CSC 8002
Advanced Programming

Dr Marta Koutny
School of Computing,
Newcastle University,
marta.koutny@ncl.ac.uk

Event Driven Programming

- The flow of control in a sequential program is the order in which statements are executed
- In traditional sequential programming, the programmer determines the flow of control
- In event driven programming, the flow of control is determined by external events
- The program responds to each event in a predictable fashion, but has no control over the order in which events occur

Examples of event driven programming

- Event driven programming is used widely to construct graphical user interfaces
- A GUI program must respond to events caused by the user's actions
- But in general, the source of events could be another program, a network or a device, or the operating system itself

Event generation and detection

- The execution of an event driven program is controlled by some external software
- The external software is responsible for detecting events and reporting them to the event driven program
- Typically, events are generated by some external source that is monitored by the external software

Event dispatching

- The process of reporting an event to an event driven program is called *event dispatching*
- Typically, event dispatching is performed by an *event loop*:

```
while (true)
{
   wait for next event
   report event to event driven program
}
```


Event handling

- The code that an event driven program executes when it receives an event is called an *event handler*
- Thus, event dispatching involves calling an event handler
- Typically, the event is passed as an argument to the event handler

Event handlers

- An event driven program can have multiple event handlers
- Each event handler is responsible for handling a particular set of events, e.g.
  - All the events from a particular source
  - All the events of a particular kind
- The relationship between events and event handlers is part of the design of an event driven program

Registering event handlers

- The event dispatching software needs to know which event handler to call for each event
- Thus, an event driven program must register its event handlers with the event dispatching software
- This is typically done during an initialisation phase before control is passed to the event dispatching software

Execution of an event driven program

1. Declare appropriate event handlers
2. Register event handlers with event dispatching mechanism
3. Pass control to event dispatcher and wait for events to occur
4. When event occurs, event dispatcher calls the appropriate event handler
5. Event handler executes and then returns control to event dispatcher

How does event handling work?

- An event handler is just a method that gets called by an event dispatcher
- Registering an event handler involves telling the event dispatcher about the object that implements the event handling method
- When the event occurs, the event dispatcher calls the event handler
- This is sometimes known as a *call-back*

Threading issues

- It is quite common for the event dispatching loop to run in its own thread
- Thus, event handlers will be executed in a different thread from the main program
- If the main program is still active during the event handling phase, some form of synchronisation may be required
- However, there is no need for synchronisation between event handlers

Event dispatching threads

- Note that if the event dispatcher runs in its own thread, there is no need to pass control to the event loop explicitly
- Indeed, once the handlers have been set up and the event dispatching thread has started, the main program can terminate
- The event dispatching thread will continue to run, dispatching events to event handlers

A simplified implementation

- We now outline a possible implementation of event handling in Java
- We need two interfaces:
  - `EventHandler`
  - `EventDispatcher`
- An event handling object can register itself with an event dispatching object
- The event dispatcher will call the event handler when an event occurs

Event handling interfaces

```
interface EventHandler
{
  void handleEvent(Event event);
}

interface EventDispatcher
{
  void registerHandler(EventHandler handler);
  void startEventLoop();
}
```

Event handling class

```
class MyEventHandler implements EventHandler
{
   void handleEvent(Event event)
   {
      System.out.println ( "Event occurred" );
   }
}
```


Event dispatching class

```
class MyEventDispatcher implements EventDispatcher
{
   EventHandler handler ;

   void registerHandler(EventHandler handler)
   {
      this.handler = handler ;
   }

   void startEventLoop() { … }

   void dispatchEvent(Event e)
   {
      handler.handleEvent(e);
   }
}
```

Event loop

```
class MyEventDispatcher implements EventDispatcher
{
   …
   void startEventLoop()
   {
      while (true)
      {
         Event e = … ; //get next event
         dispatchEvent(e);
      }
   }
   …
}
```


Event loop as thread

```
class MyEventDispatcher implements EventDispatcher,
                                   Runnable
{  …
   void startEventLoop()
   {
      new Thread(this).start();
   }

   public void run()
   {
      while (true)
      {
         Event e = … ; //get next event
         dispatchEvent(e);
      }
   }
}
```

Event driven program

```
class Program
{
   public static void main(String [] args)
   {
      MyEventHandler handler = … ;
      MyEventDispatcher dispatcher = … ;
      dispatcher.registerHandler(handler);
      dispatcher.startEventLoop();
   }
}
```


Example - Clock

- As a simple example, we will show how to build a `Clock` object from a `Ticker` object
- The `Ticker` generates tick events
- The `Clock` handles the tick events by printing out “tick” messages
- The `TickHandler` interface describes the interface between the `Clock` and the `Ticker`

Ticker and TickHandler

```
interface TickHandler
{
   void tick();
}

class Ticker  // initial idea
{
   Ticker(int delay, TickHandler h) {…};
   void start() {…};
   void stop() {…};
}
```


Clock

```
class Clock implements TickHandler
{
  public static void main (String [] args)
  {
     Ticker ticker = new Ticker(100, new Clock());
     ticker.start();
     Thread.sleep(1000);
     ticker.stop();
  }

  public void tick()
  {
     System.out.print("tick...");
     System.out.flush();
  }
}
```

Implementation of `Ticker`

- The `Ticker` class uses an internal thread to generate and dispatch tick events
- Thus, `Ticker` has a `run` method and implements the `Runnable` interface
- The `Thread.sleep` method is used to introduce a delay between tick events

Starting and stopping the ticker

- The `start` method creates a new `Thread` to execute the `run` method
- The `stop` method asks the thread to stop by setting a variable to true
- After every event, the thread checks to see if there has been a request to stop the ticker
- Note that it is not a good idea to try and stop threads directly - why not?

Ticker class

```
class Ticker implements Runnable
{
  int delay ;
  TickHandler handler ;
  boolean stopped = true ;

  Ticker (int delay, TickHandler handler)
  {
      this.delay = delay;
      this.handler = handler;
  }

  public void start() { … }
  public void stop() { … }
  public void run() { … }
}
```


start and stop methods

```
public void start()
{
     if ( stopped )
     {
        stopped = false;
         new Thread(this).start();
     }
}

public void stop()
{
     stopped = true;
}
```

run method

```
public void run()
{
      while (! stopped)
      {
         try
         {
            Thread.sleep(delay);
         }
         catch (InterruptedException e){ … }
         handler.tick();
      }
}
```


Java Timer classes

- Java provides two `Timer` classes
- The `javax.swing.Timer` class is similar to the `Ticker` class
- However, it uses the AWT event handling framework and generates `ActionEvents`
- The `java.util.Timer` class schedules tasks using `TimerTask` objects
- This corresponds to a different form of event-driven programming

An introduction to GUI Programming in Java

- Java provides extensive support for GUI programming
- But for historical reasons, there are two different packages involved
- AWT is the original GUI toolkit
- Swing is the modern replacement for AWT
- But Swing is built using AWT so you need to know something about both packages
- More recently, the JavaFX library has been added, but we will concentrate on using Swing

AWT

- AWT stands for Abstract Windowing Toolkit
- AWT is built on top of the native windowing system
- This is inefficient and means that the “look and feel” is different on each platform
- However, AWT is still used as the basis of Swing

Swing

- Swing is a lightweight user interface layer built on top of AWT
- It doesn't depend on the underlying window system so it is more efficient and more flexible
- The "look & feel" of a Swing application can be customised
- But Swing still depends on AWT for event handling and layout

History of AWT and Swing

- JDK 1.0
  - AWT with original event handling model
- JDK 1.1
  - Revised event handling model
  - Early prototypes of Swing available
- JDK 1.2 and beyond
  - Swing replaces AWT as main GUI toolkit
  - AWT and Swing become part of the Java Foundation Classes (JFC)

GUI Programming

- GUI programming makes extensive use of object-oriented libraries and frameworks
- This is because there is a lot of common behaviour that can be inherited from abstract classes
- In order to understand how to use a particular framework, you need to know how the basic abstractions fit together

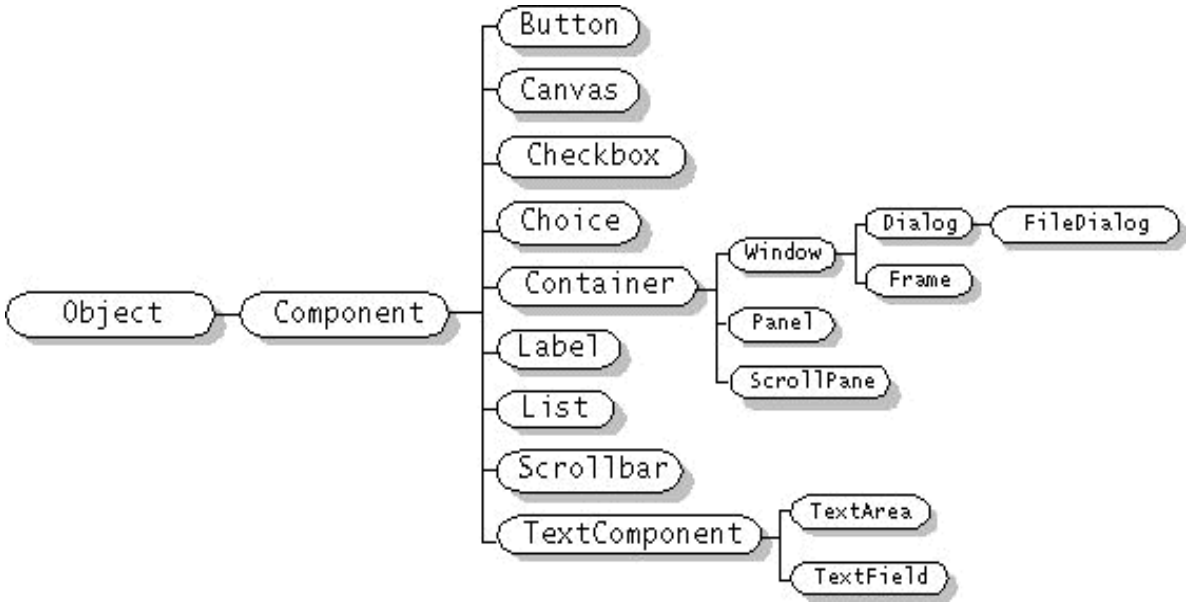
Fundamental concepts

- GUI interfaces are constructed from *components*
- Components are grouped together into *containers*
- *Layout managers* determine the size and position of components and containers
- Users interact with components and generate *events*
- *Event handlers* respond to those events by updating the state of the application

Components

- GUI interfaces are constructed from objects such as buttons, menus, dialog boxes, etc.
- Java refers to such objects as *components*
- A component is something with a graphical appearance that can interact with a user
- The Java GUI libraries contain an extensive collection of different components
- Each component inherits behaviour from the `java.awt.Component` class

AWT Components



Containers

- A container is a special kind of component that can contain other components
- A window is an example of a container
- Containers can contain other containers - for example, panels and panes of a window
- Every container inherits behaviour from the `java.awt.Container` class

Layout managers

- Each container has an associated *layout manager*
- The layout manager is responsible for determining the size and position of the components within the container
- Components are automatically resized and repositioned if the container is resized or its contents are changed

Component class

```
public abstract class Component
{
   public void setName(String name);
   public String getName();
   public void setForeground(Color c);
   public Color getForeground();
   …
   public void setSize(int w, int h);
   public void setLocation(int x, int y);
   public void setVisible(boolean b);
   public void paint(Graphics g);
   public void repaint();
}
```


Container class

```
public class Container extends Component
{
  public Component add(Component c);
  public void add(Component c, Object constraints);
  public void remove(Component c);
  public void setLayout(LayoutManager m);
  …
}
```

Top-level containers

- A top-level container is a container that isn't enclosed by another container
- Examples of top-level containers include
  - windows, frames, applets, dialogs
- Typically, a GUI application creates a top-level container, fills it with user interface components, and displays it on the screen

Windows and Frames

- The Java GUI libraries make a distinction between windows and frames
- A `Window` is "a top-level window with no borders and no menubar"
- A `Frame` is "a top-level window with a title and a border"
- GUI applications typically create frames rather than windows

Displaying windows and frames

- Initially, a newly created window (or frame) is invisible
- Before it can be displayed on the screen, its preferred size has to be calculated
- This is done by calling the `pack` method
- Alternatively, the `setSize` operation can be called instead to specify a fixed size
- Finally the `setVisible` method is used to make the window (or frame) visible

Structure of an AWT application

```
import java.awt.*;
import java.awt.event.*;
class GUI
{
   static void main(String [] args)
   {
      Frame frame = new Frame();
      // add components to frame
      frame.pack();
      frame.setVisible(true);
   }
}
```

`Window` and `Frame` classes

```
public class Window extends Container
{
  public void pack();
  …
}

public class Frame extends Window
{
  public void setTitle(String title);
  public void setMenuBar(MenuBar menuBar);
  …
}
```


Event handling in GUI programs

- Once a graphical user interface has been set up, the user is allowed to interact freely with the components of the interface
- By clicking on a button or selecting from a menu, the user can generate events
- The GUI program must respond to those events by updating its state
- Thus, GUI programs are event-driven by their very nature

Structure of GUI program

- Simple GUI programs have the following structure:
  1. Set up GUI interface components
  2. Set up event handlers
  3. Wait for events
- Once the program has initialised itself, everything else is event driven

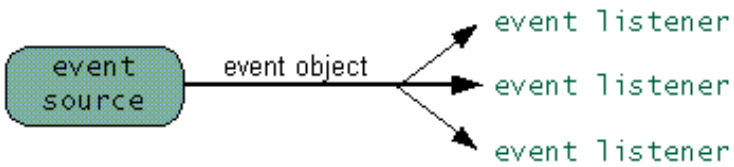
Event handling in AWT

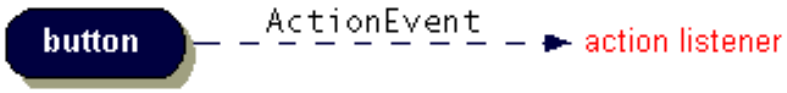
- There have been two versions of event handling in AWT
- The AWT 1.0 event handling model required each component to provide its own event handler
- The AWT 1.1 event handling model separates event handlers from components
- We will only consider the AWT 1.1 model

AWT 1.1 Event Handling Model

- Events are generated by event sources and passed to event listeners
- Event listeners can ask to be notified about events of a particular kind from a particular source
- Any object can be an event listener, simply by implementing the appropriate interface
- This is sometimes called a *delegation* model

Event concepts and example





Event objects and listeners

- Events are represented by event objects
- Each different kind of event has its own event class:
  - e.g. `ActionEvent, MouseEvent`
- There is also a corresponding event listener interface for each kind of event:
  - e.g. `ActionListener, MouseListener`
- All AWT event classes and event handling interfaces are defined in `java.awt.event`

Examples of events and listeners

Act that results in the event	**Listener Type**
User clicks a button	`ActionListener`
User closes a window	`WindowListener`
User presses a mouse button	`MouseListener`
User moves the mouse	`MouseMotionListener`
Component becomes visible	`ComponentListener`
Component gets keyboard focus	`FocusListener`
Table or list selection changes	`ListSelectionListener`

Event sources and event listeners

- An event source must provide a mechanism for registering one or more listeners for those events
- By convention, if the event source generates *Type*`Event` events, then it provides an `add`*Type*`Listener` method
- For example, buttons can generate action events so the `Button` class provides an `addActionListener` method

Example - action events and action listeners

```
public interface ActionListener extends EventListener
{
   void actionPerformed(ActionEvent e);
}

public class Button extends Component
{
   public void addActionListener(ActionListener l);
   …
}
```

How to implement an event handler

Every event handler requires the following three bits of code:

1. A class that implements the event listening interface
2. Code to handle the event
3. Code to register an instance of the event handler class with the event source

Skeleton code

```
// A class that implements the
// listener interface
class MyListener implements ActionListener
{
  …
  public void actionPerformed(ActionEvent e)
  {
       // Code to handle the event
  }
}
```

Skeleton code - continued

```
class Application
{
   …

   // Code to register an instance of the event
   // handling class with the event source

   component.addActionListener(new MyListener());

    …
}
```


Example - using a Button

- As a simple example of event handling in AWT, we will show how to set up a button that responds to events
- The GUI will consist of a single button labelled “Click me”
- Clicking the button will result in a beeping sound

Button application

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

public class ButtonHandler implements ActionListener
{
   public static void main (String [] args)
   {
     // set up GUI and event handlers
   }

   public void actionPerformed(ActionEvent e)
   {
     // handle events
   }
}
```


ButtonHandler - main

```
public class ButtonHandler implements ActionListener
{
   public static void main (String [] args)
   {
      Frame frame = new Frame();
      Button button = new Button("Click me");
      button.addActionListener(new ButtonHandler());
      frame.add(button,BorderLayout.CENTER);
      frame.pack();
      frame.setVisible(true);
   }
   …
}
```

ButtonHandler - event handler

```
public class ButtonHandler implements ActionListener
{
   …
   public void actionPerformed (ActionEvent e)
   {
      Toolkit.getDefaultToolkit().beep();
   }
}
```


Example - closing a window

- Even the simplest GUI program needs to deal with events
- Suppose a program displays a window on the screen
- What happens when the user closes the window?
- Closing a window generates a window event that must be handled by the program

Empty Window - version 1

```
import java.awt.*;
public class EmptyWindow
{
   // Program to display an empty window

   public static void main(String [] args)
   {
      Frame frame = new Frame();
      frame.setVisible(true);
   }
}
```


Window events and listeners

- Closing a window generates a window event represented by a `WindowEvent` object
- To handle this event, an event driven program needs to implement the `WindowListener` interface
- But closing a window is just one of several possible window events
- In fact, there are 7 different window events that a `WindowListener` must handle

`WindowListener` interface

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


Adapters

- Adapters are used to simplify the task of implementing listener interfaces with more than one method
- An adapter is a class that provides a default implementation of all the methods in the interface
- A class that only wants to handle a single event simply extends the adapter class and overrides the default handler for that event

Adapter classes

- Java uses consistent naming for event related classes:
  - `TypeEvent`
  - `TypeListener`
  - `TypeAdapter`
- Examples of adapter classes include `WindowAdapter` and `MouseAdapter`

Using a WindowAdapter

```
public class WindowAdapter implements WindowListener
{
  //This class provides an empty body for each
  //method of the interface WindowListener
}

import java.awt.event.*;
public class MyWindowCloser extends WindowAdapter
{
   public void windowClosing (WindowEvent e)
   {
       System.exit(0);
   }
}
```

Empty Window - version 2

```
import java.awt.*;
public class EmptyWindow
{
   // Program to display an empty window

   public static void main (String [] args)
   {
      Frame frame = new Frame();
      // deal with window closing events
      frame.addWindowListener(new MyWindowCloser());
      frame.setVisible(true);
   }
}
```


Using inner classes

- Event handlers are often written using anonymous inner classes:

```
new ClassOrInterfaceName()
{
   //anonymous class's body

   public void methodName(…)
   {
      …
   }
}
```

- The syntax is rather ugly but this approach can simplify the code if the event handler needs to refer to local variables

Empty Window - version 3

```
import java.awt.*; import java.awt.event.*;
public class EmptyWindow
{
   public static void main (String [] args)
   {
      Frame frame = new Frame();
      frame.addWindowListener(
            new WindowAdapter()
            {
              public void windowClosing(WindowEvent e)
              {
                  System.exit(0);
              }
            }
                                );
      frame.setVisible(true);
   }
}
```


Closing a Swing window

- The `setDefaultCloseOperation` method provided by the Swing `JFrame` class simplifies closing a window
- There are four possible options:
  - Do nothing
  - Hide the window
  - Dispose of the window
  - Exit from the program
- The default behaviour is to hide the window

Empty Window - version 4

```
import javax.swing.*;
public class EmptyWindow
{
   // Program to display an empty window
   public static void main (String [] args)
   {
   // Create a Swing window
   JFrame frame = new JFrame();
   // deal with window closing events
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setVisible(true);
   }
}
```


Empty Window - version 5

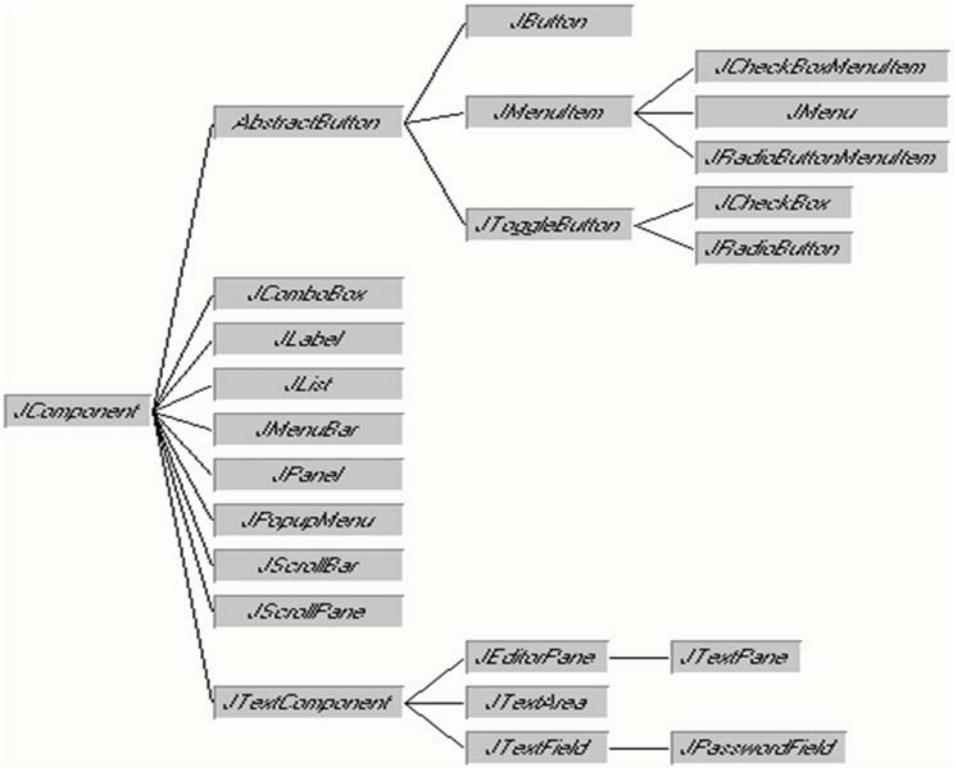
```
import javax.swing.*;
public class EmptyWindow extends JFrame
{
   public EmptyWindow()
   {
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     setVisible(true);
   }

   public static void main (String [] args)
   {
     EmptyWindow myWindow = new EmptyWindow();
   }
}
```

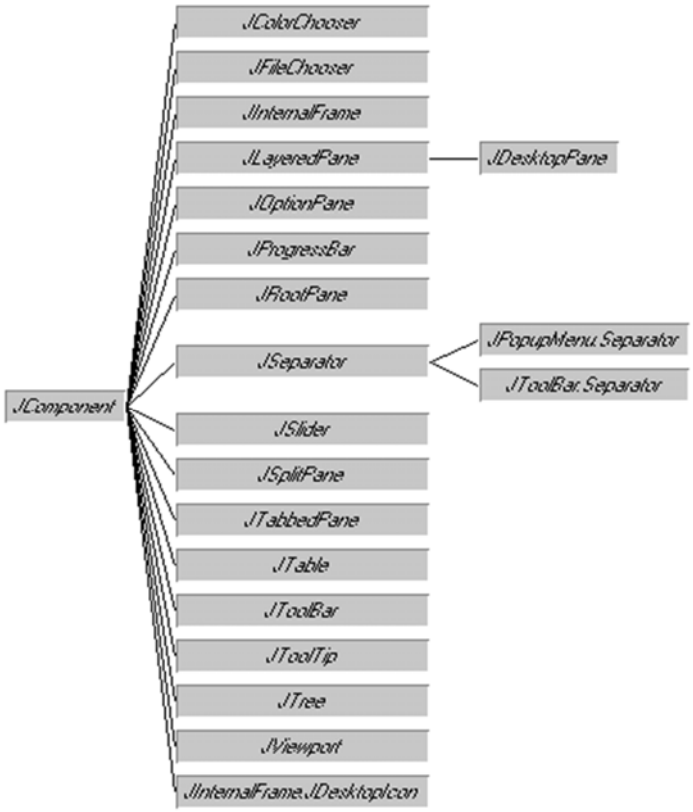
An Introduction to Swing

- Swing is a set of platform independent components built on top of AWT
- Swing components contain no native code and do not depend on the underlying window system
- Thus, they provide a richer set of functionality than AWT

Swing versions of AWT components



Additional Swing components not in AWT



Java Foundation Classes

- Swing forms part of the Java Foundation Classes (or JFC)
- JFC provides a set of features for GUI programming in Java:
  - Swing
  - Pluggable Look & Feel
  - Accessibility API
  - Java 2D API
  - Drag & Drop Support

Swing components

- Top level containers
  - applet, dialog, frame
- Intermediate level containers
  - panel, tabbed pane, split pane, scroll pane, tool bar
- Atomic components
  - button, text field, menu, list, combo box, slider
- Uneditable information displays
  - label, progress bar, tool tip
- Editable displays of formatted information
  - color chooser, file chooser, text, table, tree

Visual index to Swing

- The Java tutorial contains a useful visual index to the Swing components:
  - http://docs.oracle.com/javase/tutorial/uiswing/components/
- This contains pictures of each component and links to usage notes

Swing package

- The basic Swing components are defined in the `javax.swing` package
- This has been a standard part of the JDK since JDK 1.2
- Versions also exist for JDK 1.1
- Swing applications also need to use the standard AWT packages

JComponent

- All Swing components inherit common behaviour from `JComponent`
- This includes:
  - Support for borders
  - Support for tool tips
  - Support for painting
- Customised components should override `paintComponent` rather than `paint`

JComponent class

```
public abstract class JComponent extends Container
{
   public void setBorder(Border b);
   public void setToolTipText(String text);
   protected void paintComponent(Graphics g);
   …
}
```


Differences from AWT

- The names of Swing components begin with `J`, (e.g. `JButton` vs. `Button`)
- Swing components are all containers. However, they must be displayed in a top-level Swing container (e.g. `JFrame`)
- Swing components must be added to the *content pane* of the top-level container
  - Starting with Java version 5.0, you can add components directly to the frame, and it adds them to the content pane
- Layout is also done via the content pane
  - Starting with Java version 5.0, you can set layout directly to the frame

Swing Frames

- A `JFrame` is a top-level Swing window with a title bar and window controls
- Two important differences between `Frame` and `JFrame` are:
  - The existence of the content pane
  - The ability to set a default close operation
- `JFrame` also provides support for an optional menu bar

`JFrame` class

```
public class JFrame extends java.awt.Frame
{
  public Container getContentPane();
  public void setContentPane(Container contentPane);
  public void setDefaultCloseOperation(int op);
  public void setJMenuBar(JMenuBar menuBar);
  …
}
```

Swing Frames – changes introduced in Java 5

- Adding components to Swing Frames

Java 5	Older Compiler
`add(component)`	`getContentPane().add(component)`

- Setting layout to Swing Frames

Java 5	Older Compiler
`setLayout(manager)`	`getContentPane().setLayout(manager)`

Structure of a Swing application

```
public class SwingApplication
{
   public static void main(String [] args)
   {
      JFrame frame = new JFrame();
      // create a component
      frame.add(component,location);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Example - hello world

```
import javax.swing.*;
import java.awt.*;

public class HelloWorldSwing
{
   public static void main(String[] args)
   {
       JFrame frame = new JFrame("HelloWorldSwing");
       JLabel label = new JLabel("Hello World");
       frame.add(label,BorderLayout.CENTER);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.pack();
       frame.setVisible(true);
   }
}
```


Using `JPanel` as a content pane

- `JPanel` is an intermediate Swing container that is useful for grouping components together
- Many Swing programs use a `JPanel` as their content pane
- This works well because `JPanel` is opaque
- Other intermediate containers would not be suitable as content panes

Swing application using `JPanel`

```
public class MyPanel extends JPanel
{
   public MyPanel()
   {
     // create and add components to panel
   }

   public static void main(String [] args)
   {
      JFrame frame = new JFrame();
      frame.setContentPane(new MyPanel());
      frame.pack();
      frame.setVisible(true);
   }
}
```


Event handling in Swing

- Event handling in Swing uses exactly the same mechanisms as AWT
- Event sources generate events which are sent to registered event listeners
- Thus, to use a Swing component in a program, you need to know what kind of events it can generate

Examples of Swing components

- We will consider just a few of the Swing components
- A `JLabel` is used to provide descriptive text and cannot be edited
- A `JButton` is used to represent a button
- A `JTextField` is used to represent a simple one-line text box

Labels

- The `JLabel` class is used for displaying a fixed text or image on the screen
- Labels cannot be selected or edited and therefore don't generate events
- The text of a label can be specified using HTML

`JLabel` class

```
public class JLabel extends JComponent
{
   public JLabel(String text);
   public JLabel(Icon icon);
   public void setText(String text);
   public String getText();
   public void setIcon(Icon icon);
   public Icon getIcon();
   …
}
```


Example - using a timer

- As a simple example of event driven programming, we will show how to use the `javax.swing.Timer` class
- A timer is an object that can generate timing events at regular intervals
- We will use an instance of the `Timer` class to implement a clock that displays the time
- The clock will be represented using a `JLabel`

Timer events and listeners

- Objects of the `javax.swing.Timer` class generate events of type `ActionEvent`
- Each timer object has an associated listener object that implements the `ActionListener` interface
- The action events generated by the timer are passed to the `actionPerformed` method provided by its action listener

`Timer` class

```
package javax.swing;

public class Timer
{
  public Timer (int delay, ActionListener l);
  public void start();
  public void stop();
   …
}
```

`Clock` application

- The clock application will display the amount of time that it has been running
- A `Clock` object is a special kind of `JLabel` that is used to display the current time
- The event handler updates the label
- The `convert` method is used to convert an integer representing the number of ticks into a time string

`Clock` application – `main` method

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

public class Clock extends JLabel
                    implements ActionListener
{
   …
   public static void main (String [] args)
   {
       JFrame frame = new JFrame();
       frame.add(new Clock(),BorderLayout.CENTER);
       frame.setDefaultCloseOperation
                              (JFrame.EXIT_ON_CLOSE);
       frame.pack();
       frame.setVisible(true);
   }
}
```

Clock class - event handling

```
public class Clock extends … implements …
{
  private int ticks = 0;
  public Clock()
  {
      this.setText(convert(ticks));
      Timer timer = new Timer(100, this);
      timer.start();
  }
  public void actionPerformed (ActionEvent e)
  {
      ticks++;
      this.setText(convert(ticks));
  }
   …
}
```


Clock class - convert method

```
public class Clock extends … implements …
{
   …
   private String convert(int ticks)
   {
     int t = ticks % 10 ;         // tenths
     int s = (ticks / 10) % 60 ; // seconds
     int m = (ticks / 600) ;      // minutes
     return m + ":" + s + ":" + t ;
   }
  …
}
```

Buttons

- Buttons can have icons and text labels associated with them
- They generate action events when they are pressed
- Swing also supports check boxes and radio buttons
- All the different kinds of button inherit their behaviour directly or indirectly from the `AbstractButton` class

`AbstractButton` class

```
public abstract class AbstractButton extends JComponent
{
   public void addActionListener(ActionListener l);
   public void setText(String label);
   public String getText();
   public void setIcon(Icon icon);
   …
   public boolean isSelected();
   public void setSelected(boolean b);
}
```

Button classes

```
public class JButton extends AbstractButton {…}
public class JMenuItem extends AbstractButton {…}
public class JToggleButton extends AbstractButton {…}
public class JCheckBox extends JToggleButton {…}
public class JRadioButton extends JToggleButton {…}
```


Example - counting button clicks

- The `SwingApplication` program will create a frame with two components: a button and a label
- The button will generate action events when it is pressed
- The label will be used to display the number of button clicks
- The event handler will update the label

SwingApplication - outline

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

public class SwingApplication implements
                                ActionListener
{
  private JLabel label;
  private JButton button;
  private int numClicks = 0;

  public Component createComponents() { … }
  public static void main(String[] args) { … }
  public void actionPerformed(ActionEvent e) { … }
}
```


SwingApplication – main method

```
public static void main(String[] args)
{
  JFrame frame = new JFrame("SwingApplication");
  SwingApplication app = new SwingApplication();
  Component contents = app.createComponents();
  frame.add(contents,BorderLayout.CENTER);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(280,100);//Instead of frame.pack();
  frame.setVisible(true);
}
```

SwingApplication - set up GUI

```
public Component createComponents()
{
  button = new JButton("I'm a Swing button!");
  label = new JLabel(" Number of button clicks: "
                     + numClicks);
  button.addActionListener(this);
  JPanel pane = new JPanel();
  pane.add(button);
  pane.add(label);
  return pane;
}
```


SwingApplication - handle events

```
public void actionPerformed(ActionEvent e)
{
  numClicks++;
  label.setText(" Number of button clicks: "
                + numClicks);
}
```

`SwingApplication` - using inner class for event handling

```
public class SwingApplication
{ …
  public Component createComponents()
  {
    button = new JButton("I'm a Swing button!");
    label = new JLabel(" Number of button clicks: " + numClicks);
    button.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        numClicks++;
        label.setText(" Number of button clicks: " + numClicks);
      }
    });
    JPanel pane = new JPanel();
    pane.add(button); pane.add(label); return pane;
  } …
}
```


Text fields

- The `JTextField` class implements a one-line box for text entry
- The width of the text field can be specified as an argument to the constructor
- When the user types text into the box and presses return, an action event is generated
- It is also possible to intercept individual character strokes

Swing Text components

- Swing provides a rich set of text components
- These include built-in support for HTML and RTF document formatting
- `JTextField` inherits common behaviour from `JTextComponent`
- Other Swing text classes include `JTextArea` and `JEditorPane`

`JTextField` class

```
public class JTextField extends JTextComponent
{
   public JTextField();
   public JTextField(int columns);
   public JTextField(String text);
   public void addActionListener(ActionListener l);
   …
}
```

JTextComponent class

```
public abstract class JTextComponent
                        extends JComponent
{
   public String getText();
   public void setText(String text);
   public void setEditable(boolean b);
   public boolean isEditable();
   …
}
```


Example - temperature conversion

- The `Converter` program will create a frame with four components in a single row: two text fields, each preceded by a label
- The user will enter the temperature in Celsius into one text field and then press Enter key causing this text field to generate an action event
- The event handler will use the other text field to display the equivalent temperature in Fahrenheit

Converter class - outline

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

public class Converter implements ActionListener
{
  private JFrame converterFrame;
  private JPanel converterPanel;
  private JTextField celsiusTF, fahrenheitTF;
  private JLabel celsiusL, fahrenheitL;

  public Converter() { … }
  private void addWidgets() { … }
  public void actionPerformed(ActionEvent e) { … }

  public static void main(String[] args)
  {
    Converter converter = new Converter();
  }
}
```


Converter - setting up the frame

```
public Converter()
{
  // Create the frame and panel
  converterFrame = new JFrame("Convert Celsius to Fahrenheit");
  converterPanel = new JPanel();

  // Create and add the widgets to the panel
  addWidgets();

  // Use the panel as the content pane of the frame
  converterFrame.setContentPane(converterPanel);

  // Exit when the window is closed
  converterFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  // Show the converter
  converterFrame.pack();
  converterFrame.setVisible(true);
}
```

Converter - setting up GUI

```
private void addWidgets()
{
   celsiusTF = new JTextField(10);
   fahrenheitTF = new JTextField(10);
   celsiusL = new JLabel("Celsius");
   fahrenheitL = new JLabel("Fahrenheit");
   celsiusTF.addActionListener(this);
   converterPanel.add(celsiusL);
   converterPanel.add(celsiusTF);
   converterPanel.add(fahrenheitL);
   converterPanel.add(fahrenheitTF);
}
```


Converter - handling events

```
public void actionPerformed(ActionEvent e)
{
  // Parse degrees Celsius as a double
  // and convert to Fahrenheit
  String celsius = celsiusTF.getText();
  int fahrenheit =
  (int) ((Double.parseDouble(celsius)) * 1.8 + 32);
  fahrenheitTF.setText("" + fahrenheit);
}
```