Chapter 12:
A First Look at GUI Applications

Starting Out with Java:
From Control Structures through Objects

Fifth Edition

by Tony Gaddis
Chapter Topics

Chapter 12 discusses the following main topics:

- Introduction
- Creating Windows
- Equipping GUI Classes with a main method
- Layout Managers
- Radio Buttons and Check Boxes
- Borders
- Focus on Problem Solving: Extending Classes from JPanel
Introduction

- Many Java applications use a **graphical user interface** or **GUI** (pronounced “gooey”).
- A GUI is a graphical window or windows that provide interaction with the user.
- GUI’s accept input from:
  - the keyboard
  - a mouse.
- A window in a GUI consists of **components** that:
  - present data to the user
  - allow interaction with the application.
Introduction

• Some common GUI components are:
  – buttons, labels, text fields, check boxes, radio buttons, combo boxes, and sliders.
JFC, AWT, Swing

- Java programmers use the *Java Foundation Classes (JFC)* to create GUI applications.
- The JFC consists of several sets of classes, many of which are beyond the scope of this book.
- The two sets of JFC classes that we focus on are AWT and Swing classes.
- Java is equipped with a set of classes for drawing graphics and creating graphical user interfaces.
- These classes are part of the *Abstract Windowing Toolkit (AWT)*.
JFC, AWT, Swing

- The AWT allows creation of applications and applets with GUI components.
- The AWT does not actually draw user interface components on the screen.
- The AWT communicates with a layer of software, \textit{peer classes}.
- Each version of Java for a particular operating system has its own set of peer classes.
JFC, AWT, Swing

• Java programs using the AWT:
  – look consistent with other applications on the same system.
  – can offer only components that are common to all the operating systems that support Java.
• The behavior of components across various operating systems can differ.
• Programmers cannot easily extend the AWT components.
• AWT components are commonly called *heavyweight components*. 
JFC, AWT, Swing

- Swing was introduced with the release of Java 2.
- *Swing* is a library of classes that provide an improved alternative for creating GUI applications and applets.
- Very few Swing classes rely on peer classes, so they are referred to called *lightweight components*.
- Swing draws most of its own components.
- Swing components have a consistent look and predictable behavior on any operating system.
- Swing components can be easily extended.
Event Driven Programming

- Programs that operate in a GUI environment must be event-driven.
- An event is an action that takes place within a program, such as the clicking of a button.
- Part of writing a GUI application is creating event listeners.
- An event listener is an object that automatically executes one of its methods when a specific event occurs.
javax.swing and java.awt

- In an application that uses Swing classes, it is necessary to use the following statement:
  
  ```java
  import javax.swing.*;
  ```
  
  - Note the letter x that appears after the word java.

- Some of the AWT classes are used to determine when events, such as the clicking of a mouse, take place in applications.

- In an application that uses an AWT class, it is necessary to use the following statement.

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

  Note that there is no x after java in this package name.
Creating Windows

• Often, applications need one or more windows with various components.
• A window is a container, which is simply a component that holds other components.
• A container that can be displayed as a window is a frame.
• In a Swing application, you create a frame from the JFrame class.
Creating Windows

• A frame is a basic window that has:
  – a border around it,
  – a title bar, and
  – a set of buttons for:
    • minimizing,
    • maximizing, and
    • closing the window.

• These standard features are sometimes referred to as window *decorations*.
Creating Windows

- See example: ShowWindow.java
Creating Windows

- The following `import` statement is needed to use the swing components:
  
  ```java
  import javax.swing.*;
  ```

- In the `main` method, two constants are declared:
  
  ```java
  final int WINDOW_WIDTH = 350;
  final int WINDOW_HEIGHT = 250;
  ```

- We use these constants later in the program to set the size of the window.
- The window’s size is measured in pixels.
- A *pixel (picture element)* is one of the small dots that make up a screen display.
Creating Windows

• An instance of the JFrame class needs to be created:
  ```java
  JFrame window = new JFrame();
  ```

• This statement:
  – creates a JFrame object in memory and
  – assigns its address to the window variable.

• The string that is passed to the setTitle method will appear in the window’s title bar when it is displayed.
  ```java
  window.setTitle("A Simple Window");
  ```

• A JFrame is initially invisible.
Creating Windows

• To set the size of the window:
  
```
  window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
```

• To specify the action to take place when the user clicks on the close button.
  
```
  window.setDefaultCloseOperation(JFrame.EXIT_ON_Close);
```

• The `setDefaultCloseOperation` method takes an int argument which specifies the action.
  
  - `JFrame.EXIT_ON_CLOSE` - causes the window to be hidden from view, but the application does not end.
  - The default action is `JFrame.EXIT_ON_CLOSE`. 
Creating Windows

• The following code displays the window:
  
  ```java
  window.setVisible(true);
  ```

• The `setVisible` method takes a boolean argument.
  
  - `true` - display the window.
  - `false` - hide the window.
Extending JFrame

- We usually use inheritance to create a new class that extends the JFrame class.
- When a new class extends an existing class, it inherits many of the existing class’s members just as if they were part of the new class.
- These members act just as if they were written into the new class declaration.
- New fields and methods can be declared in the new class declaration.
- This allows specialized methods and fields to be added to your window.
- Examples: SimpleWindow.java, SimpleWindowDemo.java
Equipping GUI Classes with a `main` Method

- Java applications always start execution with a method named `main`.
- The previous example used two separate files:
  - `SimpleWindow.java` -- the class that defines the GUI window
  - `SimpleWindowDemo.java` -- contains the `main` method that creates an instance of the `SimpleWindow` class.
- Applications can also be written with the `main` method directly written into the GUI class.
- See example: [EmbeddedMain.java](#)
Adding Components

• Swing provides numerous components that can be added to a window.

• Three fundamental components are:

  JLabel : An area that can display text.

  JTextField : An area in which the user may type a single line of input from the keyboard.

  JButton : A button that can cause an action to occur when it is clicked.
Sketch of Kilometer Converter
Graphical User Interface

Window Title
Label
Text Field
Button
private JLabel message;  
private JTextField kilometers;  
private JButton calcButton;  
...  
message = new JLabel("Enter a distance in kilometers");  
kilometers = new JTextField(10);  
calcButton = new JButton("Calculate");

• This code declares and instantiates three Swing components.
Adding Components

- A content pane is a container that is part of every JFrame object.
- Every component added to a JFrame must be added to its content pane. You do this with the JFrame class's add method.
- The content pane is not visible and it does not have a border.
- A panel is also a container that can hold GUI components.
Adding Components

- Panels cannot be displayed by themselves.
- Panels are commonly used to hold and organize collections of related components.
- Create panels with the JPanel class.

```java
private JPanel panel;
...
panel = new JPanel();
panel.add(message);
panel.add(kilometers);
panel.add(calcButton);
```
Adding Components

• Components are typically placed on a panel and then the panel is added to the JFrame's content pane.

    add(panel);

• Examples: KiloConverter.java
Handling Action Events

- An *event* is an action that takes place within a program, such as the clicking of a button.
- When an event takes place, the component that is responsible for the event creates an *event object* in memory.
- The event object contains information about the event.
- The component that generated the event object is known as the *event source*.
- It is possible that the source component is connected to one or more event listeners.
Handling Action Events

- An *event listener* is an object that responds to events.
- The source component *fires* an event which is passed to a method in the event listener.
- Event listener classes are specific to each application.
- Event listener classes are commonly written as private inner classes in an application.
Writing Event Listener Classes as Private Inner Classes

A class that is defined inside of another class is known as an inner class

```java
public class Outer {
    {
        Fields and methods of the Outer class appear here.
    }

    private class Inner {
        {
            Fields and methods of the Inner class appear here.
        }
    }
}
```
Event Listeners Must Implement an Interface

- All event listener classes must implement an interface.
- An interface is something like a class containing one or more method headers.
- When you write a class that implements an interface, you are agreeing that the class will have all of the methods that are specified in the interface.
Handling Action Events

- **JButton** components generate *action events*, which require an *action listener* class.

- Action listener classes must meet the following requirements:
  - It must implement the `ActionListener` interface.
  - It must have a method named `actionPerformed`.

- The `actionPerformed` method takes an argument of the `ActionEvent` type.

```java
public void actionPerformed(ActionEvent e) {
    // Code to be executed when button is pressed goes here.
}
```
Handling Action Events

When the button is pressed …

The JButton component generates an event object and passes it to the action listener object's actionPerformed method.

Example:
KiloConverter.java
Registering A Listener

- The process of connecting an event listener object to a component is called registering the event listener.
- JButton components have a method named addActionListener.

```java
calcButton.addActionListener(new CalcButtonListener());
```

- When the user clicks on the source button, the action listener object’s `actionPerformed` method will be executed.
Background and Foreground Colors

- Many of the Swing component classes have methods named `setBackground` and `setForeground`.
- `setBackground` is used to change the color of the component itself.
- `setForeground` is used to change the color of the text displayed on the component.
- Each method takes a color constant as an argument.
Color Constants

• There are predefined constants that you can use for colors.

```
Color.BLACK  Color.BLUE
Color.CYAN    Color.DARK_GRAY
Color.GRAY    Color.GREEN
Color.LIGHT_GRAY Color.MAGENTA
Color.ORANGE  Color.PINK
Color.RED     Color.WHITE
Color.YELLOW
```

• Examples: [ColorWindow.java](#)
The **ActionEvent Object**

- Event objects contain certain information about the event.
- This information can be obtained by calling one of the event object’s methods.
- Two of these methods are:
  - `getSource` - returns a reference to the object that generated this event.
  - `getActionCommand` - returns the action command for this event as a `String`.
- Example:
  - [EventObject.java](#)
Layout Managers

• An important part of designing a GUI application is determining the layout of the components.

• The term layout refers to the positioning and sizing of components.

• In Java, you do not normally specify the exact location of a component within a window.

• A layout manager is an object that:
  – controls the positions and sizes of components, and
  – makes adjustments when necessary.
Layout Managers

• The layout manager object and the container work together.

• Java provides several layout managers:
  – FlowLayout - Arranges components in rows. This is the default for panels.
  – BorderLayout - Arranges components in five regions:
    • North, South, East, West, and Center.
    • This is the default layout manager for a JFrame object’s content pane.
  – GridLayout - Arranges components in a grid with rows and columns.
Layout Managers

- The **Container** class is one of the base classes that many components are derived from.
- Any component that is derived from the **Container** class can have a layout manager added to it.
- You add a layout manager to a container by calling the `setLayout` method.

```java
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

- In a **JFrame** constructor you might use:
  ```java
  setDefaultCloseOperation(new FlowLayout());
  ```
FlowLayout Manager

- `FlowLayout` is the default layout manager for `JPanel` objects.
- Components appear horizontally, from left to right, in the order that they were added. When there is no more room in a row, the next components “flow” to the next row.
- See example: [FlowWindow.java](#)
FlowLayout Manager

- The `FlowLayout` manager allows you to align components:
  - in the center of each row
  - along the left or right edges of each row.
- An overloaded constructor allows you to pass:
  - `FlowLayout.CENTER`,
  - `FlowLayout.LEFT`, or
  - `FlowLayout.RIGHT`.
- Example:
  ```java
  setLayout(new FlowLayout(FlowLayout.LEFT));
  ```
FlowLayout Manager

- FlowLayout inserts a gap of five pixels between components, horizontally and vertically.
- An overloaded FlowLayout constructor allows these to be adjusted.
- The constructor has the following format:

  ```java
  FlowLayout(int alignment,
             int horizontalGap,
             int verticalGap)
  ```

- Example:
  ```java
  setLayout(new FlowLayout(FlowLayout.LEFT, 10, 7));
  ```
BorderLayout Manager

BorderLayout manages five regions where components can be placed.
BorderLayout Manager

- See example: `BorderWindow.java`
- A component placed into a container that is managed by a `BorderLayout` must be placed into one of five regions:
  - `BorderLayout.NORTH`
  - `BorderLayout.SOUTH`
  - `BorderLayout.EAST`
  - `BorderLayout.WEST`
  - `BorderLayout.CENTER`
BorderLayout Manager

- Each region can hold only one component at a time.
- When a component is added to a region, it is stretched so it fills up the entire region.
- BorderLayout is the default manager for JFrame objects.

```java
add(button, BorderLayout.NORTH);
```

- If you do not pass a second argument to the add method, the component will be added to the center region.
BorderLayout Manager

- Normally the size of a button is just large enough to accommodate the text that it displays.
- The buttons displayed in BorderLayout region will not retain their normal size.
- The components are stretched to fill all of the space in their regions.
BorderLayout Manager

- If the user resizes the window, the sizes of the components will be changed as well.
- BorderLayout manager resizes components:
  - placed in the north or south regions may be resized horizontally so it fills up the entire region,
  - placed in the east or west regions may be resized vertically so it fills up the entire region.
  - A component that is placed in the center region may be resized both horizontally and vertically so it fills up the entire region.
BorderLayout Manager

- By default there is no gap between the regions.
- An overloaded BorderLayout constructor allows horizontal and vertical gaps to be specified (in pixels).
- The constructor has the following format

  ```java
  BorderLayout(int horizontalGap, int verticalGap)
  ```

- Example:
  ```java
  setLayout(new BorderLayout(5, 10));
  ```
Nesting Components in a Layout

- Adding components to panels and then nesting the panels inside the regions can overcome the single component limitation of layout regions.
- By adding buttons to a JPanel and then adding the JPanel object to a region, sophisticated layouts can be achieved.
- See example: [BorderPanelWindow.java](#)
GridLayout Manager

GridLayout creates a grid with rows and columns, much like a spreadsheet. A container that is managed by a GridLayout object is divided into equally sized cells.
GridLayout Manager

- GridLayout manager follows some simple rules:
  - Each cell can hold only one component.
  - All of the cells are the size of the largest component placed within the layout.
  - A component that is placed in a cell is automatically resized to fill up any extra space.
- You pass the number of rows and columns as arguments to the GridLayout constructor.
GridLayout Manager

- The general format of the constructor:
  `GridLayout(int rows, int columns)`

- Example
  `setLayout(new GridLayout(2, 3));`

- A zero (0) can be passed for one of the arguments but not both.
  - passing 0 for both arguments will cause an `IllegalArgumentException` to be thrown.
GridLayout **Manager**

- Components are added to a `GridLayout` in the following order (for a 5×5 grid):

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

Example: [GridWindow.java](#)

GridLayout also accepts nested components:

Example: [GridPanelWindow.java](#)
Radio Buttons

- *Radio buttons* allow the user to select one choice from several possible options.
- The `JRadioButton` class is used to create radio buttons.
- `JRadioButton` constructors:
  - `JRadioButton(String text)`
  - `JRadioButton(String text, boolean selected)`
- Example:
  ```java
  JRadioButton radiol = new JRadioButton("Choice 1");
  or
  JRadioButton radiol = new JRadioButton("Choice 1", true);
  ```
Button Groups

- Radio buttons normally are grouped together.
- In a radio button group only one of the radio buttons in the group may be selected at any time.
- Clicking on a radio button selects it and automatically deselects any other radio button in the same group.
- An instance of the `ButtonGroup` class is used to group radio buttons.
Button Groups

- The `ButtonGroup` object creates the *mutually exclusive* relationship between the radio buttons that it contains.

```java
JRadioButton radio1 = new JRadioButton("Choice 1", true);
JRadioButton radio2 = new JRadioButton("Choice 2");
JRadioButton radio3 = new JRadioButton("Choice 3");
ButtonGroup group = new ButtonGroup();
group.add(radio1);
group.add(radio2);
group.add(radio3);
```
Button Groups

- `ButtonGroup` objects are not containers like `JPanel` objects, or content frames.
- If you wish to add the radio buttons to a panel or a content frame, you must add them individually.

```java
panel.add(radio1);
panel.add(radio2);
panel.add(radio3);
```
Radio Button Events

- `JRadioButton` objects generate an action event when they are clicked.
- To respond to an action event, you must write an action listener class, just like a `JButton` event handler.
- See example: [MetricConverter.java](#)
Determining Selected Radio Buttons

- The `JRadioButton` class’s `isSelected` method returns a `boolean` value indicating if the radio button is selected.

```java
if (radio.isSelected()) {
    // Code here executes if the radio button is selected.
}
```
Selecting a Radio Button in Code

- It is also possible to select a radio button in code with the `JRadioButton` class’s `doClick` method.
- When the method is called, the radio button is selected just as if the user had clicked on it.
- As a result, an action event is generated.

```java
radio.doClick();
```
Check Boxes

- A *check box* appears as a small box with a label appearing next to it.
- Like radio buttons, check boxes may be selected or deselected at run time.
- When a check box is selected, a small check mark appears inside the box.
- Check boxes are often displayed in groups but they are not usually grouped in a `ButtonGroup`. 
Check Boxes

- The user is allowed to select any or all of the check boxes that are displayed in a group.
- The `JCheckBox` class is used to create check boxes.
- Two `JCheckBox` constructors:
  
  ```java
  JCheckBox(String text)
  JCheckBox(String text, boolean selected)
  ```

- Example:
  ```java
  JCheckBox check1 = new JCheckBox("Macaroni");
  or
  JCheckBox check1 = new JCheckBox("Macaroni", true);
  ```
Check Box Events

• When a JCheckBox object is selected or deselected, it generates an item event.
• Handling item events is similar to handling action events.
• Write an item listener class, which must meet the following requirements:
  – It must implement the ItemListener interface.
  – It must have a method named itemStateChanged.
    • This method must take an argument of the ItemEvent type.
Check Box Events

• Create an object of the class
• Register the item listener object with the JCheckBox component.
• On an event, the itemStateChanged method of the item listener object is automatically run
  – The event object is passed in as an argument.
Determining Selected Check Boxes

• The `isSelected` method will determine whether a `JCheckBox` component is selected.

• The method returns a `boolean` value.

```java
if (checkBox.isSelected()) {
    // Code here executes if the check box is selected.
}
```

• See example: [ColorCheckBoxWindow.java](#)
Selecting Check Boxes in Code

- It is possible to select check boxes in code with the `JCheckBox` class’s `doClick` method.
- When the method is called, the check box is selected just as if the user had clicked on it.
- As a result, an item event is generated.

```java
checkBox.doClick();
```
Borders

• Windows have a more organized look if related components are grouped inside borders.

• You can add a border to any component that is derived from the JComponent class.
  – Any component derived from JComponent inherits a method named setBorder
Borders

- The `setBorder` method is used to add a border to the component.
- The `setBorder` method accepts a `Border` object as its argument.
- A `Border` object contains detailed information describing the appearance of a border.
- The `BorderFactory` class, which is part of the `javax.swing` package, has static methods that return various types of borders.
<table>
<thead>
<tr>
<th>Border</th>
<th>BorderFactory Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compound border</td>
<td>createCompoundBorder</td>
<td>A border that has two parts: an inside edge and an outside edge. The inside and outside edges can be any of the other borders.</td>
</tr>
<tr>
<td>Empty border</td>
<td>createEmptyBorder</td>
<td>A border that contains only empty space.</td>
</tr>
<tr>
<td>Etched border</td>
<td>createEtchedBorder</td>
<td>A border with a 3D appearance that looks “etched” into the background.</td>
</tr>
<tr>
<td>Line border</td>
<td>createLineBorder</td>
<td>A border that appears as a line.</td>
</tr>
<tr>
<td>Lowered bevel border</td>
<td>createLoweredBevelBorder</td>
<td>A border that looks like beveled edges. It has a 3D appearance that gives the illusion of being sunken into the surrounding background.</td>
</tr>
<tr>
<td>Matte border</td>
<td>createMatteBorder</td>
<td>A line border that can have edges of different thicknesses.</td>
</tr>
<tr>
<td>Raised bevel border</td>
<td>createRaisedBevelBorder</td>
<td>A border that looks like beveled edges. It has a 3D appearance that gives the illusion of being raised above the surrounding background.</td>
</tr>
<tr>
<td>Titled border</td>
<td>createTitledBorder</td>
<td>An etched border with a title.</td>
</tr>
</tbody>
</table>
The Brandi’s Bagel House Application

- A complex application that uses numerous components can be constructed from several specialized panel components, each containing other components and related code such as event listeners.

- Examples:
  - GreetingPanel.java, BagelPanel.java, ToppingPanel.java, CoffeePanel.java, OrderCalculatorGUI.java
Splash Screens

- A splash screen is a graphic image that is displayed while an application loads into memory and starts up.
- A splash screen keeps the user's attention while a large application loads and executes.
- Beginning with Java 6, you can display splash screens with your Java applications.
Splash Screens

- To display the splash screen you use the `java` command in the following way when you run the application:

  ```java -splash:GraphicFileName ClassFileName```

- `GraphicFileName` is the name of the file that contains the graphic image, and `ClassFileName` is the name of the `.class` file that you are running.

- The graphic file can be in the GIF, PNG, or JPEG formats.
Using Console Output to Debug a GUI

- Display variable values, etc. as your application executes to identify logic errors
  - Use `System.out.println()`

```
// For debugging, display the text entered, and its value converted to a double.
System.out.println("Reading " + str + " from the text field.");
System.out.println("Converted value: " + Double.parseDouble(str));
```

- See example: `KiloConverter.java`