(More) Graphical User Interfaces

CS1092

Sean Bechhofer

seanb@cs.man.ac.uk

Topics

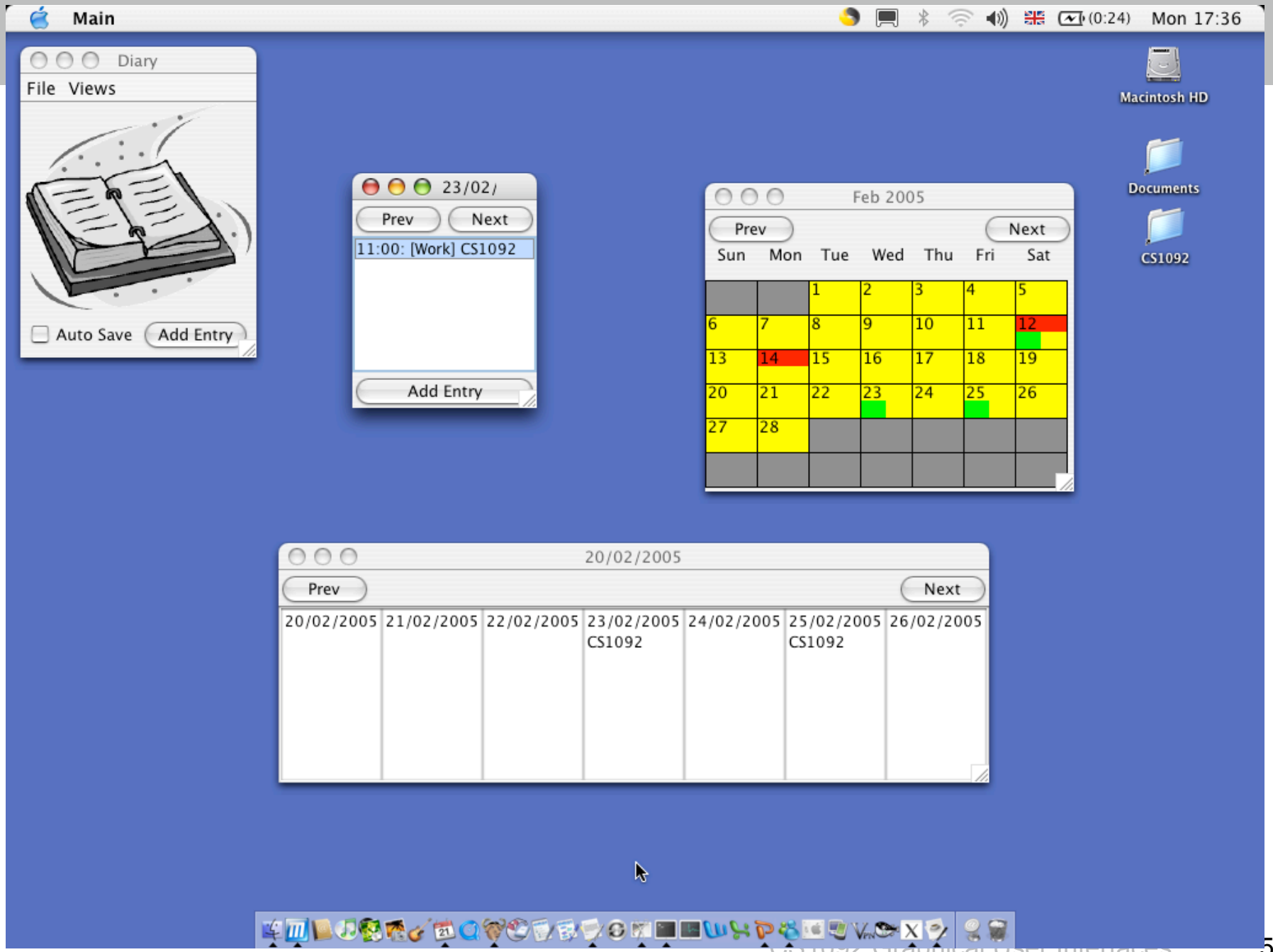
- Topics to cover in the next four lectures:
- More Widgets
  - More sophisticated menus, menu bars, check boxes, radio buttons etc.
- More Listeners
- Separating Model from View
- Applets
  - Java programs that can be included in web pages
- Graphics
  - Drawing objects on the screen
  - Custom Components/Building your own widgets.

Supporting Materials

- These lectures **won't** tell you everything you need to know.
- This is **not** a comprehensive description of all the details.
- There are many aspects of the Swing library classes that **you** should discover for yourselves.
- Liang
  - Chapters 10, 11, 12.
- Savitch
  - Chapters 16, 17, 18.
- Sun's Java Tutorial
  - http://java.sun.com/docs/books/tutorial
- Java API documentation
  - http://www.cs.man.ac.uk/campusonly/jdk1.4/docs/api/index.html

Case Study: Diary

- Throughout the next few lectures we will use a **case study** as motivation and to supply examples of the concepts being introduced.
- The case study will be a **Diary** application that allows us to keep track of entries occurring at particular times and dates.
- The Diary has a **Graphical User Interface** that allows us to view entries, add and remove them, save them to files and so on.
  - The GUI offers different views on the data (month/week/day etc).


Main
(0:24)
Mon 17:36
Diary
File Views
Auto Save
Add Entry
Macintosh HD
Documents
CS1092
23/02/
Prev
Next
11:00: [Work] CS1092
Add Entry
Feb 2005
Prev
Next
Sun Mon Tue Wed Thu Fri Sat
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28
20/02/2005
Prev
Next
20/02/2005
21/02/2005
22/02/2005
23/02/2005
CS1092
24/02/2005
25/02/2005
CS1092
26/02/2005

The Diary Model

- The Diary has a rather simple data model.
- A **Diary** is made up of a number of instances of **DiaryEntry**
- Each **DiaryEntry** has a description, a category and a date associated with it.
- Entries may be be either considered as “all day” entries, in which case they have no time, or can be considered as occuring at a particular time.
  - Of course, for a **real** diary application we’d expect to be able to add things like end times/durations, repeating entries etc.
  - However, this particular model will be sufficient to allow us to demonstrate the GUI concepts that we’re introducing without complicating matters too much with.

The Diary Model

- We specify our basic Diary and DiaryEntry objects through the use of two **interfaces**.
- Recall that an interface specifies a collection of operations or methods that we expect a class to provide, but says nothing about how that implementation is to be achieved.
- It provides a signature of the methods
  - Result type
  - Name
  - Argument types

Diary interface

```
/** A diary is a collection of entries. The diary can be queried to
   find out entries occurring within particular timeframes. A diary can
   be observed: observers will be informed of any changes that occur
   (e.g. addition or deletion of entries). */

public interface Diary {
    /** Return all entries that occur between the given dates
        (inclusive). */
    public Iterator getEntries( Calendar from,
                                Calendar to );

    /** Return all entries in the diary. Returns an Iterator over
        a collection of Diaryentries.  */
    public Iterator getAllEEntries();


    /** Return all the categories currently in use */
    public Iterator getCategories();
```

Diary interface

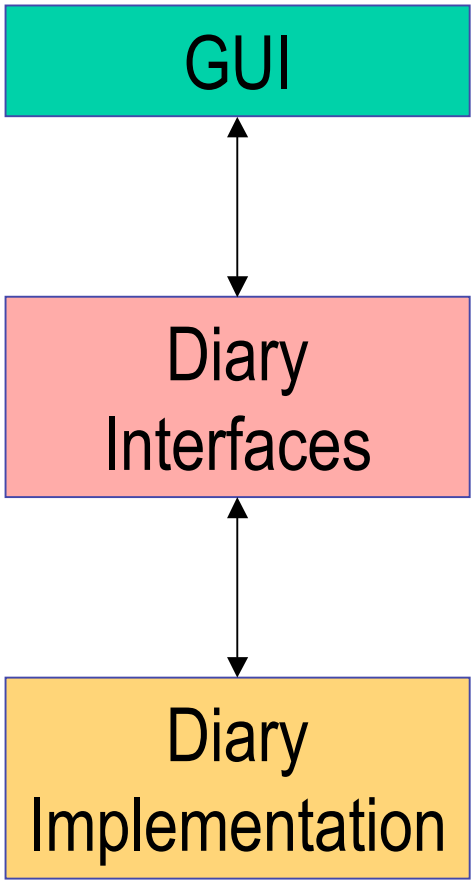
```
    /** Remove a specific entry.
        @returns true if the entry was removed */
    public boolean removeEntry( DiaryEntry entry );

    /** Add an entry to the diary
        @ returns the entry being added */
    public DiaryEntry addEntry( Calendar date,
                                String description,
                                String category,
                                boolean allDay );

     /** Write the diary. */
    public void write( Writer writer ) throws IOException;

    /** Read the diary */
    public void read( Reader reader ) throws IOException;

    /** Add an observer that wishes to be informed of any changes to
        the diary. */
    public void addObserver( Observer o );

    /** Remove an observer */
    public void deleteObserver( Observer o );

}
```

DiaryEntry interface

```
/** Represents an entry occurring in a diary. Events are associated
   with a date/time, and have a textual description. Some entries are
   marked as all day entries, in which case their time is
   immaterial. Events can be ordered, based on their dates. Untimed
   entries are considered to occur before timed ones. */

public interface DiaryEntry extends Comparable {
    /** The date that this entry occurs at. Uses Calendar
        class rather than Date. */
    public Calendar getDate();

    /** The Description applying to this entry */
    public String getDescription();

    /** The category of the entry */
    public String getCategory();

    /** Whether the entry occurs all day */
    public boolean allDay();
}
```

The Diary Model

- Note that the Diary interface says **nothing** about the way that we might expect the diary to be displayed or presented to the user.
- Also, the interfaces say **nothing** about how we might implement the model
  - All our GUI code will be be written using these interfaces
  - Allows us to swap in alternative implementations of Diary and DiaryEntry without impact on the GUI code.
- We have a clean **separation** of **responsibility**.



Packages

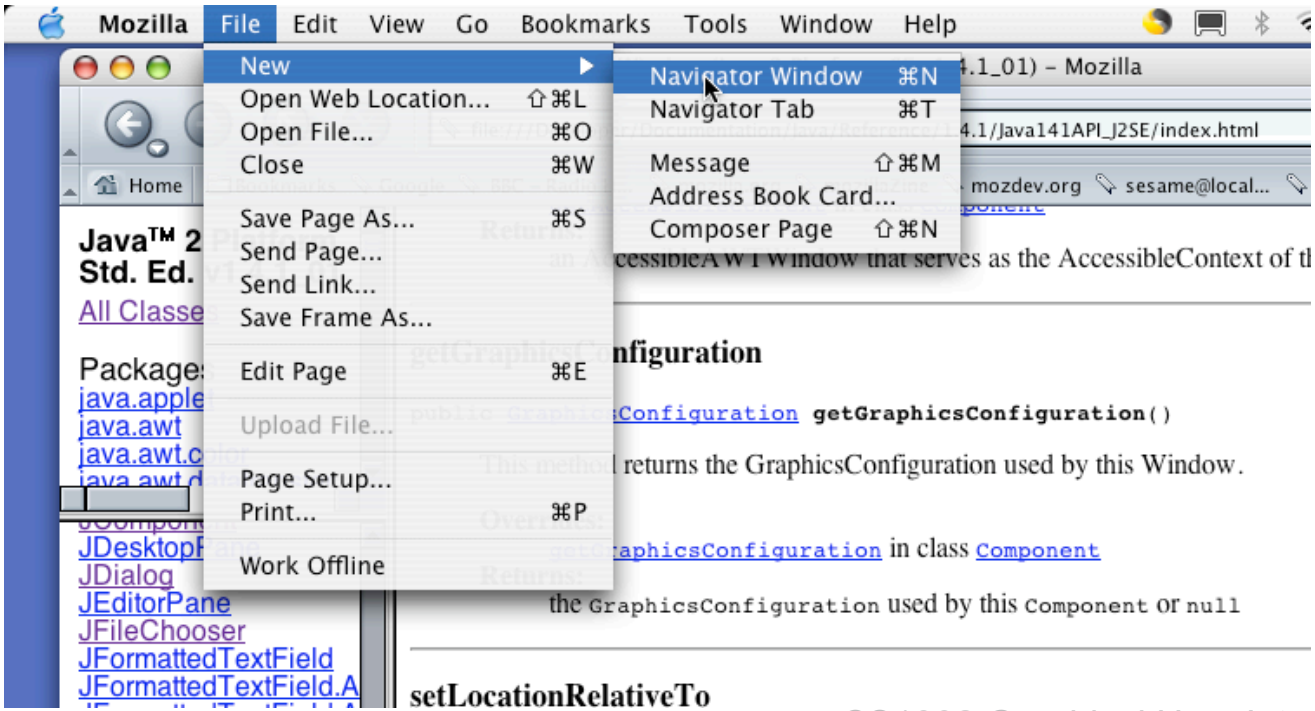
- You will also note from the sample code that we have begun to split our implementation into a number of different packages:
  - **model**: the interfaces for the diary
  - **gui**: the GUI classes.
  - **impl**: an Implementation of the diary
- Using packages in this way helps to maintain the separation between the various responsibilities.

Basic Components of the Diary GUI

- Our Diary GUI will be built up from some basic windows.
- Main Window
  - Main entry point to the program. Controls operations such as reading and writing a diary, general user preferences etc.
- Month View
  - Shows a month's worth of entries.
- Week View
  - Shows a week's worth of entries.
- Day View
  - Shows the entries occuring on a particular day, and allows addition and deletion of entries for that day.

Menus

- Menus are often seen in windowing applications and provide a convenient and space-saving way of providing access to a number of functionalities or options.
  - For example, Mozillas's menu bar offers around 50 different options and operations.



What exactly is a menu?

- A menu is essentially a **list** of items, some of which may be associated with some functionality such as
  - executing some application code,
  - setting options;
  - opening another window.
- Menus may also be **hierarchical**, containing nested menus
  - Again, this can help save space if there are a large number of options that are available.
- Swing provides some basic classes that can support all of the above.
  - JMenu represents a menu
  - JMenuItem (and subclasses) represent menu items

What's a menu bar?

- A menu bar is an area, usually at the top of an application window that contains a number of menus.
- The menu items are usually "hidden" and drop down when the user clicks on the menu.
- Swing provides a class for representing menu bars on GUI application windows:
  - JMenuBar

Adding a menu bar and menus to a frame

- Consider the creation method of DiaryGUI (a subclass of JFrame). Within that method, we create a new (empty) menu bar and add it to the frame as follows:

```
JMenuBar menuBar = new JMenuBar();
this.setJMenuBar( menuBar );
```

- We now have an empty menu bar to which we can add menus.

```
JMenu fileMenu = new JMenu( "File" );
menuBar.add( fileMenu );
```

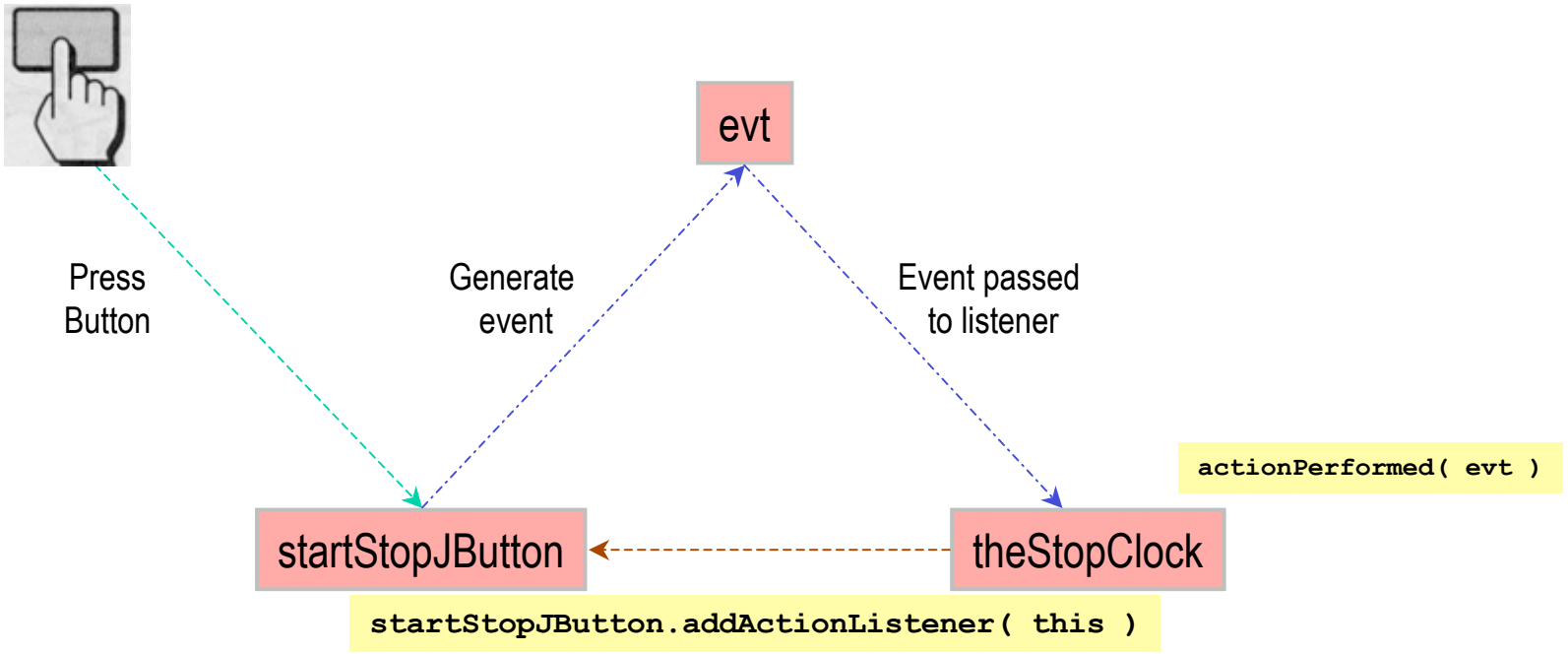
- This will create a new empty menu with the title “File”.
- Next we need to add some menu items to the menu.
  - Note that as with the components that we saw in CS1081 the order in which we add things is not important. We can add the menu bar to the frame before adding the menu items or afterwards.

Buttons and ActionListeners (Recap)

- In CS1081, we (hopefully!) learnt how to use a JButton in a Swing user interface
  1. An ActionListener is added as a listener to the JButton.
  2. When the JButton is pressed, an ActionEvent is passed to each listener that the JButton has.
  3. The listeners then perform some appropriate action.
- Recall our simple StopClock application example.
  - The StopClock had a button that would start and stop the clock.
  - The StopClock object registered as an action listener with the button.
  - When the button is pressed, the StopClock did the appropriate thing.

StopClock Sources and Listeners (Recap)

- What happens in the stop clock?



- When the button is pressed, a new `ActionEvent` object is created and passed to the `StopClock` object. The `actionPerformed()` method is then invoked with the event as argument.

Menus and MenuItems

- A JMenuItem is very like a JButton
  - In fact they share a common abstract superclass JAbstractButton.
  - When a menu item is selected, it's like pressing the button.
- So, in order to perform an action when a menu item is selected, we must add an ActionListener to the JMenuItem.
  - Just like with the JButton.

Creating and adding menu items

- Creating a new JMenuItem is also similar to creating a JButton. We can supply a label which will be used to display the item in the menu.
-

```
JMenuItem openItem = new JMenuItem( "Open...");
fileMenu.add( openItem );
```

- We then add the menu item to the menu. Items will appear in the order in which they are added.
- We can also add separators between the items.

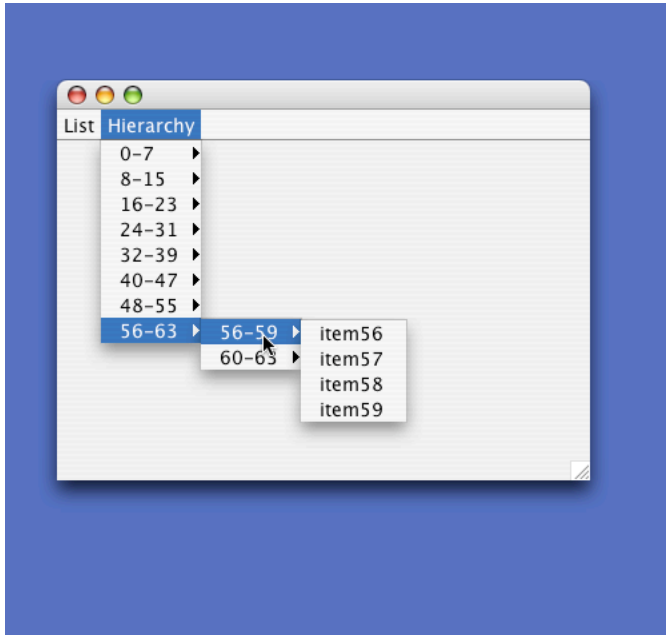
```
JMenuItem saveItem = new JMenuItem( "Save...");
fileMenu.add( saveItem );

fileMenu.addSeparator();

JMenuItem quitItem = new JMenuItem( "Quit");
fileMenu.add( quitItem );
```

Look and Feel

- The way in which things GUI components appear depends on the platform.
  - For example, menu separators appear under Windows as a line drawn between the menu items.
  - On other platforms, e.g. MacOS X, the separators appear as some blank space between the items.
  - Similarly, buttons may appear differently.
- This is known as **Look and Feel**.
- We can set the Look and Feel either programmatically or via options passed to the JVM.
- If we leave the responsibility for the Look and Feel to the **default** implementation, it can help to ensure that our applications “fit in” with the underlying operating system.
  - Users **like** consistency……

Hierarchical Menus

- Menus can be hierarchical
  - A menu item may be a menu itself
  - By default, nested menus appear as “pull-right” menus.
- The organisation of the menu items only impacts on the way in which the menu appears.



Listeners

- ActionListeners are responsible for responding to action events such as button presses and (as we've just seen) menu selections.
- So far we've seen two approaches:
  - The GUI/JFrame itself implements ActionListener and provides the actionPerformed() method.
  - Another named class implements ActionListener -- recall the log book example from the end of CS1081.
- There is another approach which we can use when supplying listeners: so-called Anonymous Classes.

Anonymous Classes

- Anonymous classes allow us to produce “one-time” implementations of interfaces without explicitly introducing a new named class
- Anonymous classes are often used if we only need one instance of the class, and the class definition is short.
  - ActionListener is often an example of this.
  - It’s usually the case that each button is going to do something **different**, so we might expect to have to provide different listeners for each button.

Anonymous Listeners

- Here's an example of the use of an anonymous ActionListener, taken from the DiaryGUI class.

```
JMenuItem monthView = new JMenuItem( "New Month View" );
monthView.addActionListener( new ActionListener() {
          public void actionPerformed( ActionEvent evt ) {
              openNewMonthView();
          }
     });
viewMenu.add( monthView );
```

- When the menu item is selected, the openMonthView() method is called.
- Note that we don't need to worry about checking what the event source was -- we know it'll be menu item that we've just created.
  - Compare with our StopClock example.

Anonymous Listeners

- Similarly, we can use anonymous listeners with JButtons
- As we've already seen, JButtons and JMenuItems are very similar to JButtons.

```
JButton addEvent = new JButton( "Add Event" );
addEvent.addActionListener( new ActionListener() {
          public void actionPerformed( ActionEvent evt ) {
               openAddEvent();
          }
     });
buttons.add( addEvent );
```

More Buttons

- So far we've seen simple Action buttons (and menu items).
- We can press/select them and an action is performed.
- Other kinds of buttons are available in Swing.

Check Box

- A checkbox is a special kind of button that has some state -- it can be **on** or **off**.
- Clicking the button toggles the state.
  - We can find out which state the button is in via the method

```
public boolean isSelected();
```

- Clicking the button also fires an ActionEvent, so we can add ActionListeners to checkboxes in exactly the same way as with buttons.

CheckBox Example (DiaryGUI)

- A Check box is added to the GUI.

```
autoSave = new JCheckBox( "Auto Save" );
autoSave.setSelected( false );
buttons.add( autoSave );
```

- If the check box is selected when an update occurs, then we perform some operation.

```
if ( autoSave.isSelected() ) {
  changesMade++;
  if ( ( changesMade % AUTO_SAVE_COUNT ) == 0 ) {
    autoSave();
    changesMade = 0;
  }
}
```

- Note here we **don't** use an ActionListener -- we're only interested in the state of the checkbox.

Radio Buttons

- Radios have buttons for selecting stations or wavebands.
  - Each button selects a band.
  - Only one can be selected at a time
  - Selecting one "**toggles**" the others to off.
- Radio buttons in java apply the same philosophy
  - Buttons are grouped together.
  - Only one button in a group can be on at one time.
- This can be useful when selecting **mutually exclusive** options.

Radio Buttons

- When we create a JRadioButton, we can add it to a ButtonGroup.
- All the buttons in a given ButtonGroup group are linked together
  - When one is selected, the others will be deselected.
  - Note, however, that only the button that is selected will fire an Action Event.
  - The buttons don't have to be in the same **place**.
- By default, all the buttons in a group are initially unselected.
- Once a button in a group is selected, one button will **always** be selected in the group.

Radio Button Example (RadioTimeChooser)

```
/* Create a new TimeChooser */
   public RadioTimeChooser() {
        ButtonGroup group = new ButtonGroup();
        setLayout( new GridLayout( 0, 4 ) );
        JRadioButton timeRadioButton =
            new JRadioButton( "No Time" );
        add( timeRadioButton );
        timeRadioButton.addActionListener( new ActionListener() {
                public void actionPerformed( ActionEvent evt ) {
                    selectedTime = "";
                }
            });
        timeRadioButton.setSelected( true );
        group.add( timeRadioButton );

        for (int time = START_TIME;
             time <= END_TIME;
             time++ ) {
            String displayTime = time + ":00";
            timeRadioButton =
                new JRadioButton( displayTime );
            add( timeRadioButton );
            group.add( timeRadioButton );
            timeRadioButton.addActionListener( this );
        }
   }
```

Radio Button Example

```
public void actionPerformed( ActionEvent evt ) {
         /* Use the action command, e.g. the string associate with the
            button */
         selectedTime = evt.getActionCommand();
    }
```

- Here we're using the getActionCommand() method on action events.
  - This returns the **command string** associated with the event.
  - By default, this is the **text** that is on the button that has been pressed.
- You may notice that JRadioButton and JCheckBox share a common abstract super class: JToggleButton
  - This represents those buttons that can toggle between two states.

Other Listeners

- So far we've seen a number of examples of ActionListener
  - An action listener responds to a single Action Event raised by a component such as a Button.
- There are many other kinds of events that may occur

? What might these be?

Other Listeners

- KeyEvents
  - A key is pressed, released or typed.
- MouseEvents
  - The mouse is clicked, moved or dragged.
- WindowEvents
  - A window is opened, closed or iconified
- Each of these events has its own listener class.

KeyListener

- KeyListeners allow us to listen for events associated with keys being pressed.
- The KeyListener interface has three methods:

```
public void keyPressed(KeyEvent e);

public void keyReleased(KeyEvent e);

public void keyTyped(KeyEvent e);
```

? Why might we want to have separate methods for press, release and type?

KeyListener Example (AddEvent)

```
description = new JTextField( 20 );

description.addKeyListener( new KeyListener() {
  public void keyPressed( KeyEvent evt ) {
    if ( evt.getKeyCode() == KeyEvent.VK_ENTER ) {
        addEvent();
        setVisible( false );
        dispose();
    }
  }
  /* Don't care about these.*/
  public void keyTyped( KeyEvent evt ) {
  }
  public void keyReleased( KeyEvent evt ) {
  }
});
```

- This adds a KeyListener to the text field that “listens” for the return key.
  - If the Return key is pressed, some action is performed.
- We’re using an anonymous listener, but have to provide **more** method implementations.

Adapters

- In the previous example, we had to implement **all** the methods specified in the KeyListener interface.
- However, we were only really interested in one of them.
  - This means a lot of work just to supply a bunch of empty method implementations.
- Consider a situation where we had an interface with **twenty** methods in it and we only wanted/needed to implement one.
  - This could involve a lot of unnecessary “empty” code.
- Listener Adapters are a solution to this problem
  - An Adapter provides empty implementations of the methods specified in an interface.
  - To use the adapter class, we extend and override only the method implementations we need.

KeyAdapter Example (AddEvent)

```
description = new JTextField( 20 );

description.addKeyListener( new KeyAdapter() {
  public void keyPressed( KeyEvent evt ) {
    if ( evt.getKeyCode() == KeyEvent.VK_ENTER ) {
         addEvent();
         setVisible( false );
         dispose();
    }
  }
});
```

- Note that we don't now need to supply the implementation of the two listening methods that we're not interested in.
- You'll find that adapters are particularly useful when creating anonymous listeners.
- Adapters are available for (among others) MouseListener, KeyListener, WindowListener.

? Why is there no adapter for ActionListener?

? Why do we have Listeners -- why not just have Adapters?

MouseListener

- A mouse listener listens for mouse events.

```
public void mouseClicked( KeyEvent e );

public void mouseEntered( KeyEvent e );

public void mouseExited( KeyEvent e );

public void mousePressed( KeyEvent e );

public void mouseReleased( KeyEvent e );
```

? Why might we want to distinguish between pressing and clicking?

MouseListener Example (WeekView)

```
displayLists[days].addMouseListener( new MouseAdapter() {
  public void mouseClicked( MouseEvent evt ) {
    if ( evt.getClickCount() == 2 ) {
    /* Open up a new day view on the selected
       date. */
      DayView view =
        new DayView( getDiary(),
                     displayDates[fdays]);
      view.setVisible( true );
    }
  }
});
```

- In this example, we listen for a **double click** of the mouse, and then execute some action
- Again, note the use of an Adapter class, which saves us from having to supply empty implementations for the four MouseListener methods that we're not really interested in.

JList

- A JList allows us to display a list of things.
- Items can be selected/unselected
- The list offers a number of functions that allow us to query which items are selected.
- We give the list a collection of objects that should be displayed.

```
display = new JList();

...
Vector v = new Vector();
/* Add the entries */
Calendar dayEnd = (Calendar) day.clone();
DiaryUtils.dayEnd( dayEnd );
for ( Iterator it = getDiary().getEntries( day,
                                           dayEnd );
      it.hasNext(); ) {
  DiaryEntry entry = (DiaryEntry) it.next();
  v.add( entry );
}
display.setListData( v );
```

JList

- We can add listeners to the list that will perform some operation based on user actions.
- Here, we add a key listener that listens for the delete key, and then deletes the currently selected entry.

```
if ( !readOnly() ) {
  display.addKeyListener( new KeyAdapter() {
    public void keyPressed( KeyEvent evt ) {
      if ( evt.getKeyCode() == KeyEvent.VK_DELETE ||
           evt.getKeyCode() == KeyEvent.VK_BACK_SPACE ) {
        DiaryEntry deletee =
          (DiaryEntry) display.getSelectedValue();
        deleteEntry( deletee );
      }
    }
  });
}
```

- The basic JList machinery takes care of selection and deselection of list items.

JComboBox

- A JComboBox is a drop down widget that allows the user to select one of a number of options.
- It's like a mixture of a menu and a list.
- Unlike a list or a collection of radio buttons, however, the combo box can allow the user to enter a **new** option if the selections provided aren't appropriate.
  - We do this by setting the JComboBox to be **editable**.

JComboBox Example (AddEntry)

- In the AddEntry window, we allow the user to select the category that the entry is in.
- The list offers all the categories that are currently in use.
  - If the user want a new one, they can type it in.

```
Vector cats = new Vector();
for (Iterator catIt = diary.getCategories();
     catIt.hasNext();) {
  String category = (String) catIt.next();
  cats.add( category );
}
categoryChooser = new JComboBox( cats );
categoryChooser.setEditable( true );

descriptionPanel.add( categoryChooser );
```

- Just like JList, we can get the **selection** from the JComboBox.

```
String category = (String) categoryChooser.getSelectedItem();
```

Widgets and toString()

- Widgets like JList and JComboBox don't really know anything about the things that are in the list -- they're just Objects.
  - When we get things out of the list, we may need to cast them to an appropriate type.
- If we add an object to a JList or a JComboBox, how will it appear on the interface?
  - For example, in the DayView window, we simply add a collection of DiaryEntry instances to the list
- By default, if we don't say anything, the widget will use the object's toString() method to provide a string that will appear in the list.
  - It is also possible to supply a "custom renderer" that will apply some alternative mechanism for working out how to display objects in the list.

Dialogs

- Dialogs are common in Graphical User Interfaces
- They are used to **inform** the user of situations such as errors.
- They are used to **confirm** user actions that may not be undo-able such as deletion.
- They are used to obtain **user input** such as
  - Locations of files/resources
  - Simple form-filling.
- Swing provides a component JDialog that supports dialogs

JOptionPane

- Swing also provides a class called JOptionPane that has a number of convenience methods that make it easy to pop up standard dialog boxes.
  - **ConfirmDialog**. Asks a confirming question, like yes/no/cancel.
  - **InputDialog**. Prompt for some input.
  - **MessageDialog**. Tell the user about something that has happened.
- The dialogs produced by JOptionPane are **modal**
  - This means that we have to finish dealing with the dialog box before we can do anything else with the GUI.

JOptionPane Example (DayView)

```
private void deleteEvent( DiaryEntry deletee ) {
  if ( deletee!=null ) {
    /* Pop up a dialog to confirm */
    String message = "Really Delete: " + deletee + "?";
    int confirm =
      JOptionPane.showConfirmDialog( this,
                                     message,
                                     "Confirm Delete",
                                     JOptionPane.YES_NO_OPTION );
    if ( confirm == JOptionPane.YES_OPTION ) {
      getDiary().removeEvent( deletee );
    }
  }
}
```

- This example pops up a dialog with a message checking that the user really wants to delete the given event.
- Note the use of **static variables** to indicate the possible options available and the returned value.

JDialog

- We can also **extend** JDialog if we want to provide some more sophisticated dialog interfaces.

```
/** Allows addition of entries. */
public class AddEvent extends JDialog {

...

  /** Create a new Dialog allowing addition of an entry. */
  public AddEvent( Diary aDiary,
                   Calendar aDate,
                   boolean fixedDate ) {
    /* Creates a modal dialog, e.g. blocking. */
    super( (Frame) null, true );
    /* Use the diary passed in */
    diary = aDiary;
    ...
  }
...
}
```

JFileChooser

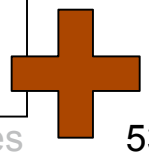
- It’s often the case that in a GUI we want to read and write from files.
- JFileChooser gives us a basic component that allows selection of a file.
- The class deals with all the unpleasant details of interacting with the underlying filesystem.

JFileChooser

```
fileChooser = new JFileChooser();
FileFilter filter = new FileFilter() {
  public boolean accept( File f ) {
    return f.getName().endsWith( ".dry" );
  }
  public String getDescription() {
    return "Diary Files";
  }
};
fileChooser.setFileFilter( filter );
```

- Creates a file chooser that looks for particular kinds of file.
- Note again the use of an anonymous class to provide the FileFilter

```
int returnVal = fileChooser.showOpenDialog( this );
if(returnVal == JFileChooser.APPROVE_OPTION) {
  try {
    diary.read( new FileReader( fileChooser.getSelectedFile() ) );
  } catch ( IOException ex ) {
    ex.printStackTrace();
  }
}
```



WindowListener

- As another example of a listener class, a WindowListener will listen for Window events such as windows being opened, closed, or iconified.

```
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 );
```

- As with other listeners, an adapter class (WindowAdapter) is provided.
- Be careful to distinguish between the different methods called when windows are closed:
  - **windowClosing()** is called when the user attempts to close a window
  - **windowClosed()** is called when the window has been closed and disposed of.

WindowListener example (DiaryGUI)

- A simple example of a window listener is one that checks to see if the application is in a sensible state to quit.
- The windowClosing() method is called when the user attempts to close the window.
- If the program doesn't explicitly hide and dispose of the window when this is called, the close is cancelled.

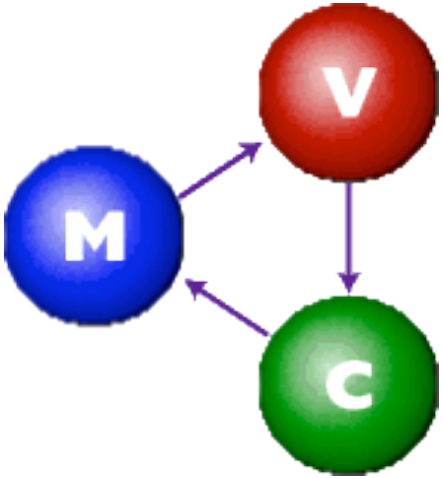
```
/* Use our own window listener to handle closing */
this.setDefaultCloseOperation( DO_NOTHING_ON_CLOSE );

addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent event ) {
      attemptQuit();
    }
  });
```

- The attemptQuit() method checks the state, checks with the user, and closes down if everything's ok.

Separating Model from View

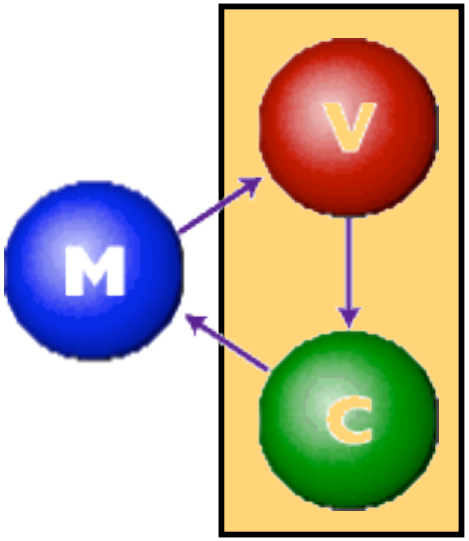
- In our early user interface programs, a **single** object (StopClock) was responsible for both holding the state of the application and for the display of the GUI that was displayed.
- Instead, we often wish to separate out
  1. The application logic (the things that we are representing along with the operations for maniputing them).
  2. The display of the interface -- buttons and widgets.
  3. The control of interactions with the interface -- e.g. interpreting mouse clicks
- 1 is often referred to as the **model**, while 2 is the **view** and 3 the **controller**.
- This so-called Model-View-Controller paradigm dates back to research at Xerox in the late 70's and a language called Smalltalk.

MVC



- The Model represents the data of the application.
- The View is the visual representation of that data.
- A Controller takes user input on the view and translates that to changes in the model.

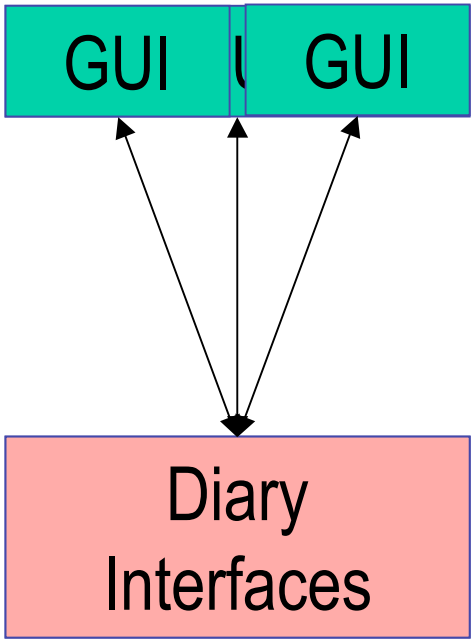
MVC in Swing



- In Swing, the View and Controller have been collapsed to a single object (UI Object)
- However, there is still a strong separation between the model and the components responsible for display and interpretation of user actions.

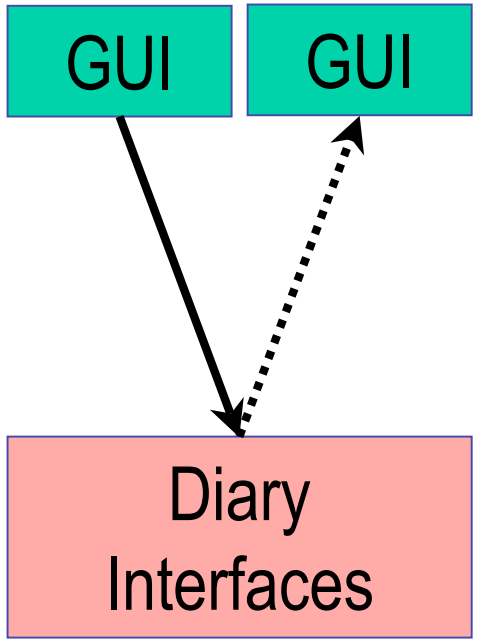
The Diary Model

- As we've already hinted at, our Diary and DiaryEntry interfaces comprise the Model in our application.
- The GUI classes provide the View and Controller.
- This separation has a number of nice consequences. In particular, it makes it easier for us to provide **multiple** views over the same data.



Update

- This is all very well, but introduces some interesting questions.
- For example, if I add a new entry to my diary, how do I make sure that these **changes** are displayed in my interface?
- This is particularly important if I have **more than one** view open on the model at the same time.



Observers and Observables

- We can solve this in part through use of the Observer/Observable pattern.
- An **Observable** is an object that can be observed to see if it changes.
- An Observable object can have a number of **Observers**.
- Whenever the Observable object changes, it notifies its Observers. They can then decide whether to perform some appropriate action.
- Java provides a class and interface in the java.util package that make it easy for us to implement Observer/Observable pairs.

Observable

- Observable is a class that keeps a list of Observers.
- It provides a number of methods for adding and removing observers, and for notifying the observers of any changes.

```
public class java.util.Observable extends java.lang.Object{
    public java.util.Observable();
    public synchronized void addObserver(java.util.Observer);
    public synchronized void deleteObserver(java.util.Observer);
    public void notifyObservers();
    public void notifyObservers(java.lang.Object);
    public synchronized void deleteObservers();
    protected synchronized void setChanged();
    protected synchronized void clearChanged();
    public synchronized boolean hasChanged();
    public synchronized int countObservers();
}
```

Observer

- Observer is an interface with a single method.
- It is this method that's called when the Observer is being notified.

```
interface java.util.Observer{
    public abstract void update(java.util.Observable,java.lang.Object);
}
```

Observer and Observable in the Diary

- Recall the Diary interface definition
- We have methods that allow us to add and remove observers.

```
    /** Write the diary. */
    public void write( Writer writer ) throws IOException;

    /** Read the diary */
    public void read( Reader reader ) throws IOException;

    /** Add an observer that wishes to be informed of any changes to
        the diary. */
    public void addObserver( Observer o );

    /** Remove an observer */
    public void deleteObserver( Observer o );

}
```

Diary Implementation

- Our implementation of the Diary interface class extends Observable.
- This provides us with the necessary implementations of the addObserver() and removeObserver() methods.
- Within our implementation, we need to make sure that whenever we make a change to the state of the Diary we notify the observers.
  - Again, methods from Observable allows us to do this.

```
public boolean removeEvent( DiaryEntry entry ) {
  if ( entries.remove( entry ) ) {
    setChanged();
    notifyObservers( this );
    return true;
  }
  return false;
}
```

DiaryView

- DiaryView is an **abstract** class providing basic functionality required in one of our diary GUI views.
  - The basic views (Day, Week and Month) extend this class.
- It extends JFrame (providing windowing functionality) and also implements the Observer interface.
- The idea here is that when we create instances of the view, we add them to the Diary's Observer collection.
- When changes are made to the Diary's internal state, the Diary will then notify all its observers, allowing the views to update themselves and maintain a consistent view.

Swing Widgets

- Many of the Swing widgets apply this philosophy as well.
- For example, JList has a JListModel that handles the internal data of the list. The model responds to actions enacted on the GUI widget and changes its internal state.
  - For example, when the mouse is clicked, an item may be selected.
- The JList GUI component may then update its appearance to reflect the change in the internal state.
  - Highlighting the selected item.

Applets

- Java is a language intended from the start to be used in the context of the Internet/World Wide Web.
- An **Applet** is a Java program that can be included in an HTML page.
- When a Java-enabled browser is used to view the page, the code for the applet is **downloaded** to your machine and **executed** by the JVM.
- Programs can be divided (roughly) into two classes:
  - **Applications**: Basic Java programs
  - **Applets**: Programs that are intended to be run across the Internet. If it's not an applet, it's an application.

Applets

- Applets are like “little applications”
  - Although there is no actual restriction on the size of an applet.
- Applets are very similar to Swing GUIs.
  - Most of the simple GUI programs that you wrote last semester could be implemented as applets.
- Applets differ from applications though in that they are not run using a main() method.
  - Instead, the applet implements methods that initialise, start or display the applet.
  - The creation of the applet instance and calling the appropriate methods is then handled by the applet viewer or web browser.

JApplet

- Applets are implemented using the JApplet class.
- A JApplet is a top level container.
  - Just like a JFrame, it has a contentPane that we can put things into and a menubar where we can add menu items.
- Rather than popping up a new window though, the applet window will appear in the browser, embedded in the HTML page.

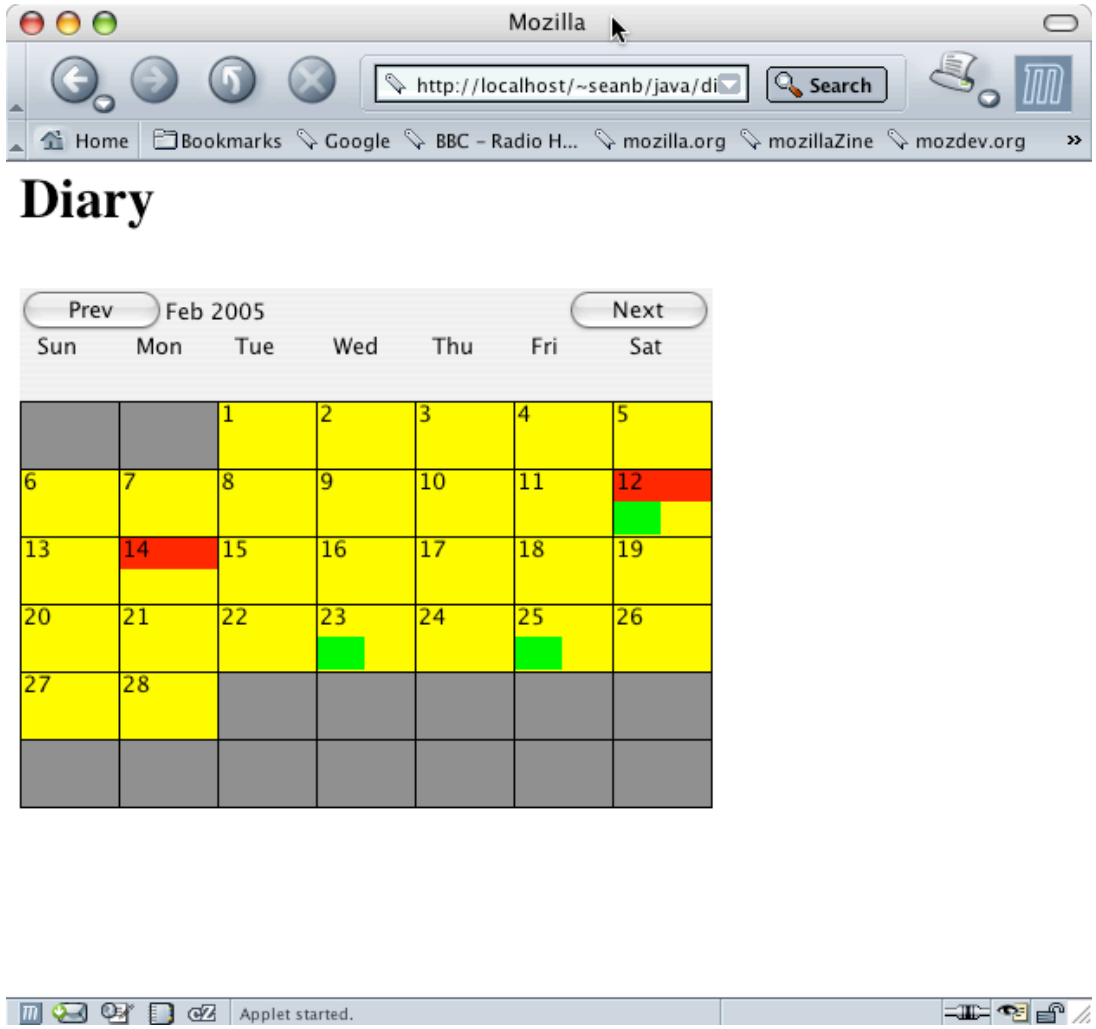
Key Applet Methods

- init()
  - Performs some initialisation. This is a good place to put code that would usually be called in the constructor.
- start()
  - Starts the applet running
- stop()
  - Stops the applet running, say when the user quits the browser.
- destroy()
  - Performs a final cleanup

Applet Example: DiaryApplet

- As an example, we provide an applet that provides a read-only view on a diary.
- The applet uses the same basic code for displaying a month, but wraps it up as a JApplet rather than a JFrame.

Applet Example: DiaryApplet



```
public class DiaryApplet extends JApplet implements Observer {
...
    /** Initialise the Diary Applet */
    public void init() {
        /* Initialise to today */
        monthStart = DiaryUtils.getMonthStart( new GregorianCalendar() );
        Container contents = getContentPane();
        contents.setLayout(new BorderLayout());

        JPanel prevNextPanel = new JPanel( new BorderLayout() );
        JButton previous = new JButton( "Prev" );
        previous.addActionListener( new ActionListener() {
                public void actionPerformed( ActionEvent evt ) {
                    previous();
                }
            });
        prevNextPanel.add( previous, BorderLayout.WEST );
...
        diary = new SimpleDiaryImpl();
...
    }
```

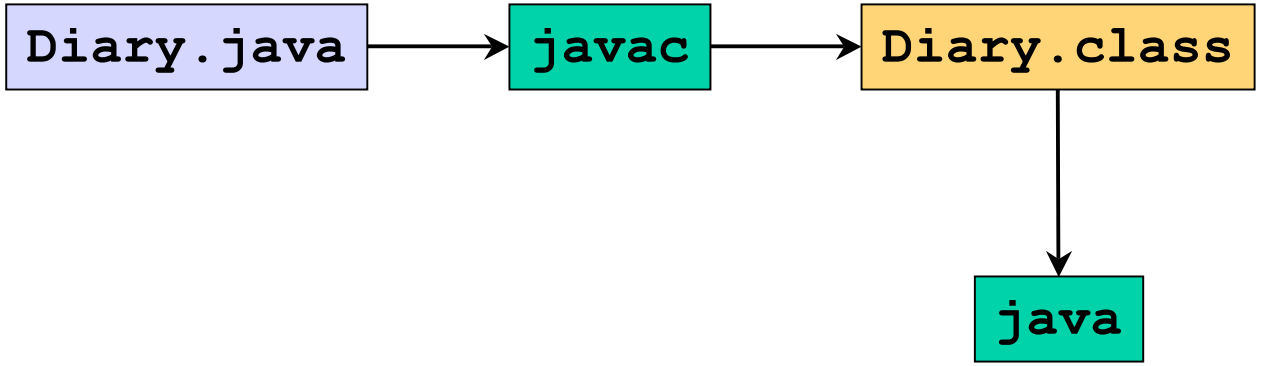
Embedding Applets

- Applets are embedded in HTML pages using the <APPLET> tag.
- This is understood by most modern browsers.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html><head>
<title>Diary</title>
</head>

<body>
<h1>Diary</h1>

<applet code="gui.DiaryApplet"
        codebase="."
        width="400" height="300">
    <param name="diary.file" value="diary.dry">
Your browser is completely ignoring the &lt;APPLET&gt; tag!
</applet>

</body>
</html>
```

Finding Class files

- With a Java application, we compile our java source code to produce class files.



- When we run an application, the JVM reads those class files.

? How does it know where to find them?

Code Locations

- So far, your applications have been compiled and run in the **same** directory
- All the .java files and the .class files are in that directory.
- By default (if you don't tell it otherwise), the JVM will look for the class files in the directory that you run it from.

? Why might this not be a good thing?

Classpath

- The CLASSPATH is a list of **places** that tells the JVM where it can find the .class files that it needs in order to run the application.
- As we've seen, by default the CLASSPATH is simply the current directory.
- The CLASSPATH is also used by the **compiler**
  - If I'm compiling a class B that uses a class A, the compiler needs to know about class A in order to check whether it has the methods being used.

Classpaths

- We can change the CLASSPATH in one of two ways:
- There is an **environment variable** CLASSPATH. If this is set, then this is the value that the JVM (and compiler) will use.
- We can explicitly pass a value for the CLASSPATH to the JVM (or compiler) using the -classpath option.

Classpaths and Applets

- In our Applet example, the **codebase** attribute tells us where to look for the classes. The appletviewer will expect the .class files to be available at that location and will download them when necessary.

```
<applet code="gui.DiaryApplet"
        codebase="."
        width="400" height="300">
    <param name="diary.file" value="diary.dry">
Your browser is completely ignoring the &lt;APPLET&gt; tag!
</applet>
```

Jar Files

- The CLASSPATH gives us a mechanism for pointing at **alternative** locations (directories) where .class files may be found.
- This allows us to use
  - Directories other than the one in which the application is being run from.
  - More than one directory.
- However, it's still the case that the directory has to contain a .class file for **every** Java class.

? Why might this be a problem?

  ? Think applets!

Jar Files

- An alternative solution is to provide a mechanism that bundles all the .class files up into a **single file**.
- A **Jar file** is an archive that contains class files and resources (for example images or configuration files) for a Java application
- Jar files can be added to the CLASSPATH, just like directories.
- When the Jar file is passed to the JVM, the JVM extracts the necessary files from the archive and then uses them in the usual way.
- The jar tool allows us to create and unpack jar files
  - Jar files actually use the same compression mechanism that zip files use

jar

```
Usage: jar {ctxu}[vfm0Mi] [jar-file] [manifest-file] [-C dir] files ...
Options:
    -c  create new archive
    -t  list table of contents for archive
    -x  extract named (or all) files from archive
    -u  update existing archive
    -v  generate verbose output on standard output
    -f  specify archive file name
    -m  include manifest information from specified manifest file
    -0  store only; use no ZIP compression
    -M  do not create a manifest file for the entries
    -i  generate index information for the specified jar files
    -C  change to the specified directory and include the following file
If any file is a directory then it is processed recursively.
The manifest file name and the archive file name needs to be specified
in the same order the 'm' and 'f' flags are specified.

Example 1: to archive two class files into an archive called classes.jar:
       jar cvf classes.jar Foo.class Bar.class
Example 2: use an existing manifest file 'mymanifest' and archive all the
           files in the foo/ directory into 'classes.jar':
       jar cvfm classes.jar mymanifest -C foo/ .
```

Advantages of Jar Files

- Compression
  - Jar files allow **compression** of files for efficient storage.
- Portability
  - The mechanism for handling JAR files is a **standard** part of the Java platform's core API.
- Decreased download time:
  - Applet code (classes and associated resources such as images) can be downloaded to the browser in **one transaction**.
- Security:
  - We can digitally **sign** the contents of a JAR file. Users who recognize your signature can then optionally grant your software security privileges it wouldn't otherwise have.

Jars and Applets

- In our Applet example, we can supply an **archive** attribute that points at a jar archive containing the classes. This has the advantage that we can pull all the classes over together in a single HTTP transaction, reducing overhead.

```
<applet code="gui.DiaryApplet"
        codebase="."
        archive="cs1092.jar"
        width="400" height="300">
    <param name="diary.file" value="diary.dry">
Your browser is completely ignoring the &lt;APPLET&gt; tag!
</applet>
```

Parameters and Applets

- We can pass parameter values to the Applet via the **param** tag. In our example, we pass in a parameter called diary.file which tells us where to find the source of the diary.
  - We can then use the parameter in the init() method.

```
<applet code="gui.DiaryApplet"
        codebase="."
        width="400" height="300">
    <param name="diary.file" value="diary.dry">
Your browser is completely ignoring the &lt;APPLET&gt; tag!
</applet>
```

```
        try {
            /* Read the diary from a URL. */
            String diaryFile = getParameter( "diary.file" );
            URL diaryURL = new URL ( getDocumentBase(), diaryFile );
            /* Create a new diary */
            diary.read( new InputStreamReader( diaryURL.openStream() ) );
        } catch ( Exception ex ) {
            ex.printStackTrace();
        }
```

Jar Files

- Jar files can also contain information (the **manifest**) that tells us extra things about the archive.
- For example, if the jar file contains classes that make up an application the manifest can tell us which class forms the **entry point** for the application
  - i.e. which class should have its main() method called?
- The application can then be launched by (for example) double clicking on the jar file.
  - This can be useful when distributing applications.

Applets and Security

- An Applet is a piece of code that I download from the web and run on my machine
- The code may be delivered as a jar file containing only the compiled classes.

? Is this a **safe** thing to do?

Applets and Security

- Applets cannot load libraries or define native methods.
- Applets can use only their own Java code and the Java API the applet viewer provides.
- An applet cannot ordinarily read or write files on the host that is executing it.
- An applet cannot make network connections except to the host that it came from.
- An applet cannot start any program on the host that is executing it.
- An applet cannot read certain system properties.

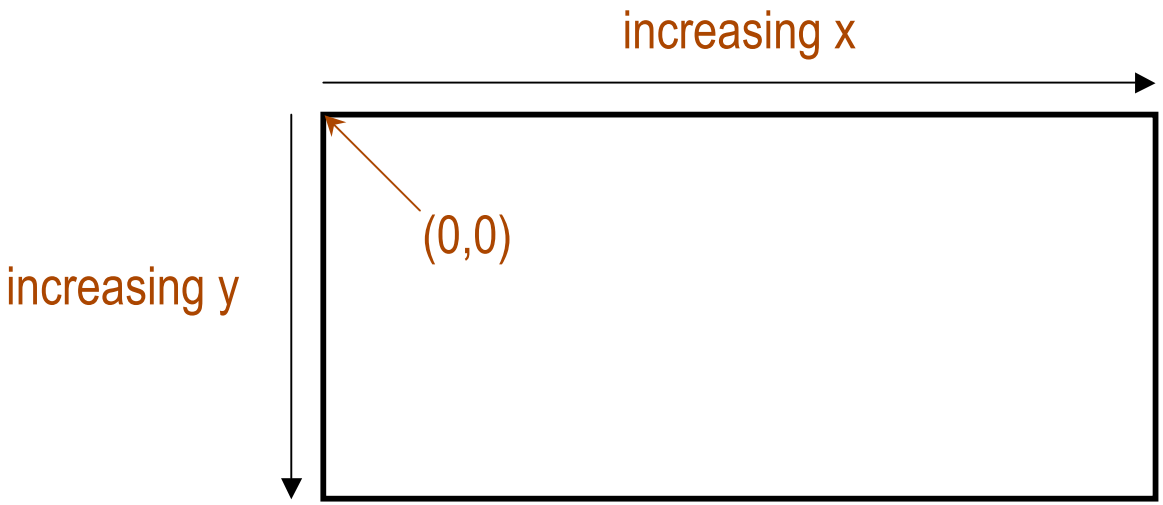
http://java.sun.com/sfaq/

Graphics

- The Graphics provides methods for drawing basic shapes and text on the screen.
- This can allow us to produce more sophisticated GUIs
- In particular, we can provide custom components -- widgets that have special rendering methods.
- Savitch Chapter 18.
- Liang Chapter 10.

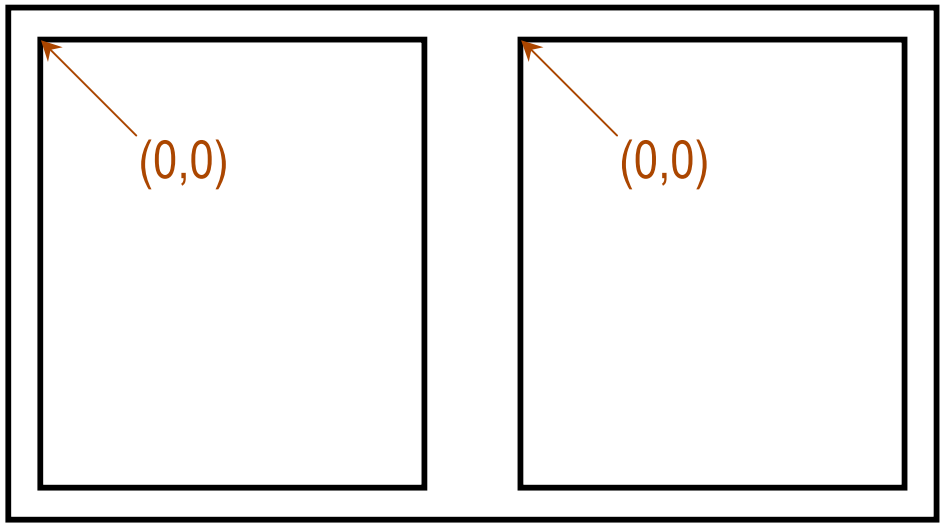
Coordinates

- Java uses an (x,y) coordinate system to refer to locations on the screen.
- The **origin** is (0,0), which occurs at the top left hand corner of the are which is being drawn on.
- The x- and y-coordinates are usually positive:
  - X-coordinates increase to the right
  - Y-coordinates increase in a downward direction.



Coordinates

- The values in (x,y) coordinates refer to pixel values.
- Note that the (x,y) coordinates are usually relative to the component that we're drawing on.
  - (0,0) is not the top left hand corner of the **screen**, but the top left hand corner of the **component**
  - This is important when we consider providing methods that paint nested components.



Graphics

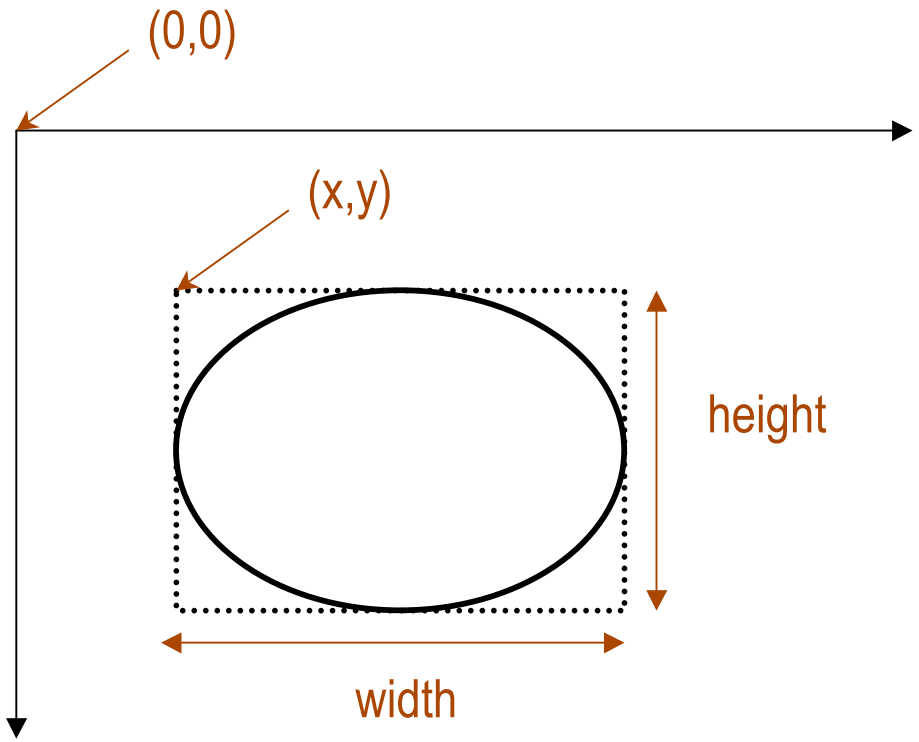
- There are a number of basic methods on Graphics for drawing objects:
  - drawLine(int x1, int y1, int x2, int y2)
    - Draw a line from the first point to the second.
  - drawRect(int x, int y, int width, int height)
    - Draw a rectangle with origin (x,y), and given width and height
  - fillRect(int x, int y, int width, int height)
    - Draw a filled rectangle with origin (x,y), and given width and height
  - drawOval(int x, int y, int width, int height)
    - Draw an oval with origin (x,y), and given width and height
  - fillOval(int x, int y, int width, int height)
    - Draw an filled oval with origin (x,y), and given width and height
  - But no drawCircle() or fillCircle()!

Graphics

- There are also methods that allow us to draw general shapes:
  - drawPolygon(int[] xPoints, int[] yPoints, int nPoints);
  - fillPolygon(int[] xPoints, int[] yPoints, int nPoints);
  - drawString(String str, int x, int y)
    - Put text at the given location.

Bounding Boxes

- When drawing objects such as ovals, the objects are positioned with respect to a **bounding box**.
- The (x,y) position given is the top left hand corner of the bounding box



Graphics and Colours

- When a drawing method like fillRect() is called, we can think of the drawing as being done using a pen.
- The colour of the pen will be the **current** colour of the Graphics object.
- We can change the current colour through the method:

```
public void setColor( Color aColor );
```

- Note the US spelling!!!
- Once the colour has been set, all subsequent drawing commands (fill and draw) will be done using that colour.

Color Class

- The Color class provides a number of static public variables that represents constants:

```
Color.red
Color.black
Color.blue
...
```

- In addition, we can create our own Color objects by passing in values for the amount of red, green and blue:

```
Color myPurple = new Color( 100,
                            255,
                            0 );
```

- Arguments are integers between 0 and 255.
  - Color(0, 0, 0) is black
  - Color(255, 255, 255) is white

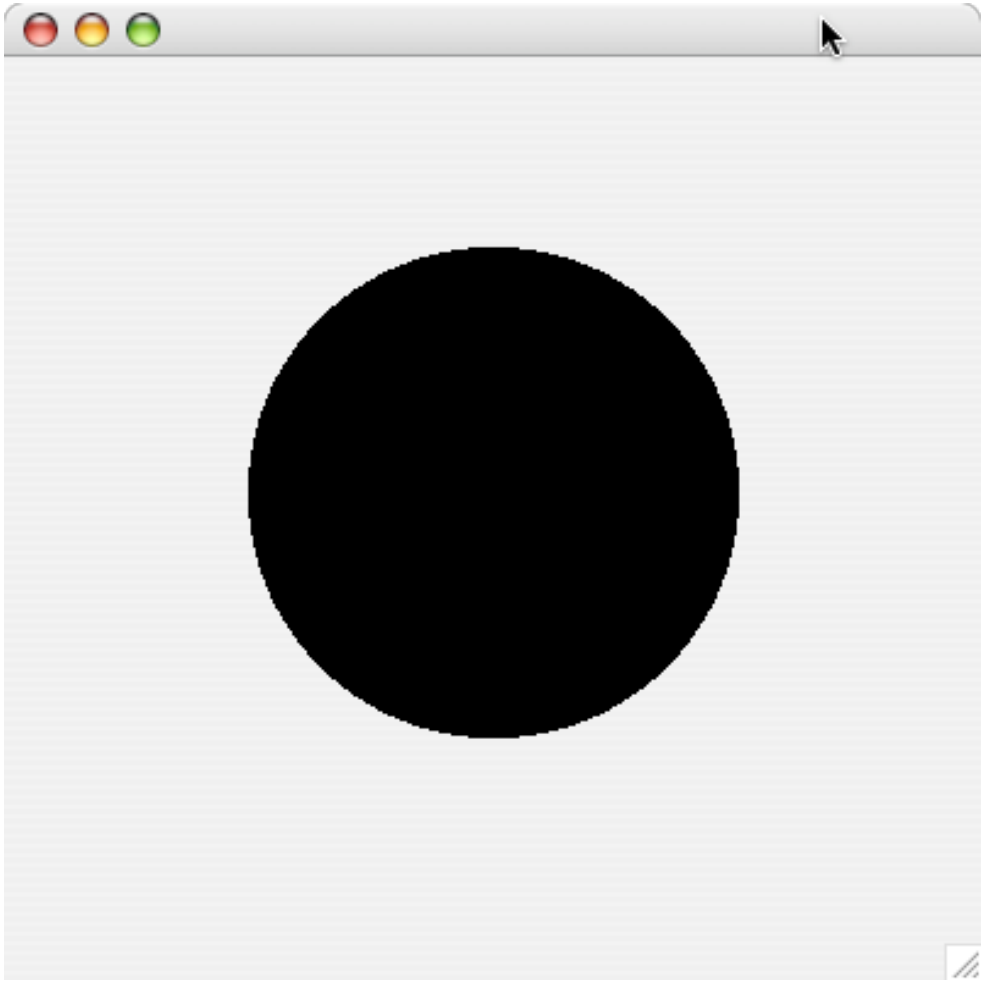
Painting Graphics

- JFrame has a method called paint().
- This takes a Graphics object as an argument
- Each Swing component has a Graphics object associated with it.
  - The Graphics object contains information specifying where on the screen the component is, the bounds etc.
  - When the paint() method is called, this Graphics object is passed in to the method.
- The paint() method will then draw the appropriate shapes on the Graphics object.

Painting a JFrame

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

public class Circle extends JFrame {

    public Circle() {
         this.setSize( new Dimension( 400, 400 ) );
         this.setDefaultCloseOperation( EXIT_ON_CLOSE );
         this.setVisible( true );
    }

    public void paint( Graphics g ) {
         super.paint( g );
         g.setColor( Color.black );
         g.fillOval( 100, 100, 200, 200);
    }

    public static void main(String[] args) {
         Circle circle = new Circle();
    }

}
```

Painting a JFrame



Calling paint()

- We do not usually need to worry about **invoking** or calling paint()
- It is called automatically when the components are drawn.
- If we don't supply a definition for paint(), the default method will be called.
- This draws the frame border, title and other basic standard features.
- All the components which are part of this frame (i.e. the components that we've added to the contentPane) will then be painted.

Custom Components

- A custom component is used when we want to have a component on an interface that has some specialised appearance.
- This is commonly used when we want an area on a GUI to display some graphics or text.
- In this case, we can use a JPanel and override the appropriate methods.
  - A JPanel simply provides an “empty” area in a GUI that we can fill with other components, or draw on.
- As a simple example, we provide a JPanel that has some circles and squares drawn on it.
  - This example may well be useful for you in the lab….

Painting Components

- As with the JFrame, the paint() method on JComponent does the following:
  - Paints the component itself -- paintComponent().
  - Paints a border -- paintBorder()
  - Paints any child components -- paintChildren()
- If we want to change the way that a JComponent looks and provide a custom painting methods for components of our interface (e.g. subclasses of JComponent) we should override the paintComponent() method rather than the paint() method.
- This ensures that borders and child components are drawn correctly.

Circles and Squares

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

public class CustomExample extends JFrame {

    private static int SIZE = 50;

    public CustomExample() {
         Container contents = this.getContentPane();
         contents.setLayout( new GridLayout( 0, 1) );
         contents.add( new Circles() );
         contents.add( new Squares() );
         this.setDefaultCloseOperation( EXIT_ON_CLOSE );
         this.pack();
         this.setVisible( true );
    }

    public static void main(String[] args) {
         CustomExample ex = new CustomExample();
    }

...
```

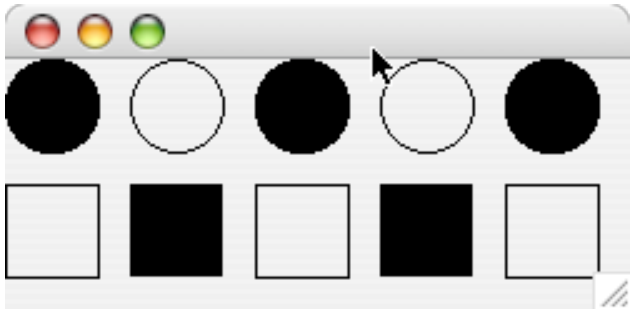
Circles and Squares

```
...
    public class Circles extends JPanel {
         public Dimension getPreferredSize() {
           return new Dimension( SIZE*5, SIZE );
         }

         public void paintComponent( Graphics graphics ) {
             super.paintComponent( graphics );
             for (int i = 0; i<5; i++) {
                  if ( i%2 == 0 ) {
                      graphics.fillOval( i*SIZE, 0, SIZE*3/4, SIZE*3/4 );
                  } else {
                      graphics.drawOval( i*SIZE, 0, SIZE*3/4, SIZE*3/4 );
                  }
              }
         }
    }
...
```

Circles and Squares

```
...
    public class Squares extends JPanel {
         public Dimension getPreferredSize() {
           return new Dimension( SIZE*5, SIZE );
         }

         public void paintComponent( Graphics graphics ) {
             super.paintComponent( graphics );
             for (int i = 0; i<5; i++) {
                 if ( i%2 == 1 ) {
                     graphics.fillRect( i*SIZE, 0, SIZE*3/4, SIZE*3/4 );
                 } else {
                     graphics.drawRect( i*SIZE, 0, SIZE*3/4, SIZE*3/4 );
                 }
             }
         }
    }
}
```

- The panels contain squares and circles drawn using the methods from Graphics.
- In this particular example, the way the panels are drawn is always the same
  - We don't make use of any local instance variables or methods
- The JPanel subclass *might* have some variables or methods that are used to change the way that the shapes are drawn.
  - For example, we might have instance variables that tells us how many circles to draw, or what colour to use.



How big is my component?

- When the layout manager is doing the layout, it needs to know the size of components so that it can allocate the right amount of space.
  - Remember that for example, in a GridLayout, all the grid cells are the size of the *biggest* component.
- Methods getMinimumSize(), getPreferredSize() and getMaximumSize() are used to provide the size of a component.
- getPreferredSize() returns a Dimension object representing the size that the component would ideally like to be. If possible, the layout manager will respect the wishes of the component.
- We can override getPreferredSize() in our custom component in order to get sensible layouts.
  - See previous examples.

Custom Component Example

- MonthViewPanel in our diary example provides a custom component that displays a month and the entries in that month
- The paintComponent() method uses a number of calls to Graphics drawing methods
- For example, here we draw grey boxes with a black border to pad the month out.

```
...
  if (days[rows][cols] == null) {
    /* Blank, so fill with grey */
    g.setColor( DiaryGUIPreferences.getGreyColour() );
    g.fillRect(currentX, currentY,
               cellX, cellY );
    g.setColor( Color.black );
    g.drawRect(currentX, currentY,
               cellX, cellY );
    } else {
...
```

Repainting

- The way in which the panel is drawn will change depending on the state of the Diary object that the panel is displaying.
  - The appearance of the month panel reflects the underlying entries in the diary for that month.
- In our diary example, we need to make sure that the panel gets repainted whenever necessary.
- The library classes will deal with repainting when the window is resized.
- However, if the diary object changes, the display of the panel may need to change to reflect this
  - An entry may have been added to a day in the displayed month.
- The repaint() method can be called to ensure that components are repainted.

Custom Component Diary Example

- Recall that the update() method in MonthView is responsible for ensuring that the view updates -- in order to make sure that this happens, we call repaint().
- Although repaint() is being called on the MonthView object, it will call paint(), which will then call the paintChildren() method, ensuring that the MonthViewPanel is updated.

```
protected void update() {
  /* Update the title */
  this.setTitle( DiaryUtils.formatDate( DiaryUtils.MONTH_AND_YEAR,
                                        monthStart ) );
  display.setDate( monthStart );
  /* Make sure everything gets redisplayed. */
  repaint();
}
```

Painting and Repainting

- In general (as with other aspects of the Swing libraries), if we "do the right thing", then our interfaces should work as expected.
- If you want to provide a custom rendering for a component such as JPanel, override the paintComponent() method.
  - Don't explicitly call the paint() or paintComponent() methods
- If you want to ensure that the component is updated with new content, call repaint().
  - Don't override the repaint() method.
- If we follow these basic guidelines, then the dependencies between the various methods should be taken care of.

Aside: Swing Components

- With the material that we've seen so far, *could* actually define our own widgets from scratch. For example, for a button we'd need to:
  - Draw an appropriate shape
  - Keep a collection of ActionListeners
  - Add a MouseListener that listened for mouse clicks
  - Add a KeyListener that listened for key clicks
  - When the user clicks or presses send appropriate ActionEvents to the listeners.

More GUIs Summary

- Widgets
  - Menus Bars, Menus and MenuItems
  - JCheckBox
  - JRadioButton
  - JList
  - JCheckBox
  - JDialog
  - JFileChooser
- Listeners
  - Anonymous Listeners
  - KeyListener
  - MouseListener
  - WindowListener
  - Adapters
- Separating Model and GUI
  - Flexibility
  - Alternative interfaces
  - Update strategies
- Applets
  - Applications delivered over the Web and run in the Browser
  - Packaging classes as jars
  - Security considerations
- Graphics
  - Drawing simple shapes
  - Custom components.