Java Chapter 12 Part 2

* GUI Programming: Layout, More Components, and Design
* CIS 255 • Shelby-Hoover Campus

Layout Managers

* A **layout manager** is an object that allows a container to place GUI components in specific positions
* The package java.awt.\* provides several layout manager classes, including the following:
  + FlowLayout: arranges elements in rows left-to-right (the default layout manager for JPanel)
  + BorderLayout: divides a container into five regions (north, south, east, west, center) (the default layout manager for the content pane of JFrame)
  + GridLayout: divides a container into a grid with rows and columns
* To apply a layout to a JFrame or JPanel object, invoke the method setLayout() with a layout manager object as the argument

FlowLayout

* The FlowLayout manager is similar to “word wrap”: it adds components to the current row until there is no more room, then starts a new row
* If the user resizes the window, the program may “reflow” the rows to move a component to a row that will accommodate its width
* The FlowLayout class contains several constructors, including a no-arg constructor and constructors with the following parameters:
  + Alignment (e.g., FlowLayout.LEFT)
  + Alignment, Horizontal Gap (pixels between elements on the same row), and Vertical Gap (pixels between rows)
* As mentioned earlier, a JPanel object uses FlowLayout as its layout manager by default

BorderLayout

* The BorderLayout manager divides a container into regions, where each region can contain only one GUI component
* The add method call for a container with a BorderLayout applied should include a second argument specifying the region (e.g., BorderLayout.NORTH); the default value (if omitted) is BorderLayout.CENTER
* The program resizes the component added to a region to fill the entire space for that region
* The constructors for BorderLayout include a no-arg constructor and one with horizontal and vertical gap arguments
* BorderLayout is the default layout manager for JFrame (and its subclasses)
* Example: In BorderWindow.java (Code Listing 12-10), the program adds a JButton to each of the five regions of the window, resizing each button to fill the entire region

BorderLayout and JPanel

* Each region in a container with a BorderLayout holds only one component, resized to fill the entire region
* If a program adds another component to the same region, it replaces the original component
* A better way to place components in each region is to use an intermediate JPanel object in each region
  + Each panel can hold multiple GUI components
  + Although the program resizes the panel, it does not resize the components contained within the panel
  + Example: In BorderPanelWindow.java (Code Listing 12-11), each JButton is contained within a JPanel

The pack Method

* In earlier examples, the setSize method gives the window an exact width and height in pixels
* The pack() method automatically sizes the window to hold the GUI components it contains
* In BorderPanelWindow, the window is large enough to hold the buttons within the panels, with a little space around each button

GridLayout

* The GridLayout manager is similar to a table or spreadsheet: it divides the container into rows and columns of cells
* Each cell holds only one component (but that component can be a container)
* All of the cells will have the same size (large enough to fit the largest contents)
* The program resizes smaller components to fill the space required by the largest component
* The constructor for a GridLayout requires two arguments: the number of rows and columns
* Components are added row by row, left to right and top to bottom (the program cannot specify the cell)
* Examples: GridWindow.java (Code Listing 12-12) and GridPanelWindow.java (Code Listing 12-13)

Radio Buttons

* When a window contains a group of **radio buttons**, the user may select only one button in that group at a time
* Two of the constructors for the class JRadioButton include these parameters:
  + A String for the text that appears next to the button
  + The text label plus a second boolean argument indicating the button’s initial selection status
* The program must add each JRadioButton object to two objects:
  + A ButtonGroup object ensures that the user may select only one JRadioButton in that group at a time
  + The display container (the content pane or a panel)

Responding to Radio Buttons

* To respond to a user’s click on a JRadioButton object, a program adds a listener object from a class that implements the ActionListener interface (same as for JButton)
* JRadioButton provides two methods to obtain or modify the status of a radio button:
  + isSelected() returns a boolean value indicating the current status of the button
  + doClick() selects the button as if the user had clicked it
* Example: In MetricConverter.java (Code Listing 12-14), the user may select one of three conversion options

Check Boxes

* A set of **check boxes** allows a user to make multiple selections from a set of options
* Two of the constructors for JCheckBox objects mirror those of JRadioButton (the option text is required; the selection status is optional)
* JCheckBox objects also support isSelected() and doClick()
* Since clicking on a check box may deselect an already-selected check box, a program uses the isSelected() method to determine whether the click selected or deselected the check box

ItemEvent and ItemListener

* When the user clicks a JCheckBox object, the program generates an ItemEvent object
  + A listener class intended to respond to this click should implement the interface ItemListener
  + The method required by the ItemListener interface is itemStateChanged, a public void method with an ItemEvent parameter
  + Register the listener with a check box using the method addItemListener
* Example: In ColorCheckBoxWindow.java (Code Listing 12-15), the listener method itemStateChanged detects which JCheckBox fired the ItemEvent object; it then obtains that JCheckBox object’s current state to toggle the colors

Borders

* A program may enclose a panel within a border
* There are several different border styles, each created by invoking a static method from the class BorderFactory in the package javax.swing (Table 12-6)
  + createEmptyBorder(top, bottom, left, right)  
    adds empty space in the amounts of pixels specified in the arguments
  + createLineBorder(color, thickness)  
    draws a line around the panel with the color and thickness (in pixels)
  + createTitledBorder(title)  
    creates an etched border around the panel with the specified title as a label in the upper-left corner of the border
* Invoke the method setBorder on the panel being bordered, with a call to the static method for that border style as the argument
* Example:   
    
  panel.setBorder(BorderFactory.createLineBorder(Color.RED, 1));

GUI Problem Solving

* A window may contain several complex panels
* It may not be wise to place the logic for every panel within a single window class
* Another approach is to create a separate class that extends JPanel for each panel to be incorporated into the window
* The window class (extending JFrame) then contains an object of each panel class type

GUI Problem Solving Example

* An order calculator for a bagel shop is broken into five panels:
  + GreetingPanel.java (Code Listing 12-16) contains a single label with a greeting
  + BagelPanel.java (Code Listing 12-17) contains two radio buttons to indicate the choice of bagel (white or wheat) and a public method to return the cost of the selected bagel
  + ToppingPanel.java (Code Listing 12-18) contains four check boxes for each of four toppings that can be added to a bagel; a public method in the class returns the sum of the costs of the selected toppings
  + CoffeePanel.java (Code Listing 12-19) contains four radio buttons to indicate the choice of coffee (none, regular, decaf, or cappuccino) and a public method to return the cost of the selected coffee choice
  + The last panel, with two buttons, is contained within the window class itself
* OrderCalculatorGUI.java (Code Listing 12-20) instantiates each of the four separate panel classes and contains two listener classes
  + The first listener, for the “Calculate” button, uses the public methods from the panels to calculate the sum of the various options, the tax, and the final total
  + The second listener exits the program if the user clicks the “Exit” button

Splash Screens

* With Java 6 and later, the command-line statement to execute a program may include a parameter that specifies a graphics file to be used as a splash screen
  + Form: java –splash:GraphicFile ClassName
  + Example: java –splash:BrandiLogo.jpg Bagel
* This parameter supports the graphic file types GIF, PNG, and JPEG
* The splash screen graphic remains visible until the initial program window is ready (helpful when a program may require a lengthy load time)

GUI Programs and the Console

* A program’s interaction should be limited to one interface or the other; mixing console I/O and GUI window I/O could be confusing to the user
* Debugging a GUI program could be difficult, as the layout of the window itself should not be modified to include space for feedback messages
* A program should use the console to display diagnostic messages to assist a programmer in isolating and resolving program issues
* A programmer should remove these console output statements prior to releasing the program

Reminders for Chapter 12 (1)

* GUI applications require classes from several different packages (javax.swing, java.awt, java.awt.event) that must be imported properly
* The class for window objects is JFrame, but many GUI programs involve window classes that extend JFrame using inheritance
* A program must set up a window’s attributes (title, size, default close operation) before making it visible; this can be done in the constructor for the derived class
* A program must add GUI components to a container (a frame or a panel) to make them visible

Reminders for Chapter 12 (2)

* The user’s interaction with a GUI component will not trigger action unless there is an event listener object registered to it
* A program often includes an event listener class as a private inner class nested within the window class (available only to the class in which it is nested)
  + The event listener class must implement the interface required for the specific type of event generated by the GUI component (e.g., ActionListener, ItemListener)
  + When a class implements an interface, the method(s) required by that interface must have the exact header(s) required by that interface
* A program registers a new event listener object to each GUI component that could fire an event by invoking the appropriate add method (e.g., addActionListener, addItemListener) on that GUI component object

Reminders for Chapter 12 (3)

* A program may set one of two colors (the foreground and the background) for each GUI component to one of several Color class constants available in Java
* Making changes to a window’s content pane requires a call to getContentPane()
* A class does not require a separate event listener class for every GUI component: the methods getActionCommand() and getSource() detect which component created an event object
* A program must add each JRadioButton object to a ButtonGroup object (to restrict the number of buttons that can be selected) as well as to the container that displays the radio buttons
* A user’s click on a JCheckBox may select or deselect the box; use isSelected() to determine the selection status

Reminders for Chapter 12 (4)

* Some of the layout managers automatically resize the component in each region to the size of the region, and each region may be limited to one component
* To maintain the original size of the components and / or to place multiple components in each region, use an intermediate JPanel object for each region
* When adding a component to a container with a BorderLayout, specify the name of the region in which to place the component
* A program adds components to a container with a GridLayout left-to-right, first row to last row (not to specific cells)
* A program uses static BorderFactory methods to create borders as arguments to the method setBorder