Software Development 2

GUI Revision

F27SB

GENERAL GUIDELINES

User Interface Design

- "Form follows function"
  - A GUI's appearance should indicate/reflect a program's functionality
  - A control's appearance should suggest what effect it has when selected

User Interface Design

- “User illusion”
  - Users focus on intention behind system
  - They are not aware of the distinction between hardware, software, OS, GUI and program
  - Interfaces should be designed to promote and maintain this illusion

User Interface Design

- The importance of separating interface code from implementation code
  - may want to change implementation without changing interface
  - may want to change interface without changing implementation

FRAMES AND LAYOUTS

JFrame

```
public class MyFrame extends JFrame {
      public MyFrame() {
            //perform initialisation
            //add contents to the frame
      }
}
```

```
public class TestMyFrame {
      public static void main(String[] args) {
            MyFrame frame = new MyFrame();
            //address global concerns
            //such as size and location of GUI
      }
}
```

Events and listeners

- To handle events when someone presses one of the buttons in the title bar:
  - import AWT Java event classes

```
import java.awt.event.*;
```

  - tell Java we have a listener for window events, using the `JFrame` method:

```
public void
    addWindowListener(WindowListener l)
```

  - build a `WindowListener` to respond to events

Events and listeners

```
public interface WindowListener extends EventListener
{
   public void windowActivated(WindowEvent e);
   public void windowClosed(WindowEvent e);
   public void windowClosing(WindowEvent e);
   public void windowDeactivated(WindowEvent e);
   public void windowDeiconified(WindowEvent e);
   public void windowIconified(WindowEvent e);
   public void windowOpened(WindowEvent e);
}
```

Events and listeners

- all methods in `WindowListener` are abstract - interface
- our class must
  - implement `WindowListener`
  - provide details for all methods
- we could implement details of all the interface methods ourselves...

Events and listeners

- but easier (and more concise) to:
  - subclass a supplied implementation of the interface
  - override methods we are concerned about

```
public class WindowAdapter
                implements WindowListener
```

- provides empty versions of all above methods
  - so subclass `WindowAdapter` and only override the methods that should do something

Alternatively

```
import java.awt.event.*;

public class JavaGUI {
    public static void main(String[] args) {
        MyFrame frame = new MyFrame();
        …
        frame.addWindowListener(
            new WindowAdapter(){
                public void windowClosing(WindowEvent e){
                    System.exit(0);
                }
            });
    }
}
```

This is an ‘anonymous class’, a way of reducing code when you only need to create a single instance of a class – it is commonly used to create event listeners.

Manipulating JFrame

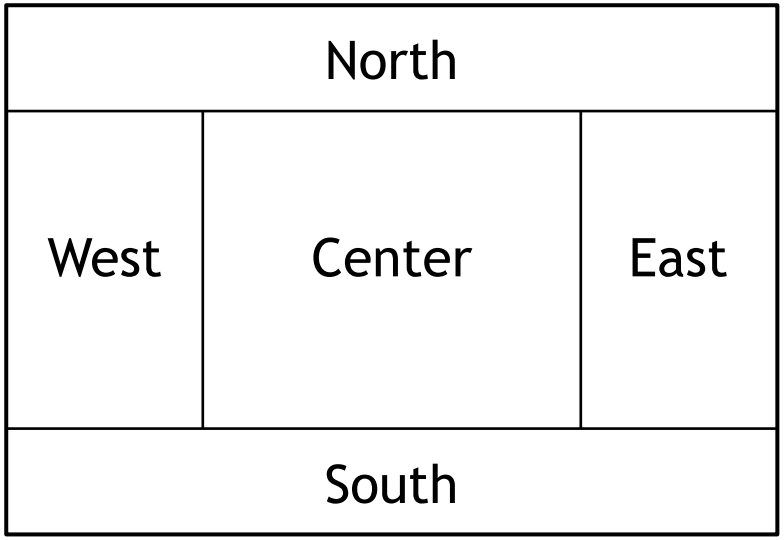
- Have to use the content pane
  - `getContentPane();`
- Some methods work implicitly on the content pane like `add()` and `setLayout()`
  - If in doubt, just use the content pane:

```
getContentPane().add(Container c);
```

Border layout

```
public class BorderLayout
             implements LayoutManager2
```

- holds `Component`s in five locations:



Border layout

```
public BorderLayout()
public Component add(Component comp,
                     int name)
```

- add *comp* to location identified by *name* from `SwingConstants`:
  - `CENTER, EAST, NORTH, WEST, SOUTH`
- added `Component` will fill area
- area sizes adjusted so that all added `Component`s fit
- `BorderLayout` is default for `JFrame`

Grid layout

`public class GridLayout`
`public GridLayout(`*`int rows`*`, `*`int cols`*`)`

- `Container` has *`rows`* rows of *`cols`* columns
- all regions are the same size

`public Component add(`*`Component comp`*`)`

- *`comp`* sized to fit exactly in area
- `Component`s added in order
  - left to right, then top to bottom

Flow layout

`public class FlowLayout`

`public Component add(`*`Component comp`*`)`

- places `Component`s
  - left to right
  - top to bottom
  - does not use a grid
- makes `Component`s as small as is compatible with contents

Mixing layouts

- Use JPanel to have different layouts in your window.
  - Each panel can have a different layout from each other and from the JFrame
  - Grouping components before adding them to the JFrame.

SIMPLE ELEMENTS

Labels and text

`public JLabel(`*`String text, int alignment`*`)`

- alignment = position of text in JLabel
- use `JLabel` constants to specify this:
  - `JLabel.CENTER`
  - `JLabel.LEFT`
  - `JLabel.RIGHT`
- Provided methods:
  - `setText(String text), getText()`
  - `setFont(Font f)`
  - `setBackground(Color c), setForeground(Color c),`
  - `setOpaque(boolean b)`

Labels and text

Can change font by providing a `Font` object:

```
public class Font
public Font(String name, int style, int size)
```

- font names are strings, e.g.
  - "Serif" - uses a default serif font
  - "Sansserif" - uses a sans serif font
  - "Monospaced" - uses a monospaced font
  - "Arial Black" - **uses the specified font**

Labels and text

Font style is specified using constants in `Font`:

`Font.PLAIN`

- this is plain serif

`Font.BOLD`

- **this is bold sanserif**

`Font.ITALIC`

- *`this is italic monospaced`*

`Font.BOLD|Font.ITALIC`

- ***this is bold and italic serif***

Labels and text

```
JLabel l = new JLabel(“Text”, JLabel.CENTER);
l.setFont(new Font(“Arial”,Font.ITALIC,24));
l.setBackground(Color.GREEN);
l.setOpaque(true);
```

INTERACTION

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

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

- 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

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 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 ...
   }
}
```

PROGRAM STRUCTURE

Guidlines

- Follow OOP principles
  - The easier it is to maintain
- Have your program inherit from `JFrame` if its main functionality is to show a window on the screen
  - If it has action sources, e.g. buttons, have it implement `ActionListener`
  - This increases cohesion by having the class handle all its own events and functionality
  - Reduces coupling by not having an external class as action listener

Guidlines

- Reduce code duplication
  - When creating a lot of labels and buttons that should have the same look, consider creating a method to do this or create a new class that inherits from JButton/JLabel.
- Use a method if you need the component only in that class
  - Reduces coupling
- Use a class if you need the component throughout a larger project and in several classes
  - Increases coupling but reduces code duplication

Guidlines

```
class MyFrame extends JFrame implements ActionListener {
    JButton b;
    …
    public MyFrame() {
       …
       b = setupButton(“Click me”);
       …
    }
    private JButton setupButton(String s) {
       JButton b = new JButton(s);
       …
       return b;
    }
}
```

Guidlines

```
class MyButton extends JButton {
    public MyButton(String s) {
       super(s);
       …
    }
}
class MyFrame extends JFrame implements ActionListener {
    JButton b;
    …
    public MyFrame() {
       …
       b = new MyButton(“Click me”);
       …
    }
}
```

DYNAMIC INTERFACE CHANGES

Dynamic interface changes

- Replace parts of the GUI with new elements.
  - Remove the old part:
    - `remove(Container c);`
    - `c.setVisible(false);`
  - Add the new part:
    - `add(Container c);`
    - `c.setVisible(true);`
- Need to set visibility of JFrame to trigger redraw: `setVisible(true);`

Dynamic interface changes

```
class MyFrame extends JFrame {
    …
    JButton b1,b2;
    …
    // remove button b1
    remove(b1); b1.setVisible(false);
    // add button b2
    add(b2); b2.setVisible(true);
    // set visibility of JFrame to redraw
    setVisible(true);
}
```

STATE DIAGRAMS

State and Interaction

- An interactive system can be seen as a sequence of *events* and associated *actions*
  - ***Event***: something that indicates the need for change, e.g. selecting a button
  - ***Action***: something that causes change, e.g. a listener method is invoked to respond to event

State and Interaction

- It is usually possible to identify distinct ***states***
- Where a state can be characterised as:
  - **a configuration**: the status of things that may change, such as variables and components
  - **valid events/actions** for current configuration
    - i.e. how change is indicated/what may be changed and how

State and Interaction

It is useful to think of the entire system as a set of states. System execution can then be seen as a series of ***state transitions***:

- Starting from a particular state
  1. event occurs
  2. action is triggered
  3. configuration is changed
  4. new state is entered

State Transition Diagrams

Depict the states and transitions in a system

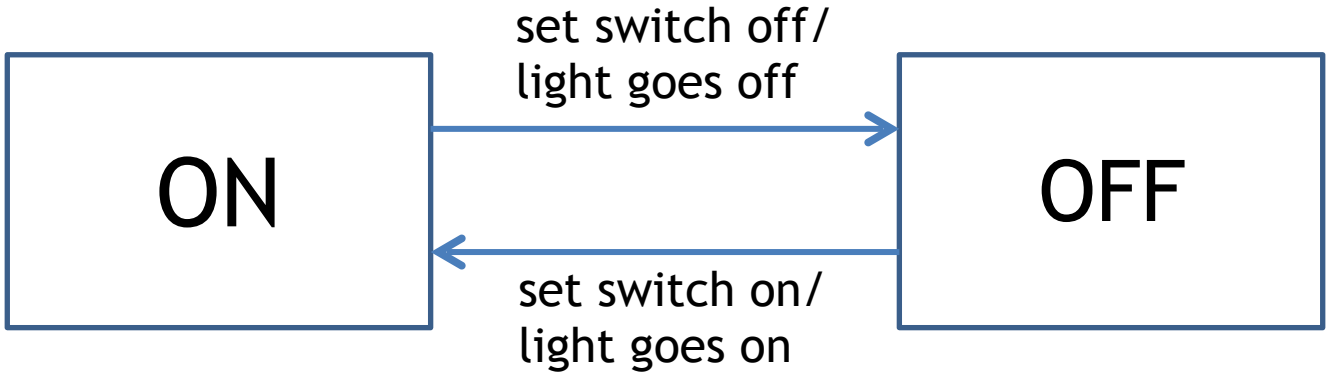
- **state**:
  - box with its name in it
- **transition**:
  - *arc* from a state to a state
  - labelled with "***event*** / ***action***"

Can attach a guard to a state: [*guard*]

- *guard* must be true for transition to occur

State Transition Diagrams

A simple example: a light circuit



PERSPECTIVES

Things To Consider: Perspectives

The user's view of a system's states is not necessarily the same as a developer's view:

- the user interprets system behaviour in terms of their conceptual model of what system is for
- the developer knows about, and thinks about, underlying programming constructs
- e.g. to use editor, user doesn't need to know how interface or file system are implemented

Things To Consider: Perspectives

A developer typically thinks of state in terms of programming constructs:

- current configuration: variable values; open files; `JFrame` `Component`s
- events: Java `Event`s
- actions: methods

- e.g. editor: developer thinks about character positions in arrays of `String`s, file I/O etc.

Form Follows Function

For effective system use, the user always needs to know what state the system is in

- *hidden mode*: in some state but no way to tell which one — avoid hidden modes!
- ensure user always knows current system state
  - unambiguous display content
  - explicit statement of mode
  - e.g. MS Word always indicates current style, font, etc.

FURTHER SWING

Editable text

`JTextArea extends javax.swing.text.JTextComponent`

- multiple lines of plain, unformatted text
- user can edit text

`JTextArea(String` *`text`*`, int` *`rows`*`, int` *`columns`*`)`

- *`text`*: text to be edited
- *`rows`*: number of displayed rows
- *`columns`*: number of displayed columns

Editable text

`public String setText(String text)`

- replaces all text with *`text`*

`public String append(String text)`

- adds *`text`* at end
- manually a line break: `append(text+"\n")`

`public String getText()`

- returns text as single `String`

`public void setEditable(boolean b)`

- makes the `TextArea` (non)interactive.

Scrolling

However, text doesn't scroll, so also need a…

```
JScrollPane extends JComponent
   implements Accessible, ScrollPaneConstants

public JScrollPane()
public JScrollPane(Component view)
```

- constructs a `JScrollPane` container for `Component` *`view`*
- allows horizontal & vertical scrolling

Scrolling

Can change presence of scroll bars with:

`setHorizontalScrollBarPolicy(int` *`policy`*`)`

`setVerticalScrollBarPolicy(int` *`policy`*`)`

- where *policy* is a `ScrollPaneConstants`:
  - `HORIZONTAL_SCROLLBAR_` or
  - `VERTICAL_SCROLLBAR_` followed by:
  - `ALWAYS` `AS_NEEDED` or `NEVER`
- e.g. `setHorizontalScrollBarPolicy(`
  `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER)`

Scrolling

Add a `Component` to a scroll pane using:

- `public void setViewportView(Component view)`
- Don't use `add()`
  - This will compile, but won't work as expected!

Menus

There's a Swing class for menu bars:

```
JMenuBar extends JComponent
      implements MenuElement
```

`JMenuBar()`

- creates a new `JMenuBar`

`setJMenuBar(JMenuBar` *`menubar`*`)`

- places *`menubar`* at the top of a `JFrame`

Menus

Also a Swing class for menus:

```
JMenu extends JMenuItem
      implements MenuElement
```

`JMenu(String s)`

- creates a new `JMenu` identified by `s`

`jmenubar``.add(`*`jmenu`*`)`

- adds *`jmenu`* to *`jmenubar`*

Menus

And a Swing class for menu items:

```
JMenuItem extends AbstractButton
      implements MenuElement
```

`JMenuItem(String s)`

- creates a new `JMenu` identified by `s`

`jmenu``.add(`*`jmenuitem`*`)`

- Adds menu item *`jmenuitem`* to *`jmenu`*

`jmenu``.add(`*`submenu`*`)`

- Adds menu *`submenu`* to *`jmenu`*

Menus

```
…
JMenuBar jb; // the menu bar
JMenu file; // the file menu
JMenuItem MNew, MOpen, MClose, MExit; // the menu items
…
jb = new JMenuBar(); // create menu bar
file = new JMenu("File"); // create named menu
// create menu items, specifying names
MNew = new JMenuItem("New");
MOpen = new JMenuItem("Open");
MClose = new JMenuItem("Close");
MExit = new JMenuItem("Exit");

// add menu items to file menu and add listeners
file.add(MNew); MNew.addActionListener(this);
file.add(MOpen); MOpen.addActionListener(this);
file.add(MClose); MClose.addActionListener(this);
file.add(MExit); MExit.addActionListener(this);

jb.add(file); // add file menu to menu bar

setJMenuBar(jb); // display menu bar (in current window)
```

File chooser

We don't want to hard code the file paths, so use:

```
JFileChooser extends JComponent
             implements Accessible
```

- a Swing component providing a standard GUI for file system navigation

`JFileChooser()`

- create `JFileChooser` component for current directory
- Does **not** support `WindowListener`

File chooser

```
int showOpenDialog(Component parent)
int showSaveDialog(Component parent)
```

- both return constants:
  - `APPROVE_OPTION` for *Open*/*Save* buttons
  - `CANCEL_OPTION` for *Cancel* button

```
File getSelectedFile()
```

- returns selected file from `JFileChooser`
- The `parent` can be `null` or the component that should be blocked while the chooser is open.

Dialogs

These dialogs can be created using:

```
JOptionPane extends JComponent
            implements Accessible
```

`JOptionPane()`

- constructor creates new empty dialog

However `new JOptionPane()` is rarely used

- instead, use static methods
- *`showXxxDialog()`*

Dialogs

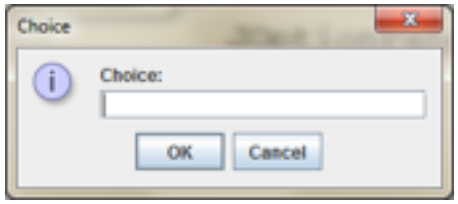
- Confirm dialog:


```
public static int showConfirmDialog
( Component parent, Object message,
  [String title, int optiontype,
  int messagetype, Icon icon])
```

  - *optiontype*: `DEFAULT_OPTION (Okay), YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION`
  - returns: `OK_OPTION, CANCEL_OPTION, YES_OPTION, NO_OPTION, CLOSED_OPTION` (window closed without selection)

Dialogs

- Input dialog:

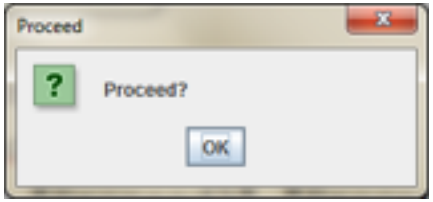



```
public static String showInputDialog
  (Component parent, Object message,
    [String title, int messagetype, ...])
```

- returns entered `String`

Dialogs

Message dialog:




```
public static void showMessageDialog
(Component parent, Object message,
 [String title, int messagetype, Icon icon])
```

- *parent*: usually a frame, can be `null`
- *message*: string to display
- other arguments are optional
  - *messagetype*: `WARNING_MESSAGE, QUESTION_MESSAGE, INFORMATION_MESSAGE, ERROR_MESSAGE, PLAIN_MESSAGE`
  - *icon*: icon to display with message
- Supports `WindowListener`

THAT'S IT!