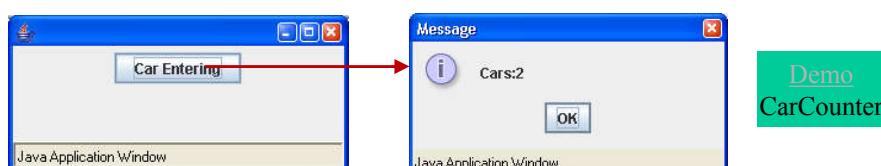


Events and GUIs (JFS Chap 6)

- In this section:
 - The need for "global" variables
 - Using a text field for output
 - Updating graphics displays
 - The concept of *event-driven programs*, and the *event loop*
 - "Event-handlers" and methods
 - Using an interactive `JSlider` "widget"
 - Using `JLabel` widgets
 - About `FlowLayout`

- Chap 6 also contains random numbers, timers:
 - Private reading

Counting cars (JFS, p89)



- A simple application to count cars entering a car park
 - Each button press increases an internal counter variable
 - And displays the new total on the screen
- Overall this application is not surprising:
 - One button
 - `actionPerformed` counts and displays message dialogue
- The problem:
 - Where should we store the count?

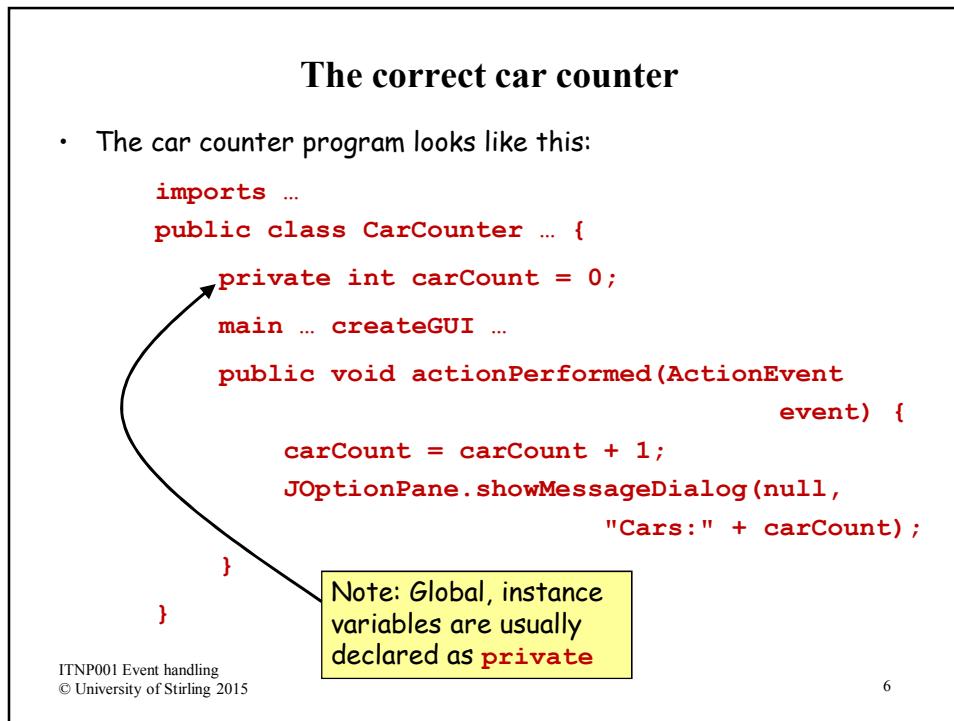
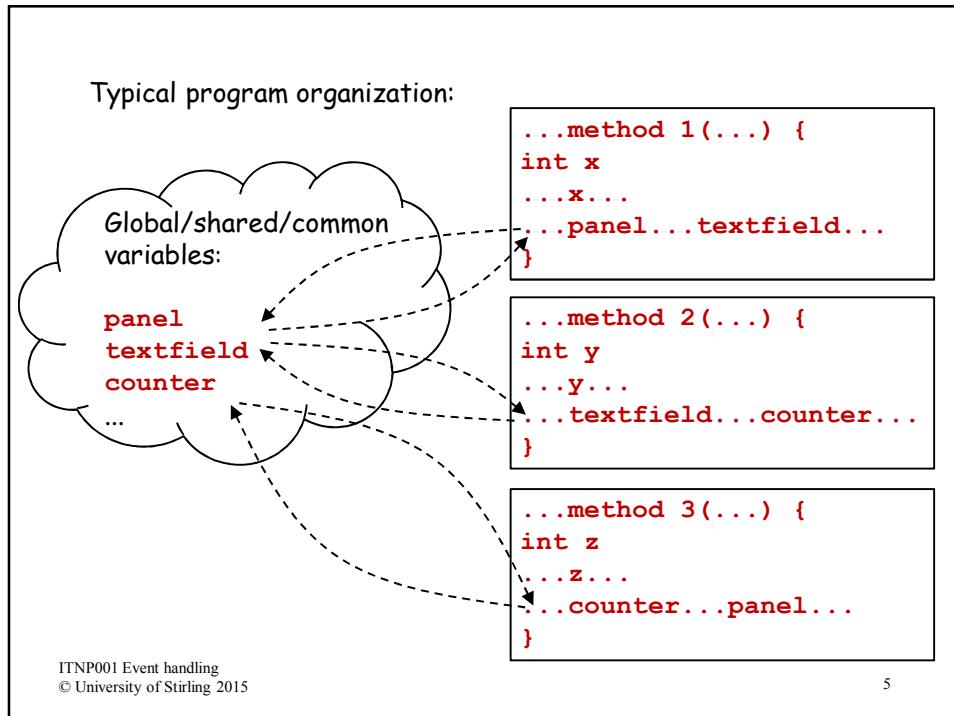
Where to store the count?

- The counter variable, `carCount`:
 - Must be declared
 - Must be set to 0 once as it is first allocated a memory slot
- We might think that the counter could be declared in the body of `actionPerformed`:


```
public void actionPerformed(ActionEvent event) {
    int carCount = 0;
    carCount = carCount + 1;
    JOptionPane.showMessageDialog(null,
        "Cars:" + carCount);
}
```
- But from the rules for local variables:
 - The memory is deallocated after each `actionPerformed`
 - And reallocated during the next call
 - So `carCount` will never get higher than 1!

Global variables

- Similarly, it is no use declaring `carCount` in any other method
- `carCount` needs to be *persistent*, it needs to outlive each method call
- Solution:
 - We can declare variables *outside the methods*
 - These are **global** variables - they can be directly referred to from inside *any* method (more properly called **instance** or **member** variables)
 - Their memory is allocated *as the program is started*
 - And is not deallocated *until the program ends*
 - If one method call assigns a new value to such a variable, then the *next* method call sees that new value
- A variable should be global if and only if
 - Its value should be persistent
 - And/or it must be referred to by more than one method



An alternative car counter, using a text field for output (not in JFS)

- The new car counter program looks like this:

```
imports ...
public class CarCounter2 ... {
    private int carCount = 0;
    private JTextField text;
    main ... createGUI ...
    public void actionPerformed(ActionEvent
        event) {
        carCount = carCount + 1;
        text.setText("Cars:" + carCount);
    }
}
```

Demo
CarCounter2

Note: Instructs the
text field to alter its
contents

ITNP001 Event handling
© University of Stirling 2015

7

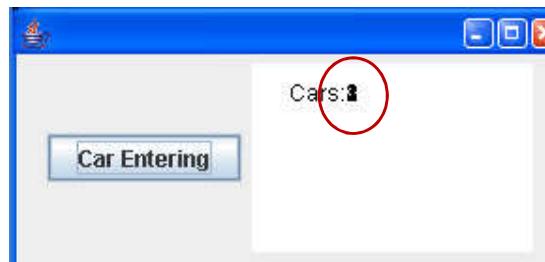
Another alternative, using graphics (not in JFS)

- The new car counter program looks like this:

```
public void actionPerformed(ActionEvent event) {
    carCount = carCount + 1;
    Graphics paper = panel.getGraphics();
    paper.drawString("Cars:" + carCount, 20, 20);
}
```

- BUT IT DOES NOT WORK CORRECTLY!

Demo
CarCounter3



ITNP001 Event handling
© University of Stirling 2015

8

- When graphics output must MODIFY/REPLACE what is on display already, then it must ERASE what is there first:

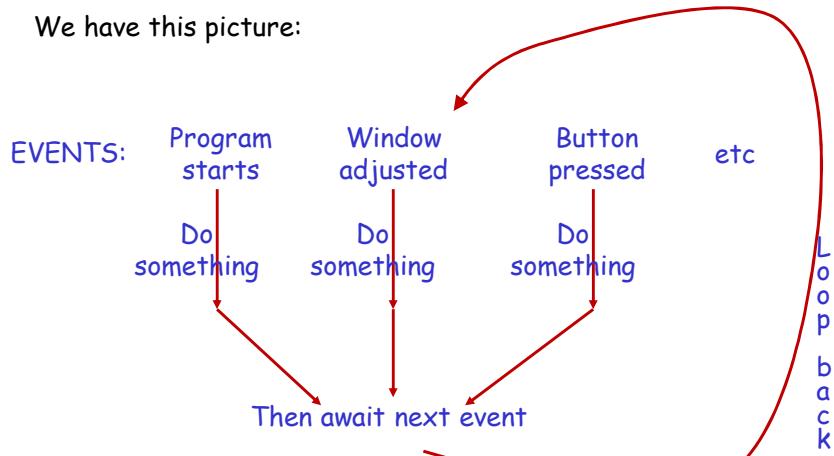
```
public void actionPerformed(ActionEvent event) {
    carCount = carCount + 1;
    Graphics paper = panel.getGraphics();
    paper.setColor(Color.white); // Erase
    paper.fillRect(0, 0, 150, 100);
    paper.setColor(Color.black); // Draw
    paper.drawString("Cars:" + carCount, 20, 20);
}
```

- Note: With care it may be possible to be more selective in choosing what is erased and re-drawn - but this is often too hard

Event-driven program execution

- The basic behaviour of most programs is
 - To start
 - To wait for an event,
 - And then to react to it
 - Then to wait for another event ...
 - And then to react to it
 - And so on (for ever!)
- This cycle is called "The Event Loop"
- Most applications are built like this. For example:
 - Microsoft Word
 - Windows itself
- Our Java programs work exactly like this

We have this picture:



Events and event-handlers

- In the event-driven approach we imagine sections of program activated by events
- Such events could be initiated by the user, interacting with the program (mouse or keyboard)
- But there are other kinds of events, too. For example:
 - Moving or re-sizing the program window
 - Exposing the program window by closing another
- Each "Do something" is called an "event-handler"
- Event-handlers are methods with specific names
 - These are called automatically by the JVM when a recognised event occurs
- Example:
 - **actionPerformed** is the event-handler for GUI button "presses"

- Most event-handlers have parameters that the JVM uses to convey details about the event *to the handler*
 - Example:


```
public void actionPerformed(ActionEvent event)
```
 - The parameter `event` stands for information about the button pressed
- A program that will be able to respond to certain kinds of events must indicate so at the start
 - For buttons we have, for example:


```
public class CarCounter extends JFrame
              implements ActionListener {
```
- And where the associated GUI “widget” is set up, the program must indicate that it wishes to be sent the relevant events:


```
button.addActionListener(this);
```
- Event-handler methods usually share information held in global variables

ITNP001 Event handling
© University of Stirling 2015

13

Introducing **JSlider** “widgets”: A simple application (not in JFS)

- Java Swing **JSlider** widgets allow the user to select a number in a set range by dragging a slider with the mouse
- The **SimpleSlider** application:



- The slider range is -200 to 2000 (in this example)
- The currently selected value is always displayed in a text field
- There is *one event* to consider: slider adjustment
 - The event handler is **stateChanged**
 - The action is to fetch the new setting and display in the text field

ITNP001 Event handling
© University of Stirling 2015

14

The SimpleSlider application

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
1. import javax.swing.event.*;
-----
public class SimpleSlider extends JFrame
2.         implements ChangeListener {
-----
3. private JLabel sliderLabel, displayLabel;
4. private JSlider slider;
5. private JTextField displayField;
-----
public static void main (String[] args) {
    SimpleSlider frame = new SimpleSlider();
    frame.setSize(400,100);
    frame.createGUI();
    frame.setVisible(true);
}

```

ITNP001 Event handling
© University of Stirling 2015

15

```

private void createGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container window = getContentPane();
    window.setLayout(new FlowLayout());
6.     sliderLabel =
        new JLabel("Adjust to select a number: ");
7.     window.add(sliderLabel);
-----
8.     slider = new JSlider(JSlider.HORIZONTAL,
                           -200, 2000, 1000);
9.     window.add(slider);
10.    slider.addChangeListener(this);
-----
11.    displayLabel =
        new JLabel("The number chosen is: ");
12.    window.add(displayLabel);
-----
13.    displayField = new JTextField("1000");
        window.add(displayField);
}

```

ITNP001 Event handling
© University of Stirling 2015

16

```

14. public void stateChanged(ChangeEvent e) {
15.     int selectedNumber = slider.getValue();
16.     displayField.setText(
17.         Integer.toString(selectedNumber));
18. }
19. // end of program

```

Key points

- Line 1: Indicates requirement for one more library
- Line 2: Indicates to the compiler/JVM that this program will be able to respond to slider changes
- Lines 3, 4, 5:
 - Global variables declared *outside all methods*:
 - **slider** and **displayField** will be referred to in **createGUI** and **stateChanged**
 - **sliderLabel** and **displayLabel** could be local within **createGUI**, but tidy to declare alongside other GUI items
 - They are **private**
 - The **JLabels** in line 3 will be used for the displayed messages
 - Line 4 does not actually *create* a slider: it just declares a variable called **slider** to hold a **JSlider**'s details
- The **main** method is standard

- Lines 6,7, 11, 12: Create and display the labels
 - A **JLabel** is a Swing GUI widget for displaying text
 - The JVM makes sure that the text is redrawn whenever necessary
 - No other interesting properties: no input, no events
 - **JLabel** is a library *class* describing the properties of a label
 - We create *actual instances* of library widgets using the keyword **new** in special "constructor expressions":


```
new JLabel("Adjust to select a number: ")
```
 - Then assign their details to suitable variables:


```
sliderLabel = new JLabel(".....");
```
 - The **JLabel** "constructor" has *one* parameter: the text to be displayed
 - Finally, the new label is added to the display:


```
window.add(sliderLabel);
```

- Similarly, **JSlider** is a Swing GUI widget
 - The JVM makes sure that the appearance is redrawn whenever necessary
 - Highly customizable
 - Causes a **ChangeEvent** when adjusted
 - **JSlider** is a library *class* describing the functions of a slider
 - We need to use the **JSlider** constructor to obtain an *actual slider*
- Line 8: Creates the slider and stores its details in **slider**
 - **new JSlider(...)** is a "constructor" with 4 parameters:
 - **JSlider.HORIZONTAL** (or **VERTICAL**)
 - Minimum, maximum and initial settings (-200, 2000, 1000 here)
- Line 9: Adds the slider to the program's display
- Line 10: Registers the program as wanting to be told about **slider**'s adjustment events

- Lines 14, 15, 16: Method `stateChanged`
 - Called by the JVM when a slider adjustment event occurs
- Line 15, in `stateChanged`:
 - `slider.getValue()` calls the slider's `getValue` method to obtain its current setting
 - `selectedNumber = ...` records the returned setting
- Line 16:
`displayField.setText(
 Integer.toString(selectedNumber));`
 instructs the text field to alter the text that it is displaying to the given message
 - `Integer.toString(selectedNumber)` is a library method call to convert an `int` to a `String` (the opposite of `parseInt`)

About `FlowLayout`

- When we `add` widgets, the JVM decides where they go
- The `createGUI` method sets "`FlowLayout`"
`window.setLayout(new FlowLayout());`
 - Widgets are placed centrally
 - From left to right
 - And in rows from the top
 - And *may be rearranged automatically if the window size changes*
- Lines 7, 9, 12, 13:
 - These determine the *order* in which the widgets appear in the window's layout:

1	2	3	4
---	---	---	---

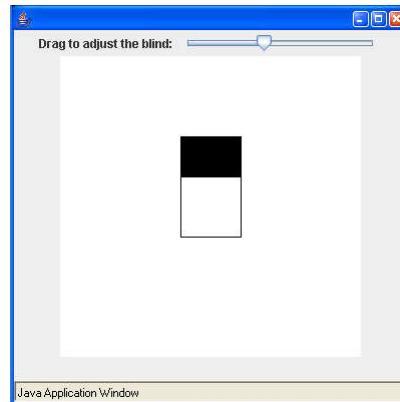
- You will learn about some other layout managers later

A **JSlider** application using graphics

WindowBlind (not in JFS)

Demo
WindowBlind

- The appearance:
 - One **JLabel** widget
 - One **JSlider** widget
Range 0-100
 - One **JPanel** for drawing
- One event to be handled:
 - **stateChanged**: Note the new setting, and draw the black blind an appropriate amount closed



The **WindowBlind** application

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
-----
public class WindowBlind extends JFrame
    implements ChangeListener {
-----
private JLabel sliderLabel;
private JSlider slider;
private JPanel panel;
-----
public static void main (String[] args) {
    WindowBlind frame = new WindowBlind();
    frame.setSize(400,400);
    frame.createGUI();
    frame.setVisible(true);
}
}
```

```

private void createGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container window = getContentPane();
    window.setLayout(new FlowLayout());
    -----
    sliderLabel =
        new JLabel("Drag to adjust the blind: ");
    window.add(sliderLabel);
    -----
    slider = new JSlider(JSlider.HORIZONTAL,
                         0, 100, 50);
    window.add(slider);
    slider.addChangeListener(this);
    -----
    panel = new JPanel();
    panel.setPreferredSize(
        new Dimension(300, 300));
    panel.setBackground(Color.white);
    window.add(panel);
}

}

```

ITNP001 Event handling
© University of Stirling 2015

25

```

public void stateChanged(ChangeEvent e) {
    Graphics paper = panel.getGraphics();
    int sliderValue = slider.getValue();
    paper.setColor(Color.white);      // Erase
    paper.fillRect(0, 0, 300, 300);
    paper.setColor(Color.black);      // Draw
    paper.drawRect(120, 80, 60, 100);
    paper.fillRect(120, 80, 60, sliderValue);
}
} // end of program

```

Note



ITNP001 Event handling
© University of Stirling 2015

26

End of section