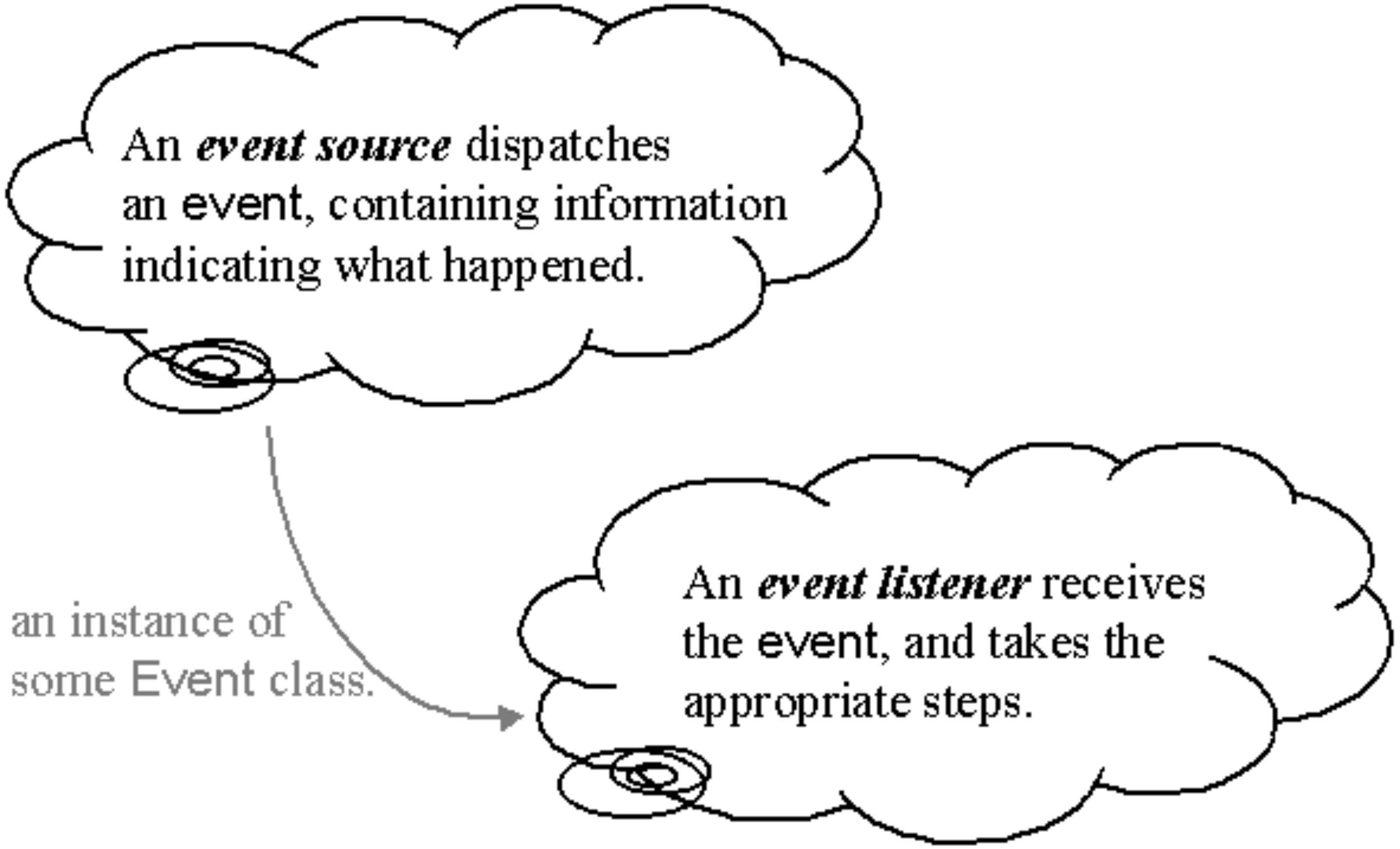
Lecture 2
Event sources and listeners

Event Sources and Listeners


An ***event source*** dispatches an event, containing information indicating what happened.
an instance of some Event class.
An ***event listener*** receives the event, and takes the appropriate steps.

Event sources and listeners (cont'd)

- The diagram on the previous slide is the Java 1.1 event listener model.
- An event is dispatched when *something* happens. The class of the event and the value of its attributes indicate *what* has happened.
- The following is required for the listener and the event source to be connected
  - The listener must implement an **interface** associated with the event
  - The listener must be *registered* with the event source.

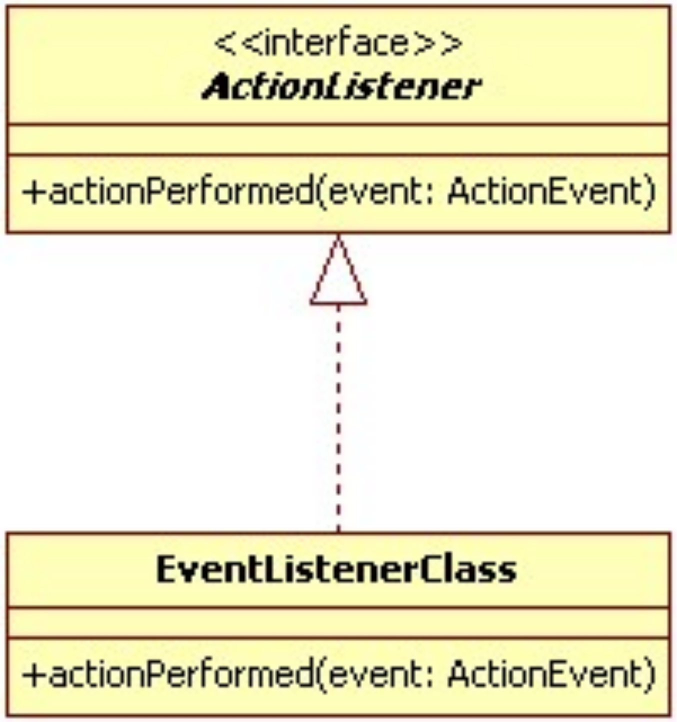
Event source: listener registration

Methods in a class that acts as a source of ActionEvents:

```
public void addActionListener (ActionListener aListener)
public void removeActionListener(ActionListener aListener)
  .
  .
  .
```

- The two methods shown indicate that the class is an event source, which is capable of generating ActionEvents (in this case) and maintaining a list of ActionListeners.
- When the ActionEvents are generated, they are dispatched to the ActionListeners.
- Examples of classes that act as event sources of ActionEvents are:
  - javax.swing.Timer
  - javax.swing.JButton

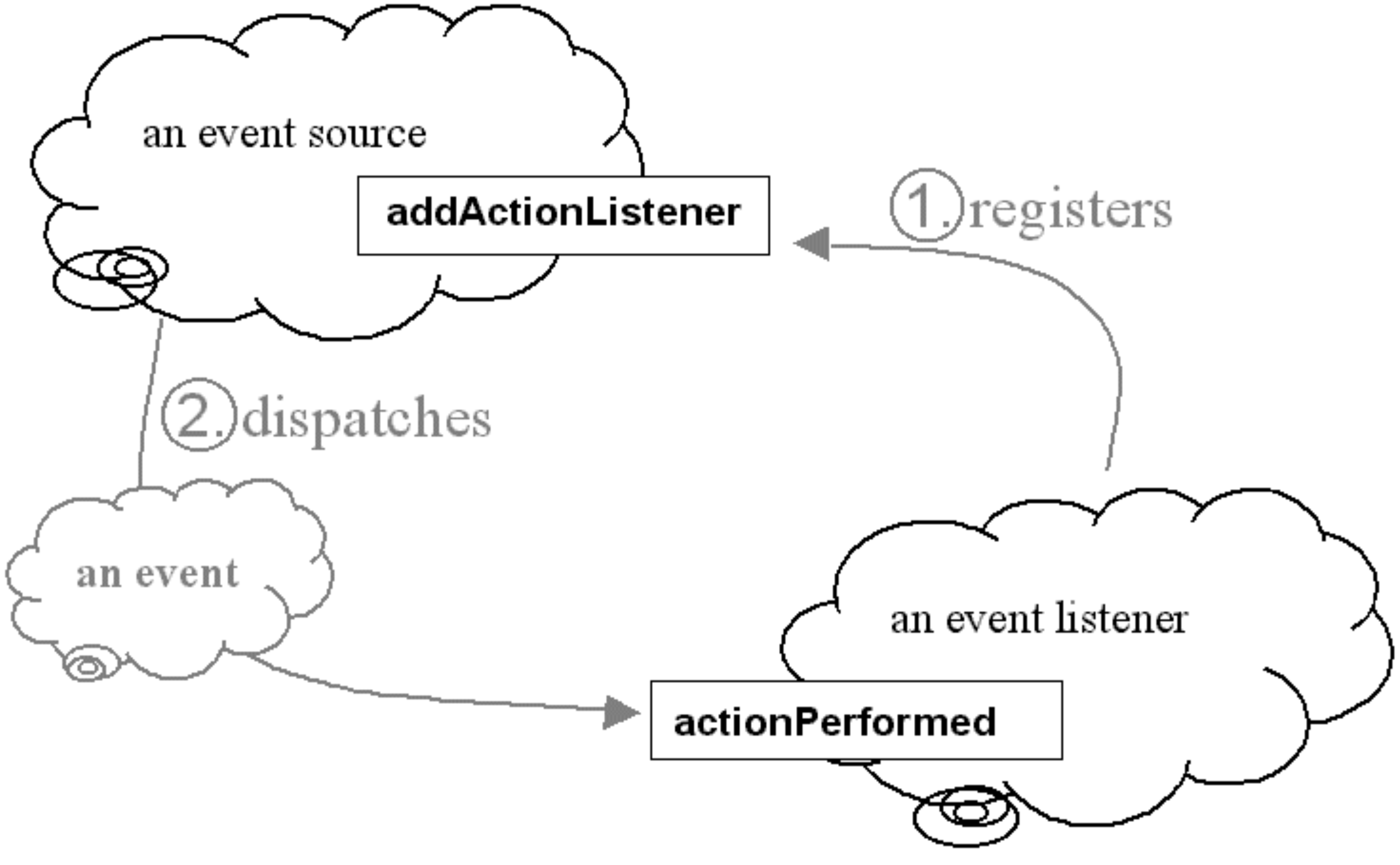
Event listeners, implementing interfaces


<<interface>>
ActionListener
+actionPerformed(event: ActionEvent)
EventListenerClass
+actionPerformed(event: ActionEvent)

Event listeners, implementing interfaces

- In order to be allowed to register as a listener the actual listener must be compatible with the formal argument, ActionListener.
- An **interface** specifies the methods that a class which implements the interface *must* supply. An interface is not the same as a class. There is no method body to any of methods specified - just the signature
- ActionListener is an interface with a single method actionPerformed() specified, taking an argument of the ActionEvent class. For a class implementing the ActionListener interface a definition of the method *actionPerformed()* is compulsory - the class won't compile if it is absent.
- When the event source generates an ActionEvent, a copy of it is passed as an argument to the *actionPerformed()* method of each of its registered listeners.

Event listener protocols



Event listener protocols (cont'd)

1. The event listener, which must implement the ActionListener interface, is registered with the event source by passing it as an argument to its *addActionListener()* method.

2. The event source can then dispatch ActionEvents passing them as arguments to the listener's *actionPerformed()* method.

- In order for the listener to be registered, it must declare that it implements the ActionListener interface. . .
  - so to be compiled it must implement an *actionPerformed()* method
- Note that the event source class doesn't need to know the actual class of the event listener passed to it in *addActionListener()*. The object passed as a parameter must only satisfy the condition that its class implements ActionListener - i.e. has an *actionPerformed()* method
  - this is an example of *polymorphism*

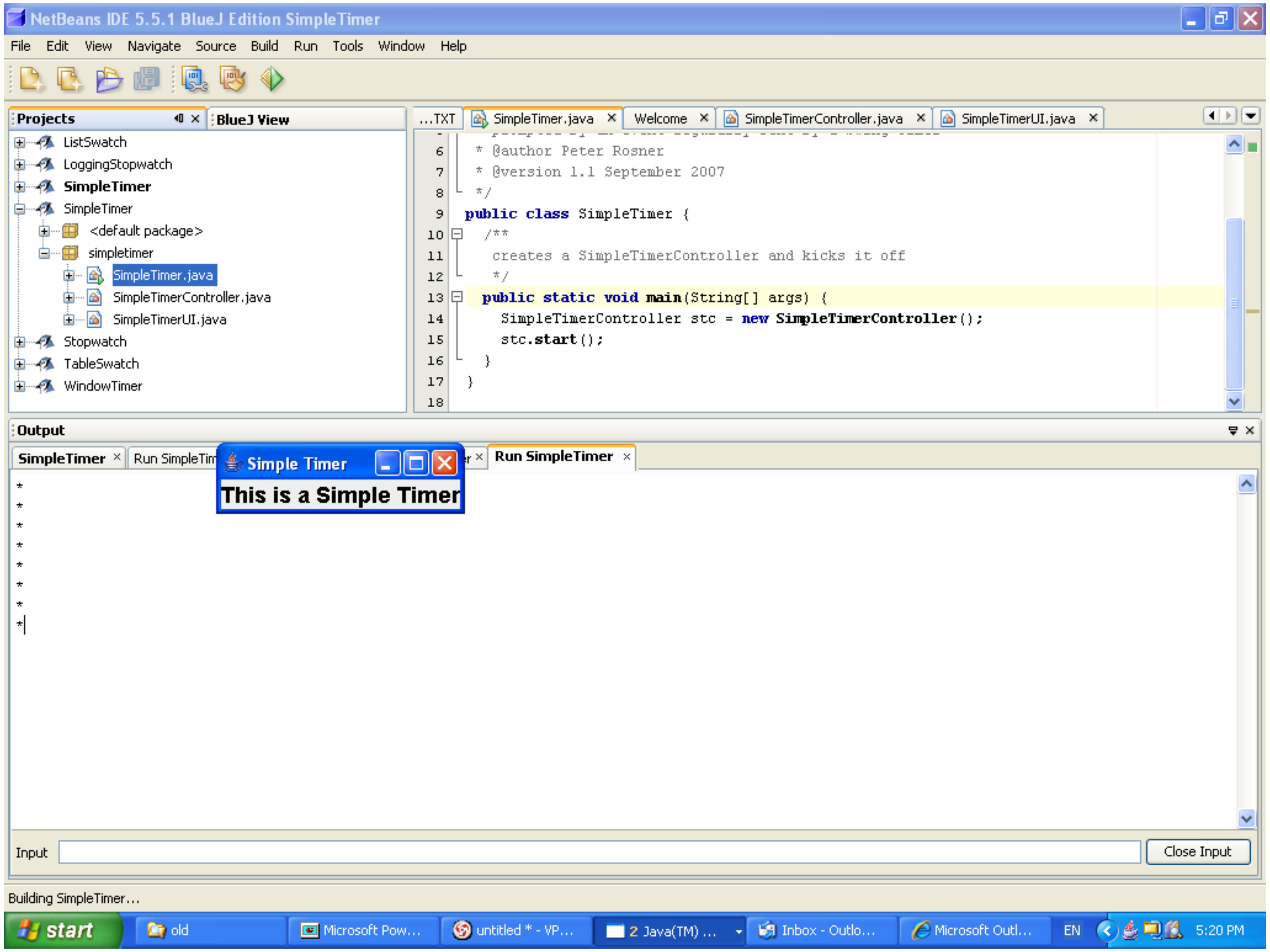
Timer constructor and methods

```
javax.swing.Timer class
constructor:
    Timer(int delay,ActionListener listener)

methods:
  public void addActionListener (ActionListener aListener)
  public void removeActionListener(ActionListener aListener)
  public void start()
  ..
  and lots more!!
```

- The Timer class is in the javax.swing (JFC) package of classes.
- The *addActionListener()* and *removeActionListener()* methods indicate that it is an event source
- Instances of the class will dispatch an ActionEvent (approximately) every *delay* milliseconds once the *start()* method has been called.

The SimpleTimer artefact



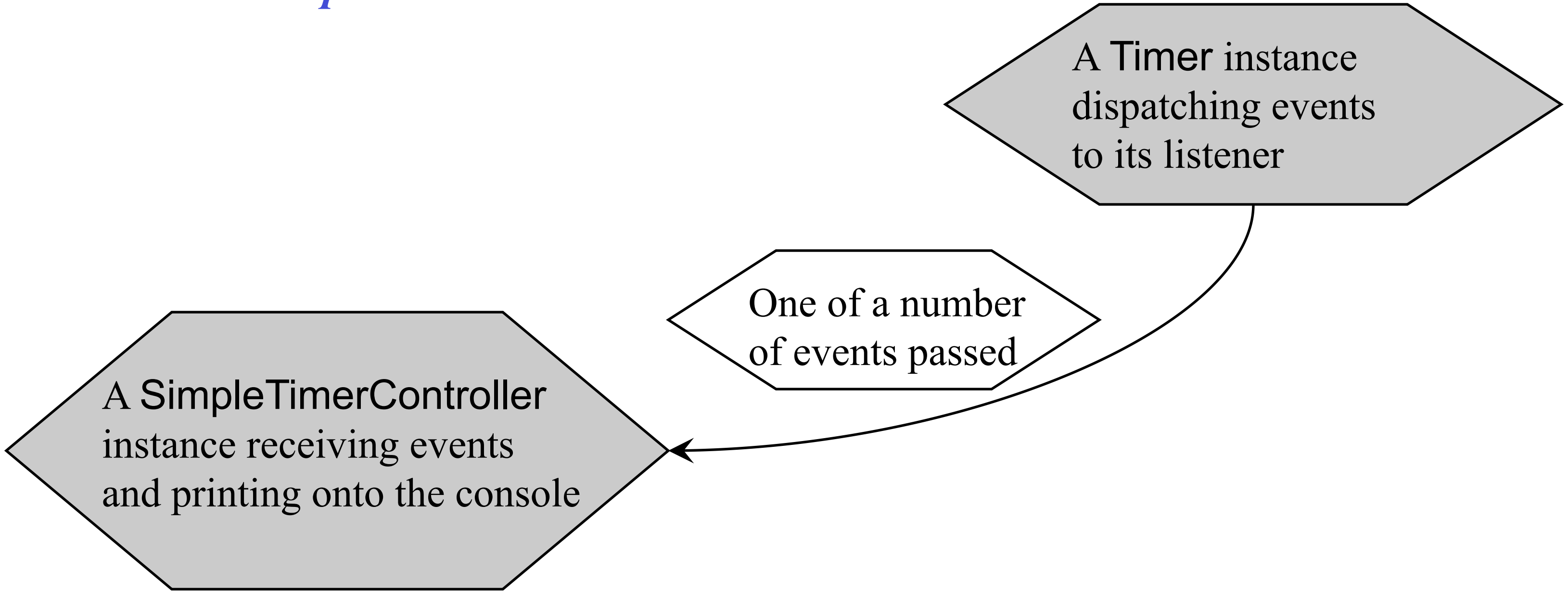
```
 * @author Peter Rosner
 * @version 1.1 September 2007
 */
public class SimpleTimer {
  /**
   creates a SimpleTimerController and kicks it off
   */
  public static void main(String[] args) {
    SimpleTimerController stc = new SimpleTimerController();
    stc.start();
  }
}
```

This is a Simple Timer

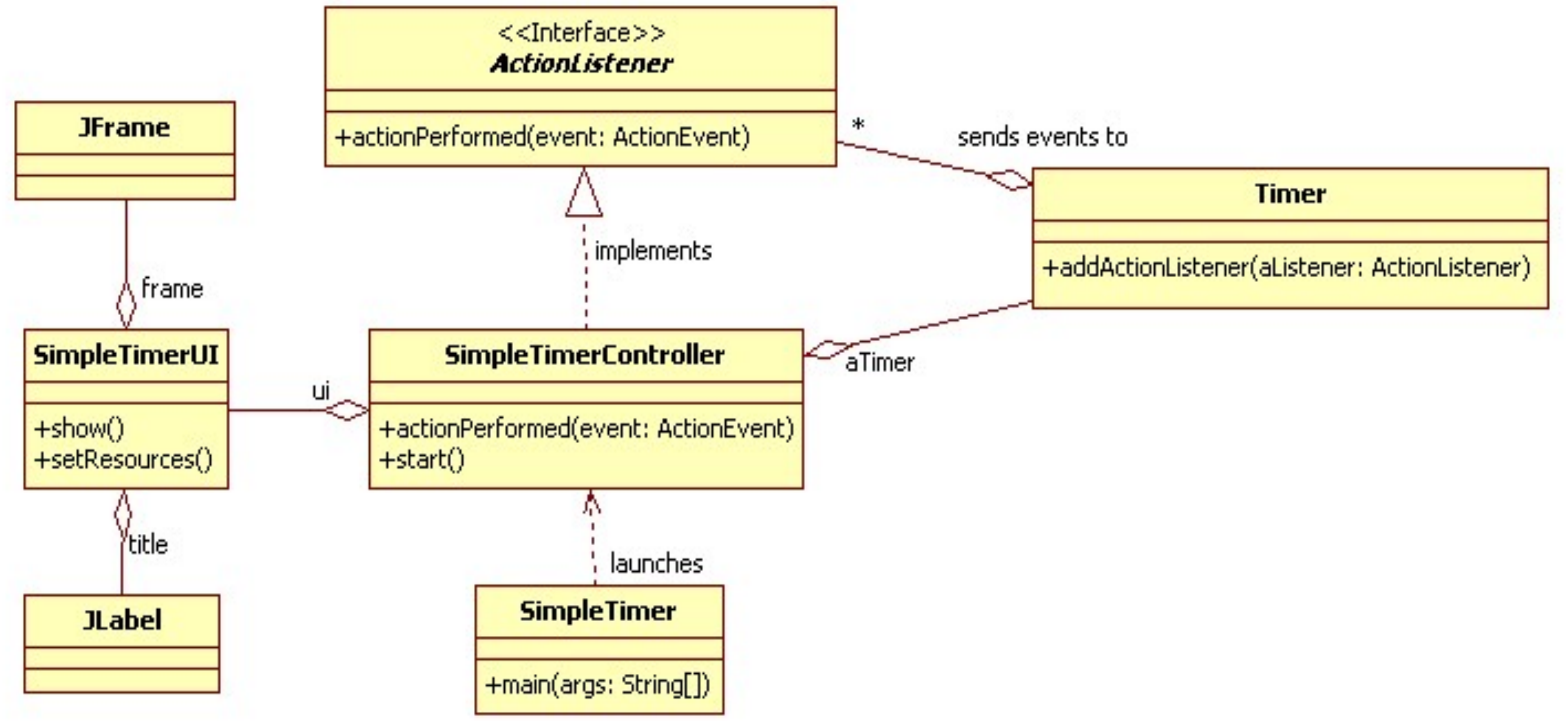
```
*
*
*
*
*
*
*
*
```

The SimpleTimer artefact - overview

- An instance of the SimpleTimerController class will construct an instance of the Timer class and register itself with it as its listener.
  - in the ***Preparation Phase***
- Once it has started the Timer instance it will receive a series of ActionEvents. On the receipt of each event it will print a message on the console
  - in the ***Operation Phase***


A Timer instance dispatching events to its listener
One of a number of events passed
A SimpleTimerController instance receiving events and printing onto the console

The SimpleTimer -class diagram



The class SimpleTimer

```
package simpletimer;
...
public class SimpleTimer extends Object {
   /**
   creates a SimpleTimerController and kicks it off
   */
  public static void main(String[] args) {
    SimpleTimerController stc = new SimpleTimerController();
    stc.start();
  }
}
```

- The artefact is started by running this class
- It constructs an instance of SimpleTimerController and start()s it

SimpleTimerController - 1

```
package simpletimer;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
. . .
public class SimpleTimerController
  implements ActionListener {
    . . .
    private Timer           aTimer;
    . . .
    private SimpleTimerUI   ui;
```

- After the declaration of the package and the imports, the SimpleTimerController class is declared
- the encapsulated Timer named *aTimer* and SimpleTimerUI named *ui* are declared but not instantiated
  - Timer is a class for generating ActionEvents regularly as described in an earlier slide
  - SimpleTimerUI is class responsible for generating the display of the fixed label for this artefact

SimpleTimerController - 2

```
. . .
 public SimpleTimerController() {
      super();
      ui = new SimpleTimerUI();
      aTimer = new Timer( 500, this);
 }
 . . .
```

- The constructor for the class
  - creates a new instance of SimpleTimer
  - creates a new instance of Timer which (when started later) will run with a period of half a second and will send ActionEvents to the SimpleTimerController currently under construction (i.e the argument ***this***)

SimpleTimerController - 3

```
. . .
public void start() {
     ui.show();
     aTimer.start();
}
 . . .
```

- The *start()* method
  - invokes *show()* on the SimpleTimerUI object causing it to actually display the user interface and to start the Swing event loop - a thread for monitoring events from the user. Since in this artefact there are no user events, this will just be an infinite loop!!
  - invokes *start()* on the Timer object which causes it to start "ticking" with the period defined when the object was constructed earlier and sending ActionEvents to ***this***
- Note that after this method is called from SimpleTimer the main Java thread exits. For every "tick" when the Timer thread yields control, the Swing event loop thread assumes control. Without the presence of another thread, the Timer thread yielding control would cause the artefact to exit!!

SimpleTimerController - 4

```
    ...
    public void actionPerformed( ActionEvent event) {
         System.out.println("*");
    }
}//end of SimpleTimerController
```

- The *actionPerformed()* method will be called every time the Timer "ticks". This because the SimpleTimerController was set up to be a listener of the Timer
- The body of the method indicates that each time this occurs, an asterisk is printed to the console.

SimpleTimerUI -1

```
package simpletimer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.*;

public class SimpleTimerUI {
    // fields
   . . .
    private JFrame frame;
    private JLabel title;
```

- After the declaration of the packages and the imports, the SimpleTimerUI class is declared
- the encapsulated JFrame named *frame*, and JLabel named *title* are declared but not instantiated.
  - JFrame is a class that represents an enclosing window, and JLabel is a class used for presenting text that cannot be modified by the user.

SimpleTimerUI -2

```
. . .
public SimpleTimerUI() {
    super();
    this.setResources();
    frame = new JFrame("Simple Timer");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    title = new JLabel("This is a Simple Timer");
    frame.add(title);
    frame.pack();
}
. . .
```

- The constructor
  - invokes *setResources()* which will be described in a later lecture
  - creates the *frame* object using a JFrame constructor whose parameter specifies the contents of the title bar

SimpleTimerUI -3

- invokes *setDefaultOnClose()* on *frame*, with the static constant EXIT_ON_CLOSE (belonging to JFrame) as argument. This will ensure that the program is terminated when the user clicks on the close box.
- creates the *title* object via a version of the constructor of JLabel that takes as parameter the string to be displayed.
- invokes *add()* on *frame* with the *title* object as argument. This will cause the JLabel to fill the window represented by the JFrame
- invokes *pack()* on frame which ensures that the JFrame takes only the size required by its contents - in this case the JLabel

SimpleTimerUI - 4

```
    . . .
    public void show() {
        frame.setVisible(true);
    }
    . . .
    private void setResources() {
      . . . . .
    }
}//end of SimpleTimerUI
```

- The *show()* method invokes *setVisible()* on the JFrame causing it to appear to the user and also starts the Swing event loop which monitors for user events. Since there is no user input in this artefact, this will be an infinite loop that will only be exited when the user clicks on the close box of the window
- The *setResources()* method will be covered in a future lecture