# CIS 3145 Class Notes: Text Chapter 22

## Graphical User Interface (GUI) controls, GridBayLayout, and custom dialogs

**Objectives**

* Describe and use label and text fields, plus other Swing controls
* Describe and use GridBagLayout manager
* Describe and use custom dialogs

**GUI Controls**

We think of GUI controls as being objects, and as such the objects have *properties* and *methods*. Most properties have both a **get** and **set** method that is used to **read** the value or **change** the value of the property (see chapter 4). A list of get and set statements is really a list of properties for that control.

The **Text Field** control is where users can input data. The most important property for this control is the *text* property. Other properties include the *Columns*, *TootTipText*, *SelectedText*, *Size*, *Editable*, and *Enabled*. Some of the properties do not use the get method. For example *Editable* and *Enabled* have an ‘is’ method: isEditable() and isEnabled() which return a Boolean value.

The Text Field constructor can take an integer argument (# of columns), a text argument (text that will be displayed), or both.

A **Label** control is like the **Text Field** control but it only displays output with the *text* property, and no input is allowed. It does not have the *Editable* property. The **Label** does have an *Enabled* property so that we can control whether or not the user can copy from the label.

The **Text Area** control is used in place of the **Text Field** control when we want the user to see more that a single line or field of text. It has similar methods and properties such as *getText*, *setText*, and *text*. Because it has more room for text the **Text Area** also has the *lineWrap*, *rows*, and *wrapStyleWord* properties. The **Text Area** will also automatically add horizontal and vertical scroll bars if there is more text than will fit in the Text Area.

There are other controls that the book does not talk about which allow for user input. The **Check Box** and **Radio Button** controls are very similar. They both have the *selected* property which is Boolean, and the *setSelected* method which takes a Boolean parameter. Both will change the state of the button as displayed on the screen. Typically, these buttons are used in the code of other buttons to check the selection a user has made, using the isSelected methods. The **isSelected()** method, which returns a Boolean, dynamically determines in the code if the control is selected. In addition, the ‘*StateChanged*’ or ‘*actionPerformed*’ **events** can be used to respond when the user changes a radio button or checkbox.

The **Radio button** is always part of a group of buttons thus the ***buttonGroup*** property must be set when creating an interface with radio buttons. Because they are in a group it is best to use “If…Else If” statements for Radio Buttons: each radio button added to a group gets another Else If branch added to the statement and the if statement checking stops once a match is found.

The **Check Box** is coded differently from the **Radio Button** and will only use an independent If statement. Each **Check Box** gets its own separate If statement.

The **Combo Box** control is a combination of a **List Box** and **Text Field**. However, the ***editable*** property can make it so that the **Combo Box** does not accept user input. The **model** property can be changed in the code with the *addItem*() method, typically in the forms constructor sub procedure, to populate the elements displayed in the **Combo Box**. Often data displayed in a **combo box** will come from a database.

Only **one** item can be selected in a **Combo Box** and the *getSelectedItem*() will return the selected item displayed, while the *getSelectedIndex*() method returns an integer value of the selected item. The elements displayed in the **Combo Box** arelike an array and are zero based (the first element displayed has a value of 0). An index value of -1 means that there are no items selected.

The **List** control does not allow for user input but does allow the user to select **one** or **more** elements of the items displayed, depending on how the **selectionMode** property is set. Like the **Combo Box** the **List** has a **model.** The model can be set in code by using the **List’s** *setModel*() method. Before using this method create a special **model** variable of the *DefaultListModel* class type and use the ***addElement***() method to add items to the model. The *DefaultListModel* class is imported from the javax.swing.DefaultListModel class.

**GridBagLayout Manager**

Compared to the Border Layout or Flow Layout Manager the **GridBagLayout** manager allows for much greater flexibility in setting up controls on a form. In a GridLayout manager each cell of the rectangular grid is the same size while the GridBayLayout allows us to customize the size and behavior of each cell. The customization is done by setting the values of a **GridBagConstraints** object.

After setting the layout of a panel to the GridBagLayout manager, each control is added to the panel with the panel’s add method. The add method can take a single **control** as an argument, or a **control** and **constraint** object as two arguments.

The **GridBagConstraints** object uses 11 public field variables to specify how components are displayed in the GridBagLayout. The **gridx** and **gridy** fields are used to determine which cell in the layout will hold the control. The **anchor** field is similar to text alignments. The LINE\_START anchor value puts the control on the left side of the cell, while LINE\_END puts the control on the right side of the cell.

The cells of the grid can be combined (like merge in Excel) by using the **gridwidth** and **gridheight** properties of the GridBagConstraint class. A component with a gridwidth of 3 will span 3 consecutive cells.

The **insets** **field** takes an **Insets** **object** which will specify the top, left, bottom, and right number of pixels that will pad the control in the cell.

**Creating Custom Dialog Forms**

The previous chapter discussed two pre-build dialog boxes created with the *showMessageDialog* and *showConfirmDialog* methods. Setting the arguments for these methods creates a dialog box that will inform the user or get input based on which button is selected. We can also create our own dialog boxes by creating a class that extends the JDialog box. We can add whatever controls we want to these custom dialog forms. Similar to frames, we can specify the layout for the dialog form and either add controls directly to the form or add pre-defined panels that have controls placed into the panel. Like the pre-build dialog boxes, the custom boxes can be displayed as modal, meaning that the dialog form must be closed before returning to the parent frame.

We send data **to** a dialog form through the constructor by adding an input argument. We send data **from** a dialog form to the parent frame by creating a public method in the frame that is called from the dialog.