Chapter 12

Graphical User Interfaces

Topics

- GUI Applications with *JFrame*
- GUI Components
- Labels
- Event Handling
- Text Fields
- Command Buttons, Radio Buttons, and Checkboxes
- Lists and Combo Boxes
- Adapter Classes
- Mouse Movements
- Layout Managers

Introduction

GUIs enable the user to

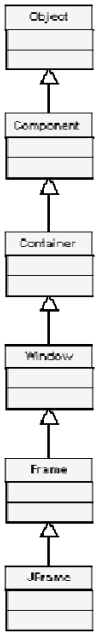
- select the next function to be performed
- enter data
- set program preferences, such as colors or fonts
- display information

GUIs also make the program easier to use

- a GUI is a familiar interface to users. Users can learn quickly to operate your program, in many cases, without consulting documentation or requiring extensive training.

The *JFrame* Class

- The *JFrame* class, in the *javax.swing* package, allows you to display a window.
- *JFrame* is a
  - *Component*, a graphical object that can be displayed
  - *Container*, a component that holds other objects
  - *Window*, a basic window
  - *Frame*, a framed window



JFrame Constructors

Constructor
`JFrame( )` constructs a *JFrame* object with no text in the title bar
`JFrame( String titleBarText )` constructs a *JFrame* object with *titleBarText* displayed in the window's title bar

Useful Methods of the *JFrame* Class

Return value	Method name and argument list
`Container`	`getContentPane( )` returns the content pane object for this window
`void`	`setDefaultCloseOperation( int operation )` sets the default operation when the user closes the window.
`void`	`setSize( int width, int height )` sizes the window to the specified *width* and *height* in pixels
`void`	`setVisible( boolean mode )` displays this window if *mode* is *true*; hides the window if *mode* is *false*

Applications extending *JFrame* inherit these methods.

A Shell GUI Application

See Example 12.1 ShellGUIApplication.java

Our application inherits from *JFrame*

```
public class ShellGUIApplication extends JFrame
```

The *main* method

- instantiates an instance of our application:

```
ShellGUIApplication basicGui =
        new ShellGUIApplication( );
```

- Specifies that the application should terminate when the user closes the window:

```
basicGui.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE );
```

A Shell GUI Application (con't)

The GUI application's constructor's job is to:

- call the constructor of the *JFrame* superclass
- get an object reference to the content pane container. We will add our GUI components to the content pane.
- set the layout manager. Layout managers arrange the GUI components in the window.
- instantiate each component
- add each component to the content pane
- set the size of the window
- make the window visible.

Common Error Trap

Be sure to call the *setSize* method to set the initial dimensions of the window and to call the *setVisible* method to display the window and its contents.

Omitting the call to the *setSize* method will create a default *JFrame* consisting of a title bar only.

If you omit the call to the *setVisible* method, the window will not open when the application begins.

GUI Components

A component performs at least one of these functions:

- displays information
- collects data from the user
- allows the user to initiate program functions.

The Java class library provides a number of component classes in the *javax.swing* package

AWT Versus Swing

Java supports two implementations of GUI components: AWT (Abstract Window Toolkit) and *swing*.

- AWT components:
  - the original implementation of Java components
  - AWT hands off some of the display and behavior of the component to the native windowing system
  - called heavyweight components
- Disadvantage:
  - because of the inconsistencies in the look-and-feel of the various windowing systems, an application may behave slightly differently on one platform than on another.

AWT Versus Swing

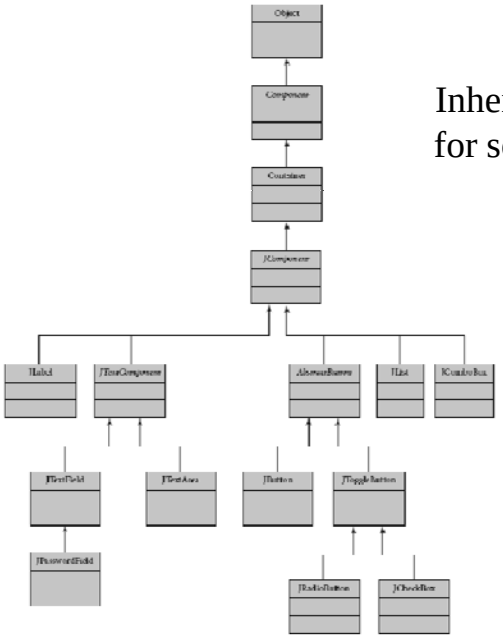
- Swing Components
  - second generation of GUI components
  - developed entirely in Java to provide a consistent look and feel from platform to platform
  - referred to as lightweight components
- Benefits:
  - Applications run consistently across platforms, which makes maintenance easier.
  - The *swing* architecture has its roots in the Model-View-Controller paradigm, which facilitates programming.
  - An application can still take on the look-and-feel of the platform on which it is running, if desired.

Java Swing Components, part 1

Component/ Java Class	Purpose
Label / *JLabel*	Displays an image or read-only text. Labels are often used to identify the contents of a text field.
Text field / *JTextField*	A single-line text box for displaying information and accepting user input
Text area / *JTextArea*	Multiple-line text field for data entry or display
Password field / *JPasswordField*	Single-line text field for accepting passwords without displaying the characters typed.

Java Swing Components, part 2

Component/ Java Class	Purpose
Button / *JButton*	Command button used to signal that an operation should be performed
Radio Button / *JRadioButton*	Toggle button used to select one option in the group.
Checkbox / *JCheckBox*	Toggle button used to select 0, 1, or more options in a group.
List / *JList*	List of items that the user clicks to select one or more items
Drop-down List / *JComboBox*	Drop-down list of items that the user clicks to select one item



Inheritance Hierarchy for some GUI Classes

Useful *JComponent* Methods

Return value	Method name and argument list
`void`	`setVisible( boolean mode )` makes the component visible if *mode* is *true*; hides the component if mode is *false*. The default is visible.
`void`	`setToolTipText( String toolTip )` sets the tool tip text to *toolTip*. When the mouse lingers over the component, the *toolTip* text will be displayed.
`void`	`setForeground( Color foreColor )` sets the foreground color of the component to *foreColor*.
`void`	`setBackground( Color backColor )` sets the background color of the component to *backColor.*

More Useful *JComponent* Methods

Return value	Method name and argument list
`void`	`setOpaque( boolean mode )` sets the component's background to opaque if *mode* is *true*; sets the component's background to transparent if *mode* is *false*. If opaque, the background is filled with the component's background color; if transparent, the background is filled with the background color of the container on which it is placed. The default is transparent.
`void`	`setEnabled( boolean mode )` enables the component if *mode* is *true*, disables the component if *mode* is *false*. An enabled component can respond to user input.

Useful *Container* Methods

We use these methods to set up the organization of the components and to add and remove components from the window:

Return value	Method name and argument list
`void`	`setLayout( LayoutManager mgr )` sets the layout manager of the window to *mgr*
`Component`	`add( Component component )` adds the *component* to the window, using the rules of the layout manager. Returns the argument.
`void`	`removeAll( )` removes all components from the window

The *FlowLayout* Layout Manager

The *FlowLayout* layout manager arranges components in rows from left to right in the order in which the components are added to the container.

Whenever a newly added component does not fit into the current row, the *FlowLayout* layout manager starts a new row.

***FlowLayout* Constructor**
`FlowLayout( )` creates a *FlowLayout* layout manager that centers components in the container.

The *JLabel* Component

A *JLabel* label component does not interact with a user

The *JLabel* displays some information, for example:

- a title
- an identifier for another component
- an image

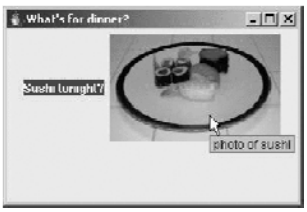
JLabel Constructors

Constructor
`JLabel( String text )` creates a *JLabel* object that displays the specified *text*
`JLabel( String text, int alignment )` creates a *JLabel* object that displays the specified *text.* The *alignment* argument specifies the alignment of the text within the label component. The *alignment* value can be any of the following *static int* constants from the *SwingConstants* interface: LEFT, CENTER, RIGHT, LEADING, or TRAILING. By default, the label text is left-adjusted.
`JLabel( Icon image )` creates a *JLabel* object that displays the *image*.

Using *JLabel* Components

- We instantiate two *JLabels:*
  - *labelText,* which displays text
    - We set the foreground and background colors and set the text to opaque.
  - *labelImage,* which displays an image.
    - We add some tooltip text

See Example 12.2 Dinner.java



Common Error Trap

As with any object reference, you must instantiate a component before using it.

Forgetting to instantiate a component before adding it to the content pane will result in a *NullPointerException* at run time when the JVM attempts to display the component.

Common Error Trap

Be sure to place the call to the *setVisible* method as the last statement in the constructor.

If you add components to the window after calling *setVisible*, those components will not be displayed until the window contents are refreshed, or repainted.

Event Handling

GUI programming uses an event-driven model of programming.

The program responds to events caused by the user interacting with a GUI component.

- For example, we might display some text fields, a few buttons, and a selectable list of items. Then our program will "sit back" and wait for the user to do something.
- When the user enters text into a text field, presses a button, or selects an item from the list, our program will respond, performing the function that the user has requested, then sit back again and wait for the user to do something else.

Handling Events

When the user interacts with a GUI component, the component fires an event.

To allow a user to interact with our application through a GUI component, we need to perform the following functions:

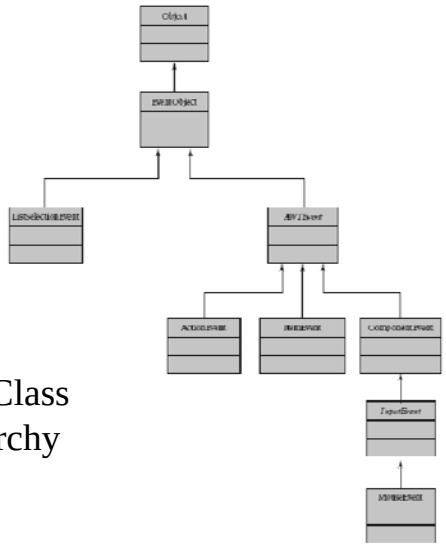
1. write an event handler class (called a listener)
2. instantiate an object of that listener
3. register the listener on one or more components

Note that an application can instantiate more than one listener.

Listener Interfaces

A typical event handler class *implements* a listener interface.

- The listener interfaces, which inherit from the *EventListener* interface, are supplied in the *java.awt.event* or *javax.swing.event* package.
- A listener interface specifies one or more *abstract* methods that an event handler class needs to override.
- The listener methods receive as a parameter an event object, which represents the event that was fired.



Event Class Hierarchy

Event Objects

Event classes are subclasses of the *EventObject* class and are in the *java.awt.event* and *javax.swing.event* packages.

From the *EventObject* class, event classes inherit the *getSource* method.

Return value	Method name and argument list
`Object`	`getSource( )` returns the object reference of the component that fired the event

Using simple *if .. else if* statements, your event handler can identify which of its registered components fired an event.

Events and Listeners

JComponent	User Activity	EventObject	Listener Interface to Implement
`JTextField`	Pressing *Enter*	*`ActionEvent`*	*`ActionListener`*
`JButton`	Pressing the button	*`ActionEvent`*	*`ActionListener`*
`JRadioButton`	Selecting a radio button	*`ItemEvent`*	*`ItemListener`*
`JCheckBox`	Selecting a checkbox	*`ItemEvent`*	*`ItemListener`*

More Events and Listeners

JComponent	User Activity	Event Object	Listener Interface to Implement
`JList`	Selecting an item	`ListSelection-Event`	`ListSelection-Listener`
`JComboBox`	Selecting an item	`ItemEvent`	`ItemListener`
`Any component`	Pressing or releasing mouse buttons	`MouseEvent`	`MouseListener`
`Any component`	Moving or dragging the mouse	`MouseEvent`	`MouseMotion-Listener`

Pattern for Event Handling

Constructor:

```
public ClassName( ) // constructor
{
   // call JFrame constructor
   // get content pane
   // set the layout manager
   // instantiate components
   // add components to the content pane
   // instantiate event handler objects
   // register event handlers on components

   // set window size
   // make window visible
}
```

Registering a Listener

In the constructor, we instantiate an object of our event handler class. Then we register that event handler on a component by calling an *add…Listener* method.

***add…Listener* APIs**
`void addActionListener( ActionListener handler )`
`void addItemListener( ItemListener handler )`
`void addListSelectionListener( ListSelectionListener handler )`
`void addMouseListener( MouseListener handler )`
`void addMouseMotionListener( MouseMotionListener handler )`

Common Error Trap

If you do not register a listener on a component, the user will still be able to interact with the component (type into a text field, for example), but events generated by the user interaction will not be sent to the listener.

Thus, even though you provide an event handler, if you do not register it on a component, the event handler will never execute.

Event Handler Pattern

An event handler as a *private* inner class:

```
private class EventHandlerName
              implements ListenerName
{
  // implement the listener interface methods
  // to process the events
}
```

A *private* inner class is defined within the *public* class and has access to all the members of the *public* class.

When an event is fired on a registered component, the appropriate listener method executes automatically.

Text Fields

We will write a GUI program that simulates a login, using these components:

JTextField (a single line of text) for the User ID

JPasswordField (a single line of text that echoes a special character for each character typed by the user) for the password

JTextArea (multiple lines of text) to display a legal warning to potential hackers

JLabels to label the text fields.

See Example 12.3 Login.java

JTextField and *JPasswordField* Constructors

JTextField
`JTextField( String text )` creates a text field initially filled with *text*
`JTextField( int numberColumns )` constructs an empty text field with the specified number of columns
JPasswordField
`JPasswordField( int numberColumns )` constructs an empty password field with the specified number of columns

JTextArea Constructors

JTextArea
`JTextArea( String text )` constructs a text area initially filled with *text*
`JTextArea( int numRows, int numColumns )` constructs an empty text area with the number of rows and columns specified by *numRows* and *numColumns*
`JTextArea( String text, int numRows, int numColumns )` constructs a text area initially filled with *text*, and with the number of rows and columns specified by *numRows* and *numColumns*

Methods Common to *JTextField*, *JTextArea*, and *JPasswordField*

Return value	Method name and argument list
`void`	`setEditable( boolean mode )` sets the properties of the text component as editable or noneditable, depending on whether *mode* is *true* or *false*. The default is editable.
`void`	`setText( String newText )` sets the text of the text component to *newText*
`String`	`getText( )` returns the text contained in the text component

Additional Methods of the *JPasswordField* Class

Return value	Method name and argument list
`void`	`setEchoChar( char c )` sets the echo character of the password field to *c*
`char[]`	`getPassword( )` returns the text entered in this password field as an array of *chars*. Note: this method is preferred over the *getText* method for retrieving the password typed by the user.

The *ActionListener* Interface

An event handler that implements the *ActionListener* interface provides code in this method to respond to the *ActionEvent* fired by any registered components.

```
public void actionPerformed( ActionEvent event )
```

Common Error Trap

Be sure that the header of the listener method you override is coded correctly. Otherwise, your method will not override the *abstract* method, as required by the interface.

For example, misspelling the *actionPerformed* method name as in this header:

```
public void actionperformed( ActionEvent a )
```

will generate a compiler error:

```
Login.TextFieldHandler is not abstract and does
not override abstract method
actionPerformed(ActionEvent)in ActionListener
```

Common Error Trap

The *java.awt.event* package is not imported with the *java.awt* package.

You will need both of these *import* statements:

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

JButton

A *JButton* component implements a command button.

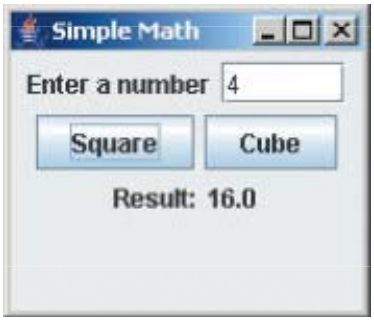
***JButton* Constructor**
`JButton( String buttonLabel )` creates a command button labeled with *buttonLabel*

Clicking on a button generates an *ActionEvent*, so our listener should implement the *ActionListener* interface.

JButton Example

See Example 12.4 SimpleMath.java

The user enters a number into the text field and presses a button. The result is displayed as a *JLabel*.



Our event handler uses the *getSource* method to determine which button was pressed.

JRadioButton and *JCheckBox*

Radio buttons are typically used to allow the user to select one option from a group.

- Radio buttons are meant to be mutually exclusive, in that clicking on any radio button deselects any previously selected radio button.

Checkboxes often are associated with the sentence "check all that apply;" that is, the user may select 0, 1, or more options.

- A checkbox is a toggle button, in that successive clicks alternate between selecting and deselecting the option for that particular checkbox.

Creating a Group of *JRadioButtons*

To create a group of mutually exclusive radio buttons, we first instantiate the buttons:

***JRadioButton* Constructors**
`JRadioButton( String buttonLabel )` constructs a radio button labeled *buttonLabel*. By default, the radio button is initially deselected.
`JRadioButton( String buttonLabel, boolean selected )` constructs a radio button labeled *buttonLabel*. If *selected* is *true*, the button is initially selected; if *selected* is *false*, the button is initially deselected.

The *ButtonGroup* Class

Then we instantiate a *ButtonGroup* and add the buttons to the group.

- A *ButtonGroup* object is used to define a mutually exclusive group of buttons.

Constructor
`ButtonGroup( )` constructs a button group

Return value	**Method name and argument list**
`void`	`add( AbstractButton button )` adds *button* to the button group. *JRadioButton* inherits from *AbstractButton*.

The *ItemListener* Interface

Clicking on a registered *JRadioButton* or a *JCheckBox* generates an *ItemEvent*, which is handled by an *ItemListener*.

- An event handler that implements the *ItemListener* interface provides code in this method to respond to the *ItemEvent* fired by any registered components.

```
public void itemStateChanged( ItemEvent event )
```

Using Radio Buttons

See Example 12.5 ChangingColors.java

The *red*, *green*, and *blue* *JRadioButtons* are added to a *ButtonGroup*.

The listener uses the *getSource* method to determine which radio button was pressed and sets the background of a *JLabel* to the selected color.

A Useful *ItemEvent* Method

Each click (select and deselect) on a registered *JCheckBox* generates an *ItemEvent*.

To distinguish between these states, we call the *getStateChange* method of the *ItemEvent* class.

Return value	Method name and argument list
`int`	`getStateChange( )` If the checkbox is selected, the value *SELECTED* is returned; if the checkbox is deselected, the value *DESELECTED* is returned, where *SELECTED* and *DESELECTED* are *static int* constants of the *ItemEvent* class.

JCheckBox Constructors

Because *JCheckBoxes* are designed to allow selection of multiple check boxes simultaneously, we do not use a button group.

***JCheckBox* Constructors**
`JCheckBox( String checkBoxLabel )` constructs a check box labeled *checkBoxLabel*. By default, the check box is initially deselected.
`JCheckBox( String checkBoxLabel, boolean selected )` constructs a checkbox labeled *checkBoxLabel.* If *selected* is *true*, the checkbox is initially selected; if *selected* is *false*, the checkbox is initially deselected.

Using *JCheckBoxes*

See Example 12.6 MixingColors.java

Using the *getStateChange* method, the listener sets the red, green, and blue color intensities depending on whether the checkbox is selected or deselected.

The *JList* Component

The *JList* component displays a list of items.

- The user can select one or more items from the list, depending on the selection mode.
- When an item is selected on a registered list, a *ListSelectionEvent* is generated.
- The listener for this event *implements* the *ListSelectionListener* interface, which has one method:

```
public void valueChanged( ListSelectionEvent e )
```

JList Constructor and Methods

Constructor
`Jlist<E>( E[] arrayName )` constructs a *JList* initially filled with the objects of type *E* in *arrayName*. Often, the objects are *Strings*.

Return value	Method name and argument list
`int`	`getSelectedIndex( )` returns the index of the selected item. The index of the first item in the list is 0.
`void`	`setSelectedIndex( int index )` selects the item at *index*. The index of the first item in the list is 0.

Another *JList* Method

Return value	Method name and argument list
`void`	`setSelectionMode( int selectionMode )` sets the number of selections that can be made at one time. The following *static int* constants of the *ListSelectionModel* interface can be used to set the selection mode: SINGLE_SELECTION – one item can be selected SINGLE_INTERVAL_SELECTION – multiple contiguous items can be selected MULTIPLE_INTERVAL_SELECTION – multiple intervals of contiguous items can be selected (This is the default.)

Using a *JList*

See Example 12.8 FoodSamplings.java

The *JList* items are an array of *Strings*. We also define a parallel array of *ImageIcons* with images corresponding to the country names.

Initially, we programmatically select the first item using the *setSelectedIndex* method and display the first image.

SOFTWARE ENGINEERING TIP

Arrange items in lists in a logical order so that the user can find the desired item quickly.

For example, list items alphabetically or in numeric order.

Also consider placing the most commonly chosen items at the top of the list.

The *JComboBox* Component

The *JComboBox* implements a drop-down list.

- When the combo box appears, one item is displayed, along with a down-arrow button.
  - When the user presses the button, the combo box "drops" open and displays a list of items, with a scroll bar for viewing more items.
  - The user can select only one item from the list.
  - When the user selects an item, the list closes and the selected item is the one item displayed.
- A *JComboBox* fires an *ItemEvent*, so the event handler implements the *ItemListener* interface.

JComboBox Constructor/Methods

Constructor	
`JComboBox<E>( E[] arrayName )` constructs a *JComboBox* initially filled with the objects of type *E* in *arrayName*. Often, the objects are *Strings*.	
Return value	**Method name and argument list**
`int`	`getSelectedIndex( )` returns the index of the selected item. The index of the first item in the list is 0.
`void`	`setSelectedIndex( int index )` selects the item at *index*. The index of the first item in the list is 0.
`void`	`setMaximumRowCount( int size )` sets the number of items visible at one time.

Example Using *JComboBox*

- Our GUI application builds a *JComboBox* dynamically.
- The file *specials.txt* contains information about vacation specials.
- The *Vacation* class defines instance variables for vacation specials.
- *The VacationList* class reads *specials.txt*, instantiates *Vacation* objects, and adds them to an *ArrayList*.
- Our application retrieves the vacation info from the *VacationList* class and uses the returned array to create the items in the *JComboBox* dynamically.

See Examples 12.9 Vacation.java, Example 12.10 VacationList, and Example 12.11 VacationSpecials.java

Mouse Events

For mouse events, there are two listeners:

- the *MouseListener* interface specifies five methods to implement
- the *MouseMotionListener* interface specifies two methods to implement

- If we want to use a *MouseListener*, but need to use only one of its five methods to process a *MouseEvent*, we still have to implement the other four methods as "do-nothing" methods with empty method bodies.

Adapter Classes

For convenience, Java provides adapter classes, each of which *implements* an interface and provides an empty body for each of the interface's methods.

- Thus, instead of implementing an interface, we can extend the appropriate adapter class and override only the method or methods we need.
- For mouse events, the adapter classes are
  - *MouseAdapter*, which implements the *MouseListener* interface
  - *MouseMotionAdapter*, which implements the *MouseMotionListener* interface

MouseEvents

Any mouse activity (clicking, moving, or dragging) by the user generates a *MouseEvent.*

- To determine the (x, y) coordinate of the mouse event, we call these *MouseEvent* methods:

Return value	Method name and argument list
`int`	`getX( )` returns the x value of the (x, y) coordinate of the mouse activity
`int`	`getY( )` returns the y value of the (x, y) coordinate of the mouse activity

MouseListener Interface Methods

`public void mousePressed( MouseEvent e )` called when the mouse button is pressed
`public void mouseReleased( MouseEvent e )` called when the mouse is released after being pressed
`public void mouseClicked( MouseEvent e )` called when the mouse button is pressed and released
`public void mouseEntered( MouseEvent e )` called when the mouse enters the registered component
`public void mouseExited( MouseEvent e )` called when the mouse exits the registered component

Using the *MouseAdapter* Class

We implement a Submarine Hunt game.

- A submarine is hidden somewhere in the window, and the user will try to sink the submarine by clicking the mouse at various locations in the window, simulating the dropping of a depth charge.

- The only mouse action we care about is a click; therefore, we are interested in overriding only one method of the *MouseListener* interface: *mouseClicked*. To simplify our code, we extend the *MouseAdapter* class.
- The listener should handle mouse clicks anywhere in the window, so we register the *MouseListener mh* on the window (*JFrame*) component:

```
this.addMouseListener( mh );
```

Removing a Listener

In the mouse handler, if the mouse click has hit the submarine, we "unregister" the mouse listener using the following method inherited from the C*omponent* class. After doing so, further mouse clicks will no longer cause the handler to be called.

Return value	Method name and argument list
`void`	`removeMouseListener( MouseListener mh )` removes the mouse listener *mh* so that it is no longer registered on this component.

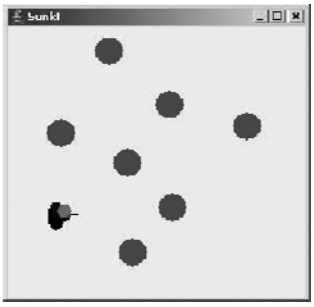
Updating the Window

- Two cases require us to update the window:
  - When the submarine has been hit, we want to draw the "sunken" submarine .
  - If the mouse click is more than two lengths from the center of the submarine, we want to draw a blue circle.
- However, we cannot call the *paint* method explicitly. Instead, we call the *repaint* method (inherited from the *Component* class):

Return value	Method name and argument list
`void`	`repaint( )` automatically forces a call to the *paint* method

Separating the GUI from the Functionality

We first create a class, *SubHunt*, that encapsulates the game functionality – creates the game, enables play, enforces the rules.



The client of this class, *SubHuntClient*, provides the user interface.

See Example 12.12 SubHunt.java
and Example 12.13 SubHuntClient.java

A Treasure Hunt Game

We implement a treasure hunt game:

- we hide a treasure in the window.
- the user attempts to find the treasure by moving the mouse around the window.
- we indicate how close the mouse is to the treasure by printing a message at the mouse location.
- when the user moves the mouse over the treasure, we draw the treasure and remove the listener to end the game.

We Use the *MouseMotionListener* Interface

`public void mouseMoved( MouseEvent e )` called when the mouse is moved onto a registered component
`public void mouseDragged( MouseEvent e )` called when the mouse is pressed on a registered component and the mouse is dragged

Instead of coding the event handler as a *private* inner class, we define our application class as implementing the *MouseMotionListener* interface. As a result, our application is a listener, and we register the listener on itself:

```
this.addMouseMotionListener( this );
```

Implementing the Game

The *TreasureHunt* class encapsulates the treasure hunt game, and the *TreasureHuntClient* class implements the user interface.

See Example 12.14 TreasureHunt.java
and Example 12.15 TreasureHuntClient.java

Layout Managers

Layout Managers determine how the components are organized in the window.

- Three Layout Managers are:
  - *FlowLayout*
    - Adds components left to right in rows
  - *GridLayout*
    - Adds components to a table-like grid with equally sized cells
  - *BorderLayout*
    - Adds a component to any of five predefined areas

GridLayout

The *GridLayout* organizes the container as a grid.

- We can visualize the layout as a table made up of equally sized cells in rows and columns.
- Each cell can contain one component

- The first component added to the container is placed in the first column of the first row; the second component is placed in the second column of the first row, and so on. When all the cells in a row are filled, the next component added is placed in the first cell of the next row.

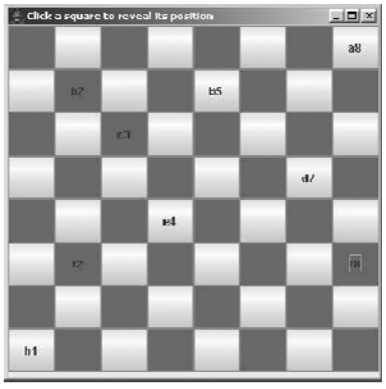
GridLayout Constructors

***GridLayout* Constructors**
`GridLayout( int numberRows, int numberColumns )` creates a grid layout with the number of rows and columns specified by the arguments.
`GridLayout( int numberRows, int numberColumns, int hGap, int vGap )` creates a grid layout with the specified number of rows and columns and with a horizontal gap of *hGap* pixels between columns and a vertical gap of *vGap* pixels between rows. Horizontal gaps are also placed at the left and right edges, and vertical gaps are placed at the top and bottom edges.

Using *GridLayout*

See Example 12.16 ChessBoard.java

We implement the chessboard as a two-dimensional array of *JButtons*. When a button is pressed, we display the corresponding text from a parallel, two-dimensional array of *Strings*.



Dynamic Layouts

- Layout managers can be instantiated dynamically based on run-time parameters or user input.
- Layouts also can be changed at run time.

We randomly generate the grid size before each new game.

See Example 12.17 TilePuzzle.java
and Example 12.18 TilePuzzleClient.java

BorderLayout

A *BorderLayout* organizes a container into five areas:

NORTH		
WEST	CENTER	EAST
SOUTH		

- Each area can hold at most one component.
- The size of each area expands or contracts depending on:
  - the size of the component in that area
  - the sizes of the components in the other areas
  - whether the other areas contain a component.

Using a *BorderLayout*

***BorderLayout* Constructors**
`BorderLayout( )` creates a border layout with no gaps between components
`BorderLayout( int hGap, int vGap )` creates a border layout with a horizontal gap of *hGap* pixels between components and a vertical gap of *vGap* pixels between components.

BorderLayout is the default layout manager for a *JFrame* object.

- so if we want to use a border layout for our GUI applications, we do not need to instantiate a new layout manager.

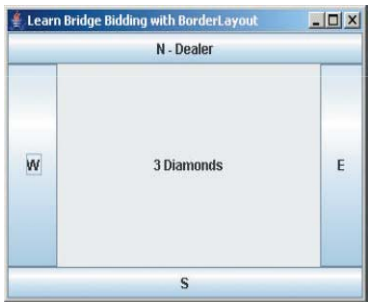
Adding Components to a *BorderLayout*

To add components to a container with a *BorderLayout* layout manager, we use this method inherited from the *Container* class.

Return value	Method name and argument list
`void`	`add( Component c, Object borderlayoutArea )` adds the component *c* to the container. The area defined by *borderlayoutArea* can be specified using any of the following *static String* constants from the *BorderLayout* class: NORTH, SOUTH, EAST, WEST, CENTER

Using a BorderLayout

We place a *JButton* in each of the NORTH, EAST, WEST, and SOUTH areas, and a *JLabel* in the CENTER area.



See Example 12.19 BridgeBidding.java
and Example 12.20 BridgeBiddingClient.java

Nesting Components

Components can be nested.

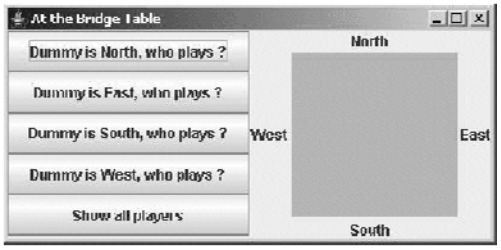
- Because the *JComponent* class is a subclass of the *Container* class, a *JComponent* object is a *Container* object as well. As such, it can contain other components.
- We can use this feature to nest components to achieve more precise layouts.

Using *JPanels* to Nest Components

The *JPanel* class is a general-purpose container, or a panel, and is typically used to hold other components.

- We usually place several components into a *JPanel*, then add the *JPanel* to the container as a single component.
- Each *JPanel* has its own layout manager, and the content pane for the *JFrame* application has its own layout manager.
- We can even have multiple levels of nesting.

Example 12.21 BridgeRules.java



- The content pane has a *GridLayout* layout (1 row, 2 columns)
- We define a *JPanel* managed by a *GridLayout* (5 rows, 1 column). We add five buttons to the *JPanel* and add the *JPanel* to column 1 of the content pane.
- We define a second *JPanel* managed by a *BorderLayout*. We put a component into each of the five *BorderLayout* areas and add the *JPanel* to column 2 of the content pane.