Software Development 2

Models of Interaction

F27SB

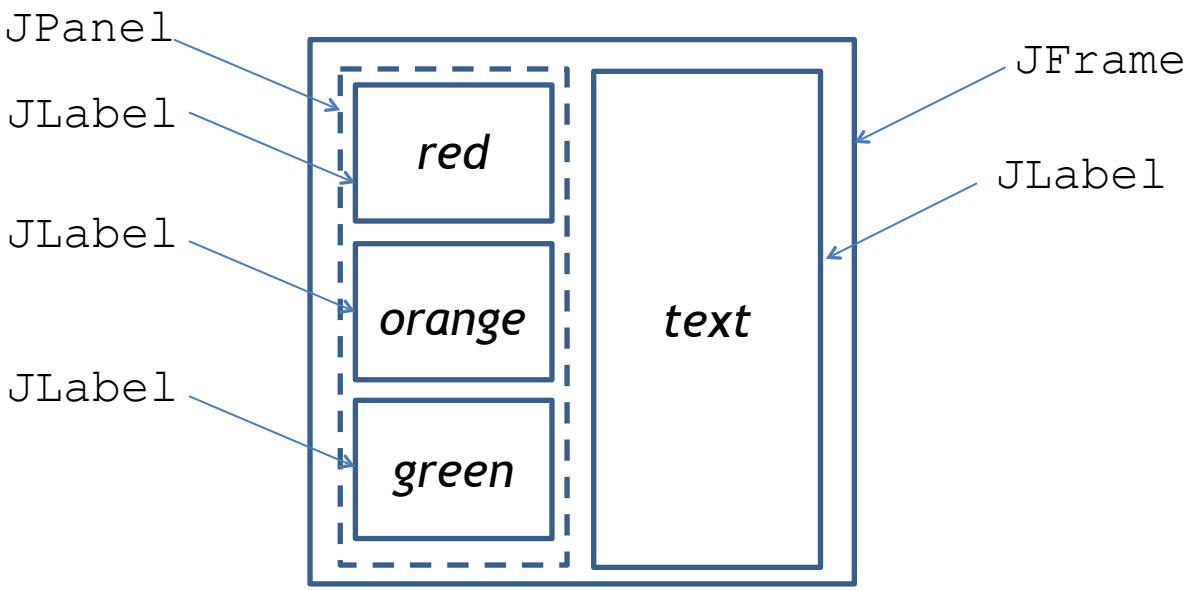
Previous Lecture

- `JLabel`s – how to display text
- `JPanel`s – containers for components
- Using layout managers to construct interfaces

Panel: example

e.g. add text message to traffic lights

- red ==> STOP; red/amber ==> READY; green ==> GO; amber ==> SLOW



Panel: example

```
JFrame => GridLayout => 1 row * 2 columns
JPanel => GridLayout => 3 rows * 1 column

...

class TLights extends JFrame
{  final int LIGHTNO = 3;
   JLabel [] lights = new JLabel[LIGHTNO];
   final long DELAY = 1000;

   JPanel display;
   JLabel text;
```

Panel: example

```
public TLights()
{   int i;

    setLayout(new GridLayout(1,2));

    lights[0] = new JLabel();
    lights[0].setBackground(Color.red);
    lights[1] = new JLabel();
    lights[1].setBackground(Color.white);
    lights[2] = new JLabel();
    lights[2].setBackground(Color.white);
```

Panel: example

```
    display = new JPanel(new GridLayout(3,1));
    for(i=0;i<LIGHTNO;i++)
    {  lights[i].setOpaque(true);
       display.add(lights[i]);
    }
    add(display);

    text = new JLabel("STOP",JLabel.CENTER);
    text.setFont
          (new Font("Serif",Font.BOLD,24));
    text.setBackground(Color.lightGray);
    text.setOpaque(true);
    add(text);
}
```

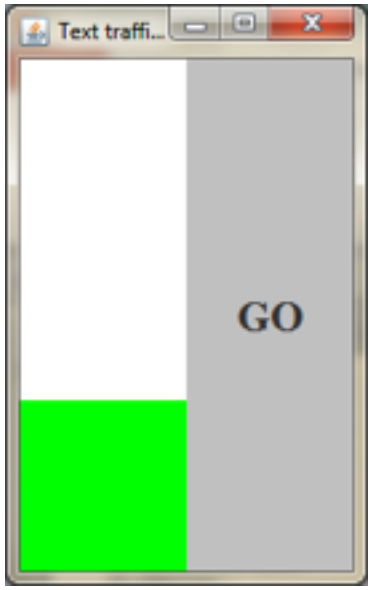
Panel: example

```
public void change()
{  while(true)
   {  pause(DELAY);
      lights[1].setBackground(Color.orange);
      text.setText("READY");
      pause(DELAY);
      lights[0].setBackground(Color.white);
      lights[1].setBackground(Color.white);
      lights[2].setBackground(Color.green);
      text.setText("GO");
      pause(DELAY);
      lights[1].setBackground(Color.orange);
      lights[2].setBackground(Color.white);
      text.setText("SLOW");
```

Panel: example

```
            pause(DELAY);
            lights[0].setBackground(Color.red);
            lights[1].setBackground(Color.white);
            text.setText("STOP");
        }
    }
}

class TestTLights
{   public static void main(String [] args)
    {   TLights tl = new TLights(); ... }
}
```

Panel: example

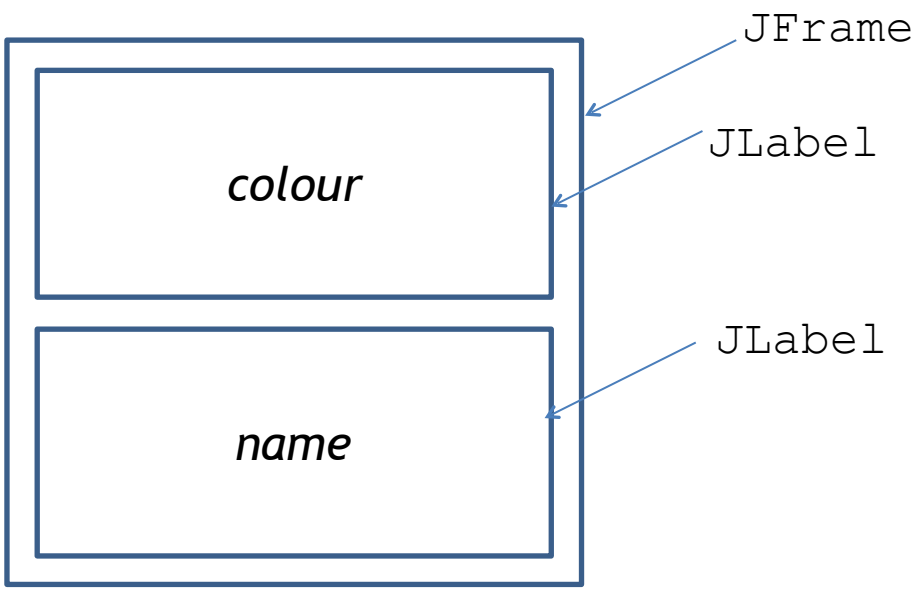


Window interaction with keyboard & display

- `JFrame` created by a program is in addition to and independent of screen/ keyboard window
- can still interact with program via display/keyboard
- use program interaction to change `JFrame`

Window interaction with keyboard & display

- e.g. repeatedly request/input name of colour via screen/keyboard, and display colour and name in `JFrame`



Window interaction with keyboard & display

```
class Colours extends JFrame
{  JLabel C,T;

   public Colours()
   {  C = new JLabel();
      C.setBackground(Color.white);
      C.setOpaque(true);
      T = new JLabel("",JLabel.CENTER);
      T.setFont(new Font("Serif",Font.BOLD,36));
      T.setBackground(Color.white);
      T.setOpaque(true);
      setLayout(new GridLayout(2,1));
      add(C);
      add(T);
   }
```

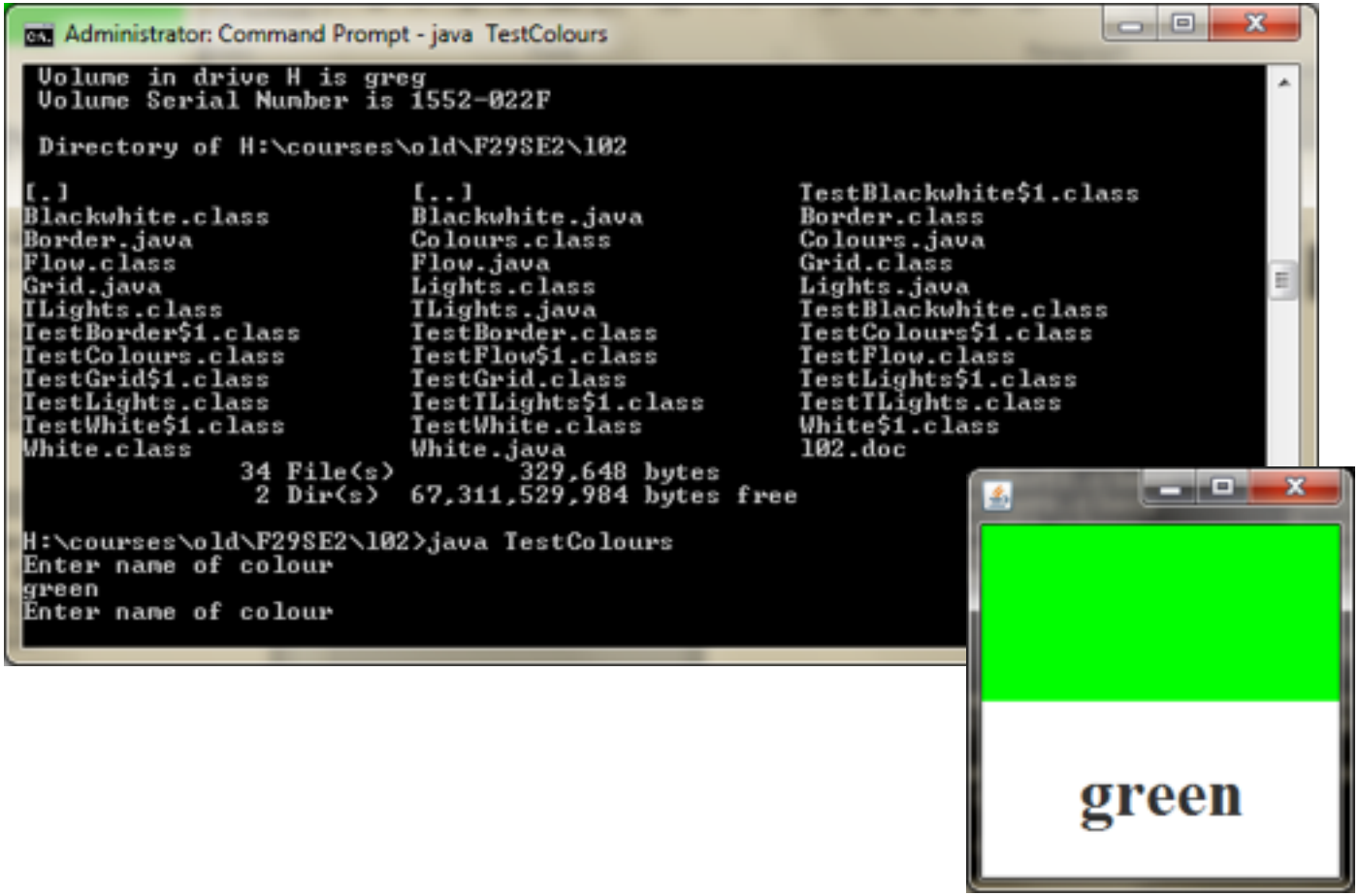
Window interaction with keyboard & display

```
public void setColour(String name)
{  Color colour = Color.white;
   if(name.equals("green"))
    colour=Color.green;
   else
   if(name.equals("red"))
    colour=Color.red;
   else
   if(name.equals("blue"))
    colour=Color.blue;
   else
    name="?"+name+"?";
   C.setBackground(colour);
   T.setText(name);
} }
```

Window interaction with keyboard & display

```
class TestColours
{  static BufferedReader keyboard =
    new BufferedReader
        (new InputStreamReader(System.in));
   static PrintWriter screen =
    new PrintWriter(System.out,true);

   public static void main(String [] args) throws IOException
   {  Colours c = new Colours();
      ...
      while(true)
      {  screen.println("Enter name of colour");
         c.setColour(keyboard.readLine());
      }
   } }
```

Window interaction with keyboard & display



Finally some buttons!!!111one

EVENT HANDLING

Today's Lecture

- Models of user interaction
- Java event handling
- `JButton`s

Models of interaction

Polling

- generalises simple prompt/input/process/output approach

- program
  - anticipates multiple input sources
  - e.g. keyboard; mouse
  - repeatedly checks each until one is ready
  - deals with source and continues

Models of interaction

- advantage
  - under user control
- disadvantages
  - need to decide when to poll
  - nothing else can be done while program is polling

Similar to busy waiting, remember the pause function.

Models of interaction

Interrupt

- originally a hardware concept

- interrupt may be:
  - external/hard
    - caused by something outside the CPU
    - e.g. I/O device, real-time clock
  - internal/soft
    - caused by a sub-program
    - e.g. operating system scheduler interrupting a process to let another run

Models of interaction

- When interrupt occurs:
  - CPU stops what it is doing
  - saves information about current state of program in dedicated memory area
    - i.e. values of variables and which instruction was being executed when interrupt occurred
  - runs code to handle interrupt

Models of interaction

- After interrupt:
  - information about current state of program restored from dedicated memory area
  - processing resumes at interrupted instruction

Models of interaction

- Java distinguishes:
  - *exception*
    - soft interrupt
    - program generated
  - *event*
    - hard interrupt
    - external activity
    - after event, program continues with current activity

Models of interaction

- GUI interaction starts with external activity ***raising*** event
  - e.g. key/mouse button pressed; mouse moved
- system
  - demands attention from program
- program
  - stops what it is doing
  - identifies source of event
  - ***handles*** event
  - continues with activity

Models of interaction

- advantage
  - does not waste time checking for external activity
  - if not reliant on external activity then do something else until event happens
- disadvantage
  - event will break flow of processing

Models of interaction

- How to handle event?
  - ignore it
  - note it and deal with it later
  - deal with it immediately

Java events

- general events are subclasses of `java.util.EventObject`
- GUI events from AWT
  - AWT events are subclasses of `java.awt.AWTEvent`
  - concrete AWT events collected in `java.awt.event`
  - e.g. from previous lecture: `java.awt.event.WindowEvent`

Java events

event object

- contains information about the event
- and methods for obtaining this information
  - e.g. `getWindow()` for `WindowEvent`

Java events

listener interface

- must be **implemented** to receive an event
  - E.g. `MyListener implements WindowListener`
- must be **registered** so that system knows which listener to execute when event is raised
  - E.g. JFrame's `addWindowListener()`

Java events

listener method

- must be implemented to handle a particular event
  - E.g. `windowClosing(WindowEvent e)`

Java events

source object

- the Java object that created the event
- `public Object getSource()`
  - this method exists for all event classes
  - returns the source object that caused the event
  - E.g. an object of class `JFrame`, `JButton`, etc.

Java events

```
class name implements event_class_listener_interface {
   ...
   object_class source_object1;
   public name() {
      source_object1
            .addevent_class_listener_interface(this);
   }
   ...
   listener_method(event_class var) {
      if(var.getSource()==source_object1) {
         // code to handle event from source_object1
      } else ...
   }
}
```

Java events

```
class name implements event_class_listener_interface {
   ...
   object_class source_object1;
   public name() {
      source_object1
            .addevent                                  his);
   }
   ...
   listener_method(event_class var) {
      if(var.getSource()==source_object1) {
         // code to handle event from source_object1
      } else ...
   }
}
```

Objects of *object_class* can generate events of *event_class*

Java events

```
class name implements event_class_listener_interface {
   ...
   object_class source_c
   public name() {
      source_object1
            .addevent_class_listener_interface(this);
   }
   ...
   listener_method(event_class var) {
      if(var.getSource()==source_object1) {
         // code to handle event from source_object1
      } else ...
   }
}
```

Its containing class knows how to handle events of *event_class*

Java events

```
class name implements event_class_listener_interface {
   ...
   object_class source_object1;
   public name() {
      source_object1
            .addevent_class_listener_interface(this);
   }
   ...
   listener_method(eve
      if(var.getSource()==source_object1) {
         // code to handle event from source_object1
      } else ...
   }
}
```

So we tell `source_object` that this is the case

Java events

```
class name implements event_class_listener_interface {
   ...
   object_class source_object1;
   public name() {
      sour
                                    er_interface(this);
   }
   ...
   listener_method(event_class var) {
      if(var.getSource()==source_object1) {
         // code to handle event from source_object1
      } else ...
   }
}
```

So, when `source_object` raises an event (e.g. by someone clicking on it), `listener_method` is called

Java events

```
class name implements event_class_listener_interface {
   ...
   object_class source_object1;
   public name() {
      source_object1
            .addevent_cla                                  is);
   }
   ...
   listener_method(event_class                  ) {
      if(var.getSource()==source_object1) {
         // code to handle event from source_object1
      } else ...
   }
}
```

listener_method then checks which object generated the event, and handles it accordingly

Déjà vu (ish)

```
class MyProgram implements WindowListener {
   ...
   JFrame frame = new JFrame();
   public MyProgram() {
      frame.addWindowListener(this);
   }
   ...
   windowClosing(WindowEvent var) {
      if(var.getSource()==frame) {
         // clean up any state associated with window
      } else ...
   }
}
```

Java events

The system will postpone other events until the current one has been handled

- nothing further can happen until exit from *listener method*

- exit to where?
  - `main` has often completed at this point
  - so, nothing happens until next event...
- this is known as an ***event driven system***

Java events

- typical flow of execution for a GUI program:
  - system calls main
  - main
    - calls constructor(s) to set up user interface
    - terminates
  - nothing happens until...
  - user causes event
    - system calls listener method
    - listener method handles event & terminates

Using event handlers

BUTTONS

Buttons

- Icon for selection by mouse

```
public JButton(String s)
    implements JComponent
```

- *s* is the text to be written on the button
- to change text:
  `setText(String text)`

Buttons

`ActionEvent`

- event class for buttons
- raised when `JButton` is selected

`ActionListener`

- listener interface

`actionPerformed`

- listener method

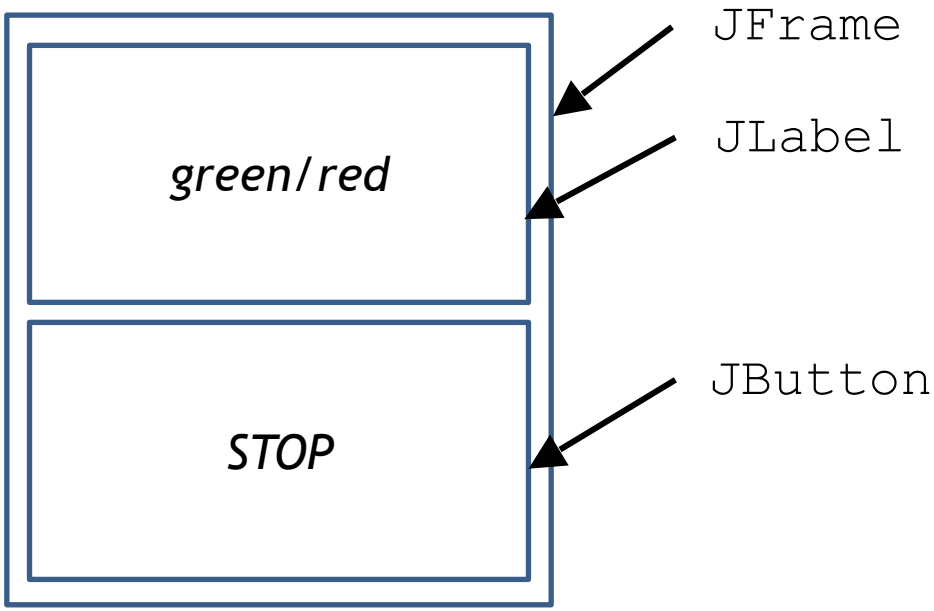
Buttons

```
class MyProgram implements ActionListener {
   ...
   JButton button = new JButton("Press me!");
   public MyProgram() {
      button.addActionListener(this);
   }
   ...
   public void actionPerformed(ActionEvent var) {
      if(var.getSource()==button) {
         // run the appropriate code
      } else ...
   }
}
```

Buttons

```
class MyProgram extends JFrame implements ActionListener
{  ...
   JButton button = new JButton("Press me!");
   public MyProgram() {
      button.addActionListener(this);
   }
   ...
   public void actionPerformed(ActionEvent var) {
      if(var.getSource()==button) {
         // run the appropriate code
      } else ...
   }
}
```

Example: stop sign



- `JLabel` is initially green
- select *STOP* `JButton` to change `JLabel` to red

Example: stop sign

```
...
class Stopsign
 extends JFrame implements ActionListener
{   JLabel l;
    JButton b;

    public Stopsign()
    {   setLayout(new GridLayout(2,1));

        l = new JLabel();
        l.setBackground(Color.green);
        l.setOpaque(true);
        add(l);
```

set up `Jlabel l`'s look - initially green

Example: stop sign

```
    b = new JButton("STOP");
    b.setFont(new
        Font("Serif",Font.ITALIC,36));
    b.setBackground(Color.white);
    b.setOpaque(true);
    add(b);
    b.addActionListener(this);
}
```

sets up `Jbutton` `b`'s look

add action listener for `b` to this `JFrame`

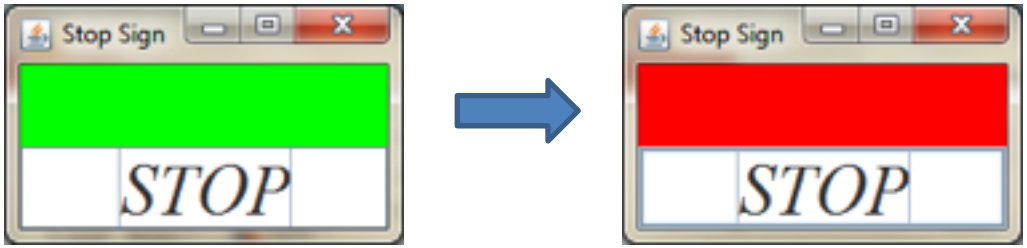
Example: stop sign

```
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==b)
            l.setBackground(Color.red);
    }
}
```

- defines listener method `actionPerformed`
  - if source of `ActionEvent` is `JButton b`
  - change `JLabel l`'s colour to red

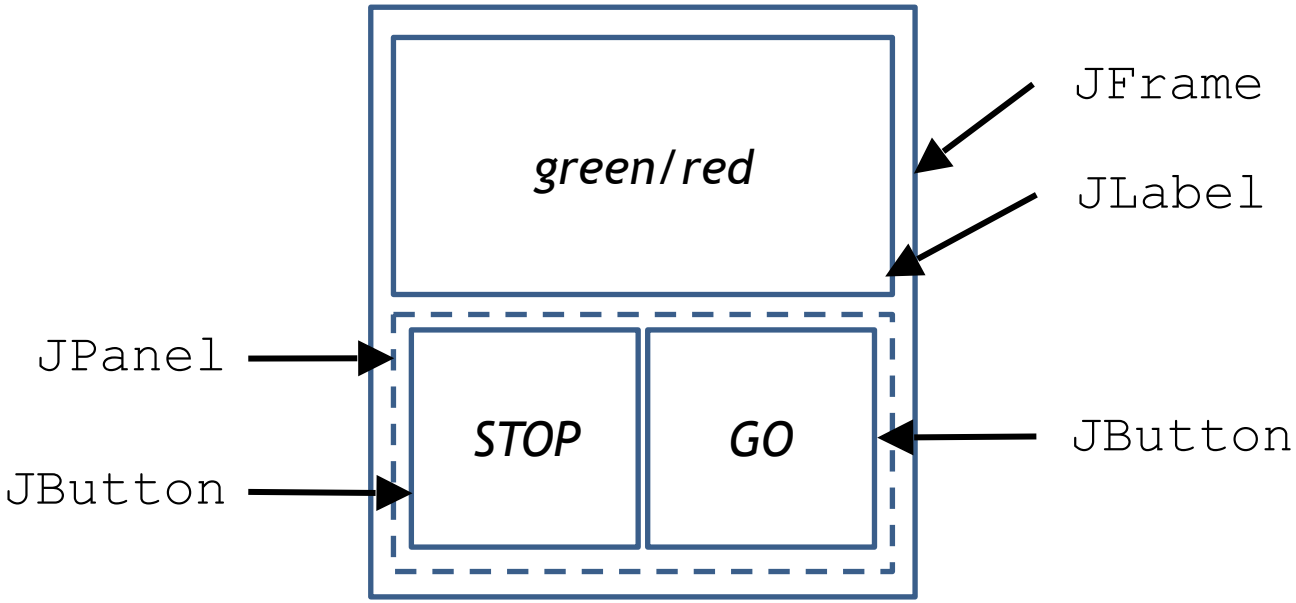
Example: stop sign

```
class TestStopsign {
   public static void main(String [] args){
      StopSign sign = new StopSign()
      ...
   }
}
```



Example: stop/go 1

- every time `b` is pressed, colour is set to red…
- also need a *GO* button to change back to green



Example: stop/go 1

```
...
class Stopgo
 extends JFrame implements ActionListener
{  JLabel l;
   JButton sb,gb;
   JPanel p;

   public Stopgo()
   {  setLayout(new GridLayout(2,1));

      l = new JLabel();
      l.setBackground(Color.green);
      l.setOpaque(true);
      add(l);
```

Example: stop/go 1

```
p = new JPanel(new GridLayout(1,2));

Font f = new Font("Serif",Font.ITALIC,18);

sb = new JButton("STOP");
sb.setFont(f);
sb.setBackground(Color.white);
sb.setOpaque(true);
p.add(sb);
sb.addActionListener(this);
```

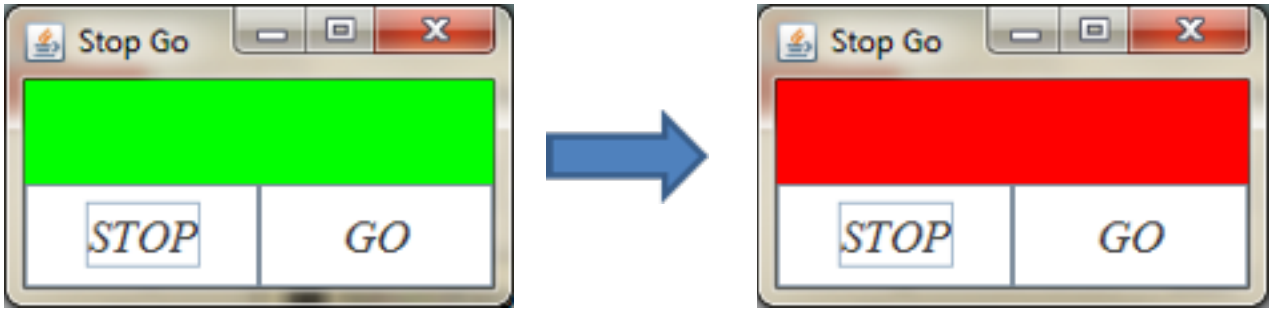
Example: stop/go 1

```
    gb = new JButton("GO");
    gb.setFont(f);
    gb.setBackground(Color.white);
    gb.setOpaque(true);
    p.add(gb);
    gb.addActionListener(this);

    add(p);
}
```

Example: stop/go 1

```
   public void actionPerformed(ActionEvent e)
   {  if(e.getSource()==sb)
         l.setBackground(Color.red);
      else
         if(e.getSource()==gb)
            l.setBackground(Color.green);
   }
}

class TestStopgo
{  public static void main(String [] args)
   {  ... }
}
```

Example: stop/go 1



Example: stop/go 2

- can still
  - press *GO* when light is green
  - press *STOP* when light is red
- good interface design practice to restrict interaction options to those that apply at each stage
- this minimises
  - user confusion over what is possible next
  - system errors through inappropriate action sequences

Example: stop/go 2

- change visibility of JButtons?

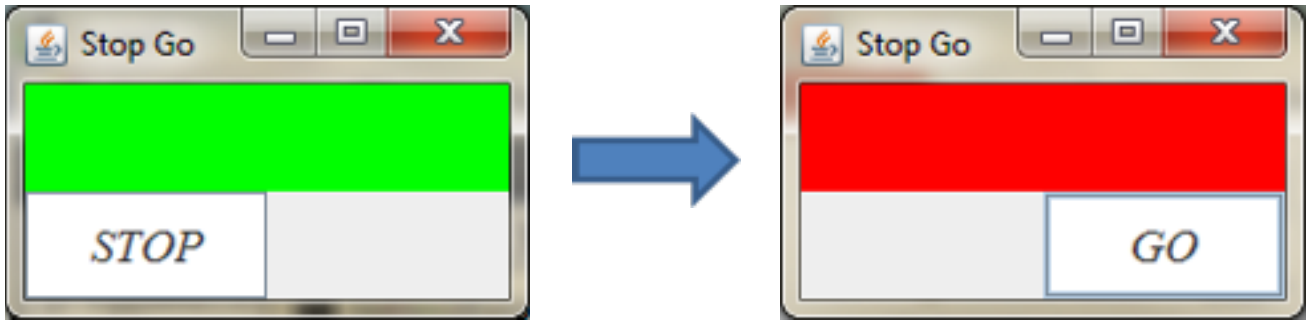
```
gb = new JButton("GO");
gb.setFont(f);
gb.setBackground(Color.white);
gb.setOpaque(true);
gb.setVisible(false);
p.add(gb);
gb.addActionListener(this);
```

Example: stop/go 2

- change visibility of JButtons?

```
public void actionPerformed(ActionEvent e {
    if(e.getSource()==sb) {
        l.setBackground(Color.red);
        sb.setVisible(false);
        gb.setVisible(true);
    } else if(e.getSource()==gb {
        l.setBackground(Color.green);
        gb.setVisible(false);
        sb.setVisible(true);
    }
}
```

Example: stop/go 2



- user can't see alternatives
  - which might make it hard to understand the system's functionality

Example: stop/go 2

`setEnabled(boolean enabled)`

- *enable/disable* `Component`s
- `Component`s remain visible
- `setEnabled(true)`
  - `Component` is active
- `setEnabled(false)`
  - `Component` is passive

Example: stop/go 2

- disabled `JButton`
  - will not react to selection
  - changes colour but still present
- *GO* selected
  - set *GO* `JButton` non-enabled
  - set *STOP* `JButton` enabled
- *STOP* selected
  - set *STOP* `JButton` non-enabled
  - set *GO* `JButton` enabled

Example: stop/go 2

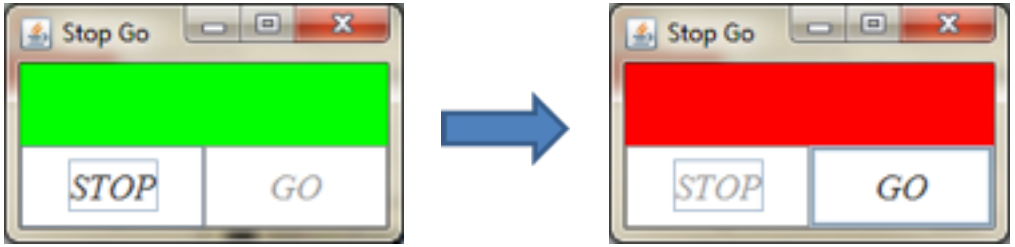
```
...
class Stopgo2
 extends JFrame implements ActionListener
{  JLabel l;
   JButton sb,gb;
   JPanel p;

   public Stopgo2()
   {  ...  }
```

Example: stop/go 2

```
   public void actionPerformed(ActionEvent e)
   {  if(e.getSource()==sb)
      {  l.setBackground(Color.red);
         sb.setEnabled(false);
         gb.setEnabled(true);
      }
      else
      if(e.getSource()==gb)
      {  l.setBackground(Color.green);
         gb.setEnabled(false);
         sb.setEnabled(true);
      }
   }
}
```

Example: stop/go 2

```
class TestStopgo2
{   ... }
```



Example: stop/go 3

- do we need both `JButtons`?
- revert to one `JButton`
- at each stage remember whether light is red/stop or green/go
  - *light state*

Example: stop/go 3

- when `JButton` is selected
  - identify current state
  - change `JLabel` colour
    - red ==> green; green ==> red
  - change text on `JButton`
  - *GO* ==> *STOP*; *STOP* ==> *GO*
  - remember new light state
- *state variable*
  - only 2 possibilities
  - Use boolean variable

Example: stop/go 3

```
...
class Stopgo3
 extends JFrame implements ActionListener
{  JLabel l;
   JButton sg;
   boolean isStopped = false;
```

- `isStopped` ==> *state variable*
  - `true` == stopped/red
  - `false` == not stopped/green

Example: stop/go 3

```
public Stopgo3()
{   setLayout(new GridLayout(2,1));
    Font f = new Font("Serif",Font.ITALIC,18);
    l = new JLabel();
    l.setBackground(Color.green);
    l.setOpaque(true);
    add(l);
    sg = new JButton("STOP");
    sg.setFont(f);
    sg.setBackground(Color.white);
    sg.setOpaque(true);
    sg.addActionListener(this);
    add(sg);
}
```

Example: stop/go 3

```
    public void actionPerformed(ActionEvent e){
        if(e.getSource()==sg)
            if(isStopped) {
                l.setBackground(Color.green);
                isStopped=false;
                sg.setText("STOP");
            } else {
                l.setBackground(Color.red);
                isStopped = true;
                sg.setText("GO");
            }
    }
}

class TestStopgo3{  ... }
```

Example: stop/go 3

THAT’S IT

Next Lecture

- GUI program structure
- Editable text
- Dynamic interfaces