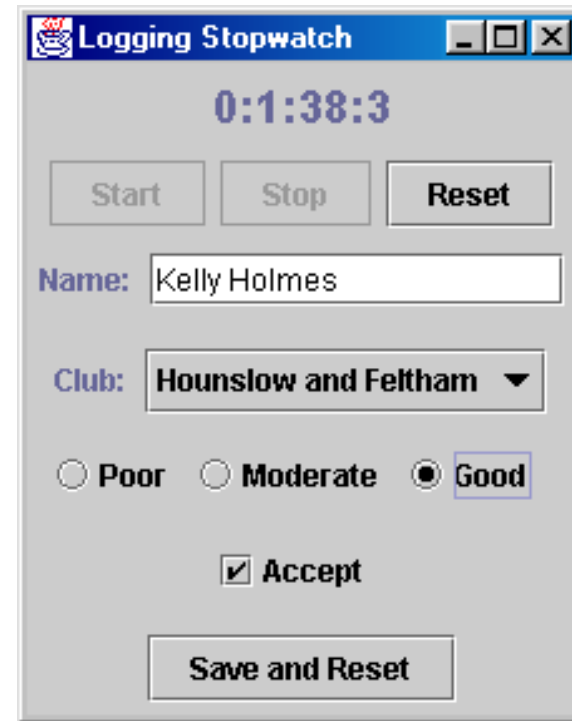
Selection-type components

- So far we have looked in detail at the operation of `JButton`. This allows the user to *initiate some action*, but normally the visual appearance of the button does not change after the user presses it – unless it is specifically disabled by the program code as in the case of the Stopwatch artifact
- Selection-type components are for the user to make a selection or toggle a “yes-or-no” condition. The visual state of a selection-type component changes as a result of the user interacting with the component.
- We will be looking at
 - single selection-type components `JComboBox` and `JCheckBox`
 - `JRadioButtons` which can be assembled into a composite selection component
- We will also be introducing `JTextField` into the LoggingStopwatch artefact

The LoggingStopwatch artefact



The screenshot shows the 'Logging Stopwatch' application window. At the top, the title bar reads 'Logging Stopwatch'. Below the title bar, the timer display shows '0:0:0:0'. Underneath the timer are three buttons: 'Start', 'Stop', and 'Reset'. Below these buttons is a text input field labeled 'Name:' containing the text 'Kelly Holmes'. Below the name field is a dropdown menu labeled 'Club:' with 'Hounslow and Feltham' selected. Below the dropdown are three radio buttons: 'Poor' (selected), 'Moderate', and 'Good'. Below the radio buttons is a checkbox labeled 'Accept' which is checked. At the bottom is a button labeled 'Save and Reset'.

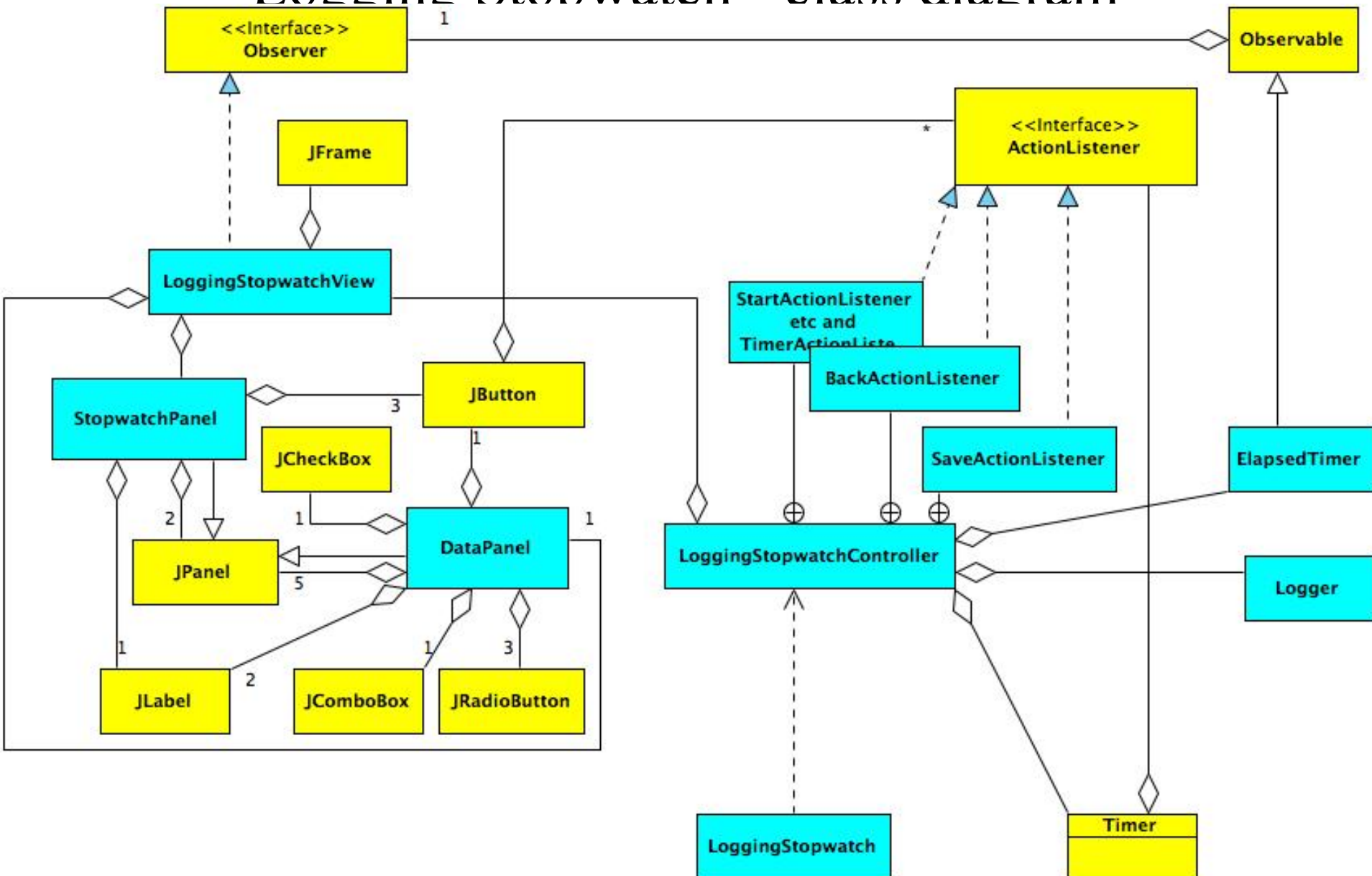


The screenshot shows the 'Logging Stopwatch' application window after some time has passed. The timer display now shows '0:1:38:3'. The 'Name:' field still contains 'Kelly Holmes' and the 'Club:' dropdown still shows 'Hounslow and Feltham'. However, the 'Good' radio button is now selected, while 'Poor' and 'Moderate' are unselected. The 'Accept' checkbox remains checked. The 'Save and Reset' button is still at the bottom.

Logging Stopwatch artefact (cont'd)

- The artefact is an extension of Stopwatch with several user interface components added
 - An extra JButton to carry out the *Save and Reset* function
 - A JTextField for entry of the name of the runner
 - A JComboBox to indicate the club to which the runner belongs
 - Three JRadioButtons for the choice of Poor/Moderate/Good to classify the runner's performance
 - One JCheckBox to indicate whether the runner has been selected for the national running team
- After the stopwatch has been stopped, the user can chose the various options presented by the selection components and can then press the *Save and Reset* button. This causes the Name and the other data chosen to be displayed in the console.

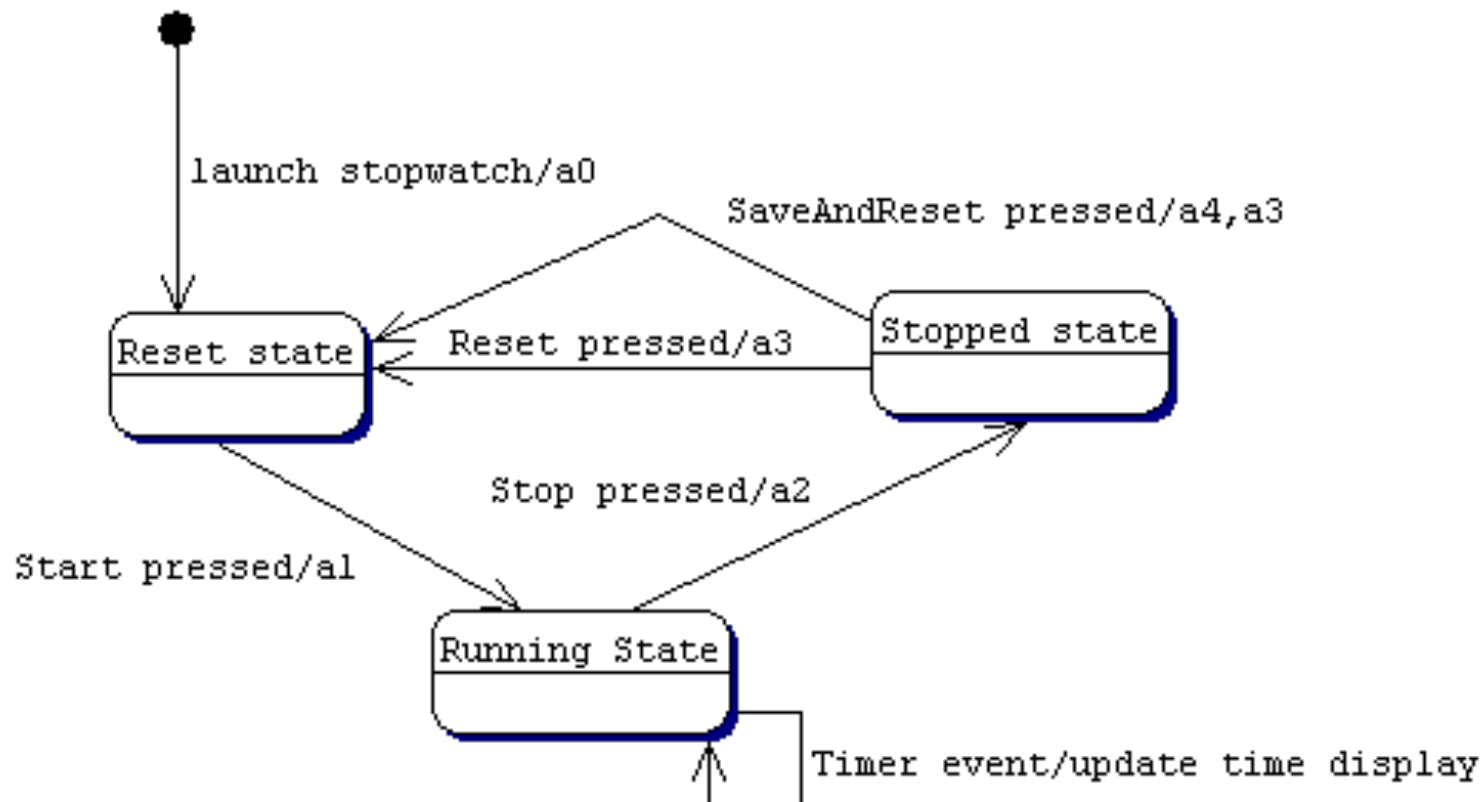
Logging Stopwatch - class diagram



LoggingStopwatch class diagram (cont'd)

- As can be seen from the diagram on the previous slide, the structure is an extension of that of Stopwatch
- LoggingStopwatchView now aggregates StopwatchPanel and DataPanel. The JRadioButtons, the JCheckBox, the JComboBox and the extra JButton *saveButton* are part of DataPanel and *saveButton* is registered with SaveActionListener - another inner class of LoggingStopwatchController.
- Additionally an extra application layer class called Logger has been added which is responsible for displaying the results of each run to the console.
- Note that the View layer does not get any extra data from the Model layer, beyond that already obtained in Stopwatch.

LoggingStopwatch – state transition diagram



a0: enable Start; disable Stop, Reset, SaveAndReset, data entry components
a1: start timing; enable Stop; disable Start, Reset, SaveAndReset, data entry components
a2: stop timing; enable Reset, SaveAndReset, data entry components; disable Start, Stop
a3: reset display; enable Start; disable Stop, Reset, SaveAndReset, data entry components
a4: Log details

DataPanel - field declarations 1

```
. . .  
private JTextField name;  
private JComboBox club;  
private JRadioButton speedGood;  
private JRadioButton speedModerate;  
private JRadioButton speedPoor;  
. . .
```

- *name* will refer to an instance of **JTextField**. This will be used for entry of the runner's name
- *club* will refer to a **JComboBox** which will display the athletic clubs from which the user will make their selection
- *speedGood*, *speedModerate*, *speedPoor* will refer to the three **JRadioButtons**, one of which will be selected for the user to give the grading of the runner.

DataPanel - field declarations 2

```
. . .  
private JCheckBox accept;  
private JButton saveButton;  
private ButtonGroup speedGroup;  
private static final Object[] clubList = {"Belgrave  
Harriers", "Loughborough", "Newcastle Road Stars", "Hounslow and  
Feltham"};  
. . .
```

- *accept* will refer to the JCheckBox which the user will either tick or untick to indicate whether the runner will be accepted in the national team
- *saveButton* will refer to the JButton "Save and Reset" which the user will press if they wish the time and the data entered to be logged
- *speedGroup* will refer to a ButtonGroup to which the JRadioButtons will be added to give composite radio button behaviour
- *clubList* refers to an Object[] array storing the list of strings corresponding to the club names that will appear in the *club* JComboBox

DataPanel- constructor 1

```
public DataPanel(LoggingStopwatchController controller) {  
    . . .  
    name = new JTextField(15);  
    club = new JComboBox(clubList);  
    speedGood = new JRadioButton("Good");  
    . . .  
}
```

- The constructor is passed a reference *controller* to the `LoggingStopwatchController`. This will be used to enable the *saveButton* to register an instance of the inner class `SaveActionListener`.
- The *name* `JTextField` is constructed using the version of the constructor that allocates it the given number of characters (i.e.15 here)
- The *club* `JComboBox` is constructed using the version of the constructor `JComboBox (Object[] items)` that creates a `JComboBox` containing the elements in the specified array.
- The *speedGood* `JRadioButton` is constructed to display "Good". The other `JRadioButtons` are constructed in a similar fashion.

DataPanel - constructor 2

. . .

```
accept = new JCheckBox("Accept");  
saveButton = new JButton("Save and Reset");  
saveButton.addActionListener(controller.new  
    SaveActionListener());
```

. . .

- The `JCheckBox` *accept* is constructed to display text "Accept".
- The `JButton` *saveButton* is constructed to display "Save and Reset" and a new instance of the inner class `SaveActionListener` of `LoggingStopwatchController` is registered with it
 - Which means its *actionPerformed(...)* method will be run when the button is pressed in the operation phase.

DataPanel – constructor 3

```
. . .  
speedGroup = new ButtonGroup();  
speedGroup.add(speedGood);  
speedGroup.add(speedModerate);  
speedGroup.add(speedPoor);  
speedGroup.setSelected(speedModerate.getModel(), true);  
. . .
```

- The *speedGroup* ButtonGroup is then constructed and the 3 JRadioButtons are *add()*ed to it to give radio button behaviour
- The *speedModerate* JRadioButton is set to be the default selected. This is achieved by calling the *setSelected()* method on the ButtonGroup, supplying as the first parameter the ButtonModel of *speedModerate* - obtained by calling *getModel()* on *speedModerate*.

DataPanel

- *getAccepted()* and *getSpeedQuality()*

```
public boolean getAccepted() {  
    return accept.isSelected();  
}
```

```
public String getSpeedQuality() {  
    if (speedPoor.isSelected()) {  
        return "Poor";  
    } else if (speedModerate.isSelected()) {  
        return "Moderate";  
    } else {  
        return "Good";  
    }  
}
```

- The *getAccepted()* method returns whether the *accept* JCheckBox is ticked by querying it via the *isSelected()* method
- The *getSpeedQuality()* method returns one of three strings by querying two of the JRadioButtons for whether they are selected via the *isSelected()* method

DataPanel

- *getName()* and *getClub()*

```
public String getName(){  
    return name.getText();  
}
```

```
public String getClub{  
    return (String) (club.getSelectedItemAt());  
}
```

- *getName()* calls the *getText()* method on the *JTextField name* to obtain the text currently entered in the field, and returns the *String* to the caller
- *getClub()* calls the method (belonging to *JComboBox*):

Object **getSelectedItem()**

in order to obtain the item currently selected in the *club JComboBox*. The returned value needs to be cast from *Object* to *String* before returning it to the caller.

DataPanel

- *enableDataControls*

```
public void enableDataControls(boolean b) {  
    club.setEnabled(b);  
    speedPoor.setEnabled(b);  
    speedModerate.setEnabled(b);  
    speedGood.setEnabled(b);  
    name.setEnabled(b);  
    accept.setEnabled(b);  
    saveButton.setEnabled(b);  
}
```

- This method is called by the controller to either enable or disable data entry on the DataPanel. All the widgets are disabled if data entry is disabled.

LoggingStopwatchView

```
public class LoggingStopwatchView implements Observer {
    private StopwatchPanel stopwatchPanel;
    private DataPanel dataPanel;
    . . .
    public LoggingStopwatchView
        (LoggingStopwatchController controller) {
        . . .
        dataPanel = new DataPanel(controller);

        frame.add(stopwatchPanel, BorderLayout.NORTH);
        frame.add(dataPanel, BorderLayout.SOUTH);
        frame.pack();
    }
    . . .
}
```

- There is an extra field *dataPanel* which will reference the DataPanel (see previous slides)
- In the constructor, the DataPanel is constructed based on the controller passed in.
- The StopwatchPanel is now added to the North of the frame and the DataPanel is added to the South (JFrame by default has BorderLayout)

LoggingStopwatchView – getters

```
. . .  
public boolean getAccepted() {  
    return dataPanel.getAccepted();  
}  
  
public String getSpeedQuality() {  
    return dataPanel.getSpeedQuality();  
}  
  
public String getName() {  
    return dataPanel.getName();  
}  
  
public String getClub() {  
    return dataPanel.getClub();  
}  
. . .
```

- These getters merely pass through the getter calls to the DataPanel

LoggingStopwatchView – setting state

```
public void setResetState() {  
    stopwatchPanel.setButtons(true, false, false);  
    dataPanel.enableDataControls(false);  
}
```

```
public void setRunningState() {  
    stopwatchPanel.setButtons(false, true, false);  
    dataPanel.enableDataControls(false);  
}
```

```
public void setStoppedState() {  
    stopwatchPanel.setButtons(false, false, true);  
    dataPanel.enableDataControls(true);  
}
```

- The methods for setting state now contain an extra call to the *enableDataControls()* method of the **DataPanel** to either enable the data entry (with parameter `true`) or disable data entry (with parameter `false`). Data entry is only enabled in the Stopped state.

LoggingStopwatchController

- field declarations and constructor

```
public class LoggingStopwatchController {  
    . . .  
    private Logger theLogger;  
    . . .  
    public LoggingStopwatchController() {  
        . . .  
        theLogger = new Logger();  
        . . .  
    }  
    . . .  
}
```

- The field *theLogger* holds a reference to an instance of `Logger` - which will have the job of logging the time reading and the data entered.

LoggingStopwatchController

- inner class SaveActionListener - 1

```
...  
public class SaveActionListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        if (theState == STOPPED) {  
            String acceptedString;  
            if (theView.getAccepted()) {  
                acceptedString = "accepted";  
            } else {  
                acceptedString = "not accepted";  
            }  
        }  
        ...  
    }  
}  
...
```

- The actionPerformed() method of the inner class SaveActionListener will be called when Save and Reset button is clicked.
- In the Stopped state, local String variable *acceptedString* is set to “accepted” or “not accepted”, depending on the value returned from the LoggingStopwatchView method *getAccepted()*;

LoggingStopwatchController

- inner class SaveActionListener - 2

```
. . .  
public class SaveActionListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        . . .  
        theLogger.log(elapsedTimer.getTime(), theView.getName(),  
                      theView.getClub(), theView.getSpeedQuality(),  
                      acceptedString);  
        elapsedTimer.reset();  
        theView.setResetState();  
        theState = RESET;  
    }  
}  
. . .
```

- The *actionPerformed()* method continues by calling *log()* on the **Logger** passing as parameters time obtained from the **ElapsedTimer** and the data obtained for the run from the **View**.
- It then calls *reset()* on the **ElapsedTimer** – which will have the effect of propagating this change (with zero value) to the displayed time on the **View**; it then gets the **View** to set the appropriate enabling/disabling for the reset state and finally, transitions to the reset state

The class Logger

```
public class Logger extends Object {  
    public Logger() {  
        super();  
    }  
  
    public void log(String time,String name, String club,  
                    String quality,String accepted){  
        System.out.println();  
        System.out.println(time);  
        System.out.println(name);  
        System.out.println(club);  
        System.out.println(quality);  
        System.out.println(accepted);  
    }  
}
```

- This class is part of the Model but unlike ElapsedTimer doesn't actually involve changing the View
- It merely takes the string for the different pieces of data for a run, passed in as parameters, and prints them out to the console.