Companion Chapter 1

Creating Java User Interfaces with Swing

What You Will Learn in This Chapter

Users interact with contemporary information systems, as with most modern software, through a graphical user interface (GUI). Console-based applications, though still in use in many contexts, are less common. Information systems professionals who are well rounded in their design and development skills should have some experience in the planning, implementation, and refinement of a user interface (UI) for their applications beyond interactions in the console. Java's Swing framework provides an easy-to-learn, object-oriented class library that can help Java developers quickly build professional-looking, business-ready Java GUI applications that incorporate several good "front-end" design practices. The Swing framework is still popular and widely used for Java GUI application development and is well worth the information systems professional's time and effort to learn.

Specifically, this chapter will help you do the following:

1. Understand the similarities and differences between the AWT, Swing, and JavaFX graphical libraries
2. Understand the nature of the Swing framework and its approach to designing a GUI-based Java application
3. Understand how to use layout managers and Swing components, how to specify a visual relationship between these, and how to display them as a GUI-based application
4. Understand how to handle user-generated events in a Swing GUI application, the event delegation model, and the three techniques for event handling
5. Understand the approach to multiform GUI development in Swing in a compact and secondary window approach
6. Understand the manner in which many Swing components store data and how to interact with that data

Opening Scenario

After extensive work, you and your team have developed an object-oriented, class-based data model for the mom-and-pop grocery inventory system. Using Java's exception-handling classes and studying best practices in Java development efforts, your team has prepared error-checking and error-mitigation logic to be integrated into the client's system. After a thorough review, your project manager believes both you and the team are ready for the final stage in the prototype development, prior to a "so-far" demo to the client: "We need to bring this application into the twenty-first century." The next step is to develop a GUI front end for the application, as the console-based menu system is a little awkward. Your project manager says, "Since they use GUI-based applications on their personal computers, the owners are not going to want to see a console-based, 'type-a-number-1-here-type-a-number-2-there' system. For this client, it just won't be a good fit. People want to point and click."

Sitting down at your development machine, you realize that you have never thought a lot about how the software that you use every day looks and feels. Immediately all the occasions where applications have frustrated you come to mind—like those times you have spent many minutes trying to find this or that menu option or this and that feature of the application and so on. You recall how some software has been easy to learn and how some applications needed a very thick manual that you now set your coffee cup on. "We have to be mindful of ease of use as well," you think, knowing that time spent carefully designing the look and feel of the client's system will be just as important as time spent writing its code and implementing the client's business rules. You've heard about a few Java GUI libraries but have never needed to take a closer look until now. One more time, refill your coffee mug and dive into the research. You think, "If we can get this right, the clients will be very happy with our abilities and will want to continue this project far into the future . . ."

CC1.1 Overview of GUI Frameworks in Java

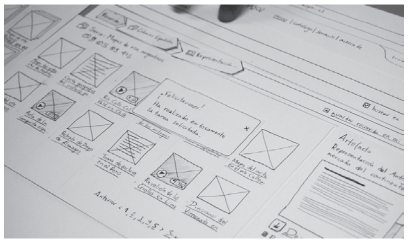


Figure CC1.1. Wireframe Diagrams Are Often Used during the User Interface Design Process
Source: "Wireframes" by Christian_Campos is licensed under CC BY 2.0, https://www.flickr.com/photos/23438340@N02/3883508604.

The development and refinement of the graphical user interface (GUI) truly helped bring computing to the masses. Forms, folders, icons, images, "physical" controls—all these real-world things were digitally analogized in the form of the GUI interface, eliminating the need for people to learn terse and often confusing command-line syntax. This opened personal computing up to the broad population and laid the groundwork for the web, mobile, artificial intelligence (AI), and "metaverse" platforms so popular today. Companies like Microsoft in the 1980s and later Apple, among others, pioneered early work in adding a GUI layer over top of their operating systems for personal computing systems. This is not to say that earlier computing platforms (both consumer and business focused) did not have a visual user interface. Companies like Xerox and Amiga had earlier (than Apple and Microsoft) introduced point-and-click quasi-graphical systems, but their use was limited and often found only in experimental and lab usages.* Personal computer operating systems interfaces ranged from simple console-like command interfaces (like MS-DOS) to text-/menu-based systems that allowed free movement between commands. The move to GUI systems was often rapid and (again, in the case of Microsoft and Apple) contentious, but innovation along with the more natural and effective use of computing that comes with a GUI drove this computing interface approach to dominance.

Today, computer users leverage the user interface that works best for them and the tasks they want to accomplish. In business, you will find a mix of both console-based applications and GUI-driven software. Often, console applications are running in a GUI-based operating system (i.e., Windows PowerShell for issuing advanced OS commands within Windows). But the GUI design philosophy reigns supreme, both in desktop applications and in web, mobile, and tablet software implementations. In information systems (IS) scenarios, software interactions occur through form-based data entry, tabular reporting, tabbed organization of information, visual display of analyzed data through charting and other visualization types, and so on. As information systems professionals, you will encounter these and other GUI scenarios in your daily planning and implementation work.

Directly related to the consideration of a GUI implemented for a system are questions such as the following: Is this data entry method effective? Should this screen be redesigned? How can we reduce the amount of time it takes to accomplish a task? Is the UI intuitive, easy, and efficient? Researchers in IS tell us that two primary reasons (among many) why users choose to either adopt and use or reject an IS are due to their perception of self-benefit ("Will using this system help me in my job?") and ease of use ("Is this system easy to use, easy to learn, and better than the 'old' way?"). For the IS professional engaged in systems analysis and design, potentially working on the future "to-be" system, these are big hurdles to overcome. Part of your planning and design work should be laser-focused on these two questions when considering a redesign of an existing system or when planning for a new one.

Many contemporary, popular programming languages include a framework for developing a GUI application. This includes Java, which has two built into the language and a third that was recently moved out of the official JDK by Oracle to an open-source community. Prior chapters have prepared you with a solid basis

* Eric Raymond tours much of this earlier work at http://www.catb.org/~esr/writings/taouu/html/ch02s05.html.

for understanding Java's object-oriented (OO) nature. This chapter covers the usage of Swing, a popular object-oriented Java GUI framework. First, it will be helpful to discuss the GUI frameworks available in Java. Chapter 11 will walk you through the JavaFX framework if you are interested (or if your instructor covers that framework instead).

Abstract Window Toolkit (AWT): This was the earliest official framework for creating a graphical, windowed user interface for a Java program. AWT was innovative and made GUI development easily available to Java developers. Originally developed by Sun Microsystems (the originators of the Java language), it was popular in its time, but it had several drawbacks:

- **Lack of portability:** AWT GUI elements (buttons, scroll bars, check boxes, etc.) were not part of AWT themselves. Rather, the framework required the use of whatever native controls were available through the operating system. If you wanted to move your AWT Java application to another OS, some rewriting of code was required to use that OS's controls. This made regular Java programs much more portable than AWT-based Java programs. This tie to the underlying OS limited the controls that could be used. If you wanted a new control, you had to code it from scratch.
- **Tightly integrated logic and UI:** There was no "separation of concerns," a popular contemporary application design paradigm where the logic of an application and its user interface are developed separately. With AWT, you were forced to write logic and UI together with no alternative, decreasing modularity and hindering ease of maintenance by multiple developers.
- **Resource inefficiency:** The objects created by using the AWT framework were bulky (back when computing resources were more limited), and the behavior of AWT programs could be buggy/inconsistent.

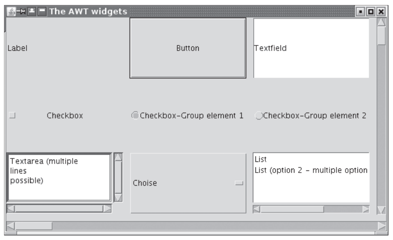


Figure CC1.2. Example of an AWT Java Application in the Linux OS
Source: "AWT_at_Linux.jpg" by Sven is public domain, https://commons.wikimedia.org/wiki/File:AWT_at_Linux.png.

Swing: Developed by Sun Microsystems through a partnership with Netscape in the 1990s, the Swing framework resolved many of the AWT drawbacks and became a popular successor. Swing was built upon the AWT framework and shares some compatibility with it. Swing is still in broad use today. Many of its benefits include the following:

- Platform independence: Swing controls are part of the framework itself, so the developer is not forced to use those built into the OS. Swing applications use AWT to create a window, and then Swing handles the creating/display of controls and interactions with the OS generated by user clicks and activity. This allows for easy portability of a Swing Java application from one OS to another, as Swing interprets the activity generated and decides how to send that to the OS, instead of having the developer make those OS calls directly. Think of it like a mini–virtual machine (VM) for GUI-based Java applications!
- Since Swing controls were part of the framework itself, this gave developers a much higher variety of controls to work with other than just those natively included on the OS within which the application was running (see figure CC1.3 for an example of this variety).
- Since Swing handles its own controls, it has a much richer set of them than any one OS. Due to Swing's portability, the user experience is consistent across operating systems: a button in a Swing app running on Windows will look and act (mostly) the same as one on Mac OS X, Linux, and so on.

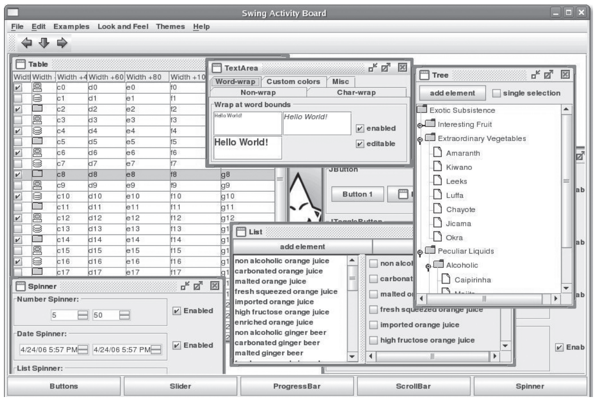


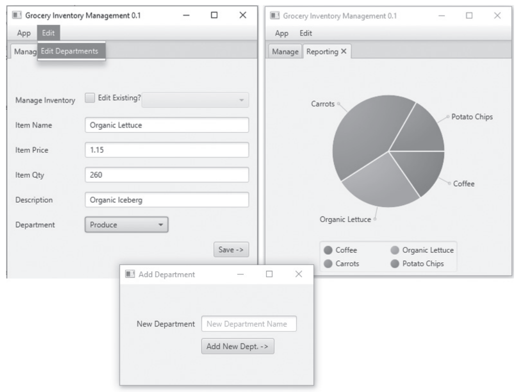
Figure CC1.3. Example of a Java Swing UI Application
Source: "The Swing (Graphics User Interface) Demo of the GNU Classpath Project" by Audrius Meskauskas is licensed as free software under the GNU General Public License, https://commons.wikimedia.org/wiki/File:GC_SwingDemo.PNG.

- Swing allows for a "model-view-controller" approach to GUI app development, which adheres to the "separation of concerns" design paradigm. The user interface code can be separate from the logic that executes when the UI is used. This allows easier maintenance: one or more developers can work on the UI separately and in parallel with other developers who might be working on the application's logic.
- Swing is still relatively popular today. Developers like its reliability (far less buggy!), consistency, and ease with which development can occur.

JavaFX: Developed originally by Sun Microsystems and expanded upon by Oracle, JavaFX was intended as the natural successor to Swing (and still is to some extent). In 2018, Oracle moved JavaFX from "in-house" development and out under the umbrella of OpenJDK (open-source version of Java), into the OpenJFX project. JavaFX is no longer included as a native part of the newest Java JDK releases, though it is still included in JDK 8.[*,†] This adds some extra steps needed to run a JavaFX project in your IDE of choice (see appendix A for JavaFX installation instructions), but this allows the framework to thrive under speedy community development. Some of the advantages that the JavaFX framework adds above and beyond Swing are the following:

- **Web-ready GUI framework:** JavaFX was intended to be compatible with the web. Web content can be embedded within a JavaFX program, and a JavaFX Program can be deployed as a website itself.
- **JavaFX supports multitouch scenarios:** Multitouch, along with a web-compatible nature, makes JavaFX a great development choice for desktop, web, mobile, or tablet-based use cases. Like Swing, JavaFX applications are portable across operating systems, enabling a consistent user experience across both platforms and devices.
- **2D and 3D graphics and animations, rich text, advanced charting, and multimedia support:** JavaFX allows for a much richer user experience than Swing. Visual effects such as rotation, opacity, blurring, shadows, and more can be defined for almost any control on a JavaFX form. Paired with animations, developers can create fairly rich UI "front ends" for their information systems. JavaFX has visually appealing charting ability built in, giving developers quick access to business-ready visualizations.
- **High-degree of customizability and continued MVC support:** Almost every aspect of a JavaFX program can be altered aesthetically through Cascading Style Sheet (CSS) styling. Similar to Swing, JavaFX continues support for the MVC design paradigm, allowing the UI and the logic to be built separately.

Swing or JavaFX? Oracle and the development community have for some time intended for JavaFX to be the natural successor to Swing, though there has not been an aggressive push for this to happen. One positive aspect of the broader Java development community is the flexibility of the language provided by its variety of frameworks and technologies. For the Java developer interested in GUI application development, there is room for both Swing and JavaFX in your "mental toolbox." Swing is still widely used in the Java development community and shares a lot of syntax similarities with JavaFX's framework. The Swing framework will be overviewed in this chapter. The basics, nature, and customizability of Swing (as well as JavaFX) are quickly learned, enhancing your skills as a developer working in the Java language. By learning how to work with a popular Java GUI framework, you become a more well-rounded developer as well, gaining experience in both the "front end" of development (GUI, user experience, etc.) and the "back end," consisting of the logic that represents the business rules and required application functionality. Additionally, knowledge of a GUI framework like Swing will help the developer who finds themselves involved in a project where other GUI frameworks like JavaFX, for example, might be used. The similarities between the two will help you learn to transition from one to the other.



Manage Inventory ☐ Edit Existing?
Item Name: Organic Lettuce
Item Price: 1.15
Item Qty: 260
Description: Organic Iceberg
Department: Produce

● Coffee ● Organic Lettuce
● Carrots ● Potato Chips

New Department: New Department Name

Figure CC1.4. Example of a JavaFX Form with Charting and Other Controls

[*] https://wiki.openjdk.java.net/display/OpenJFX/Main.

[†] https://openjfx.io/.

SUMMARY POINTS

- The graphical user interface is the dominant method by which users interact with software and computing-based systems.
- Information systems professionals consider several factors when designing software and the interface through which users will interact with it, focusing on issues like self-efficacy and ease of use.
- The Abstract Window Toolkit (AWT) was the earliest GUI framework used in the Java language for creating a visual user interface for Java applications. The framework was heavily dependent on the UI resources of the operating system the Java application was targeted for.
- The Swing framework, next offered as a GUI library for Java, improved upon AWT's weaknesses—namely, through operating system independence and internally implemented UI controls.
- The JavaFX framework, now outsourced to the OpenJFX group with Oracle consultation, modernizes both the GUI controls available for Java applications and their functionality in web/internet contexts.

QUICK PROBLEMS

1. **Think:** Think about the various computing-based devices you interact with on a daily basis. What are some of the things that are common across all of them in terms of the GUI / user interface? Can all tasks (i.e., spreadsheet editing) work effectively on device screens of all sizes (i.e., your smartphone)?
2. **Think:** What is a benefit to having a GUI framework in Java be "operating system independent," and how does this mirror the benefits of the Java language itself?
3. **Think:** Can you perform wireframing and user interface planning for non-GUI applications, like a console/terminal application?

CC1.2 Building a Basic Swing GUI Java Application

Oracle describes[*] the Swing framework as an API (application programming interface) that is itself part of the Java Foundation Classes. Like the base Java language itself, the Swing is intended to function the same on every technology and to have the same look and feel.[†] Swing is an object-oriented GUI framework, meaning that the OO syntax you have learned across many chapters in this textbook will pay off here. The Swing framework may be new to you, but you are diving in with a knowledge of how classes, constructors, instance objects, data fields, getters, setters, and other OO concepts work. This knowledge will help you understand the sequence and order of Swing syntax quickly, since all of these syntax concepts are used in this framework. Your OO knowledge will keep you "grounded." There are many classes that make up the Swing framework, and each class has many aspects. The framework itself is huge and could not possibly be covered all in one chapter.

Understanding the Swing framework: The Swing framework was created with the idea of allowing developers to create a Java GUI-based application that would look and feel the same when executed across any platform that runs Java. Swing is considered the successor to the Abstract Window Toolkit (AWT). Swing is considered to be more "lightweight" than AWT and more resource efficient with a more convenient OO syntax and a more universal look and feel across operating systems/platforms to the furthest extent possible. Far less work is needed by the developer to ensure uniformity across technology platforms with Swing thanks to built-in UI controls.

As discussed in chapter 9, the Java API itself is hierarchical in nature ("parent" and "child" classes, inheritance of public members of parent classes, etc.). Swing's class library is similarly hierarchical, which allows for efficient extension of classes into subclasses with specific functionality. Figure CC1.5 displays part (but

[*] https://docs.oracle.com/javase/tutorial/uiswing/start/about.html.

[†] https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html.

not all!) of the Swing framework's hierarchically related classes as discussed in the official Oracle documentation.*

Since Swing is a GUI framework, it manages the display of the visual equivalents of these classes (the controls you can click on and interact with in the application) in a hierarchical manner as well. Swing uses a **containment hierarchy** to manage the different display layers for any GUI window. A GUI window in Swing could be a traditional frame, a dialogue window, or (less common today) an applet for display on a website. Each of these would have its own containment hierarchy where GUI **components** are stored, arranged, and displayed. Each containment hierarchy starts with a **top-level container**, which is the "highest" in the hierarchy, and everything else will "nest" inside of it. For example, if your Swing application has a single traditional GUI window and two dialogue windows (we will explore these later), then there would be three separate containment hierarchies in your application (*think:* three "family trees" of GUI objects that are displayed).

To understand the basic parts of a Swing application, consider the following example. In a Swing GUI form are placed two `JLabel` components, two `JTextField` components, a `JButton`, and a `JTextArea` to display output. Notice the class names are fairly self-descriptive. Assume this application's purpose is to have the user type in a first and second name, and the application will concatenate the two names together and display the full name in the `JTextArea`. Figure CC1.6 shows a rough wireframe sketch of this simple application and its possible look and feel.

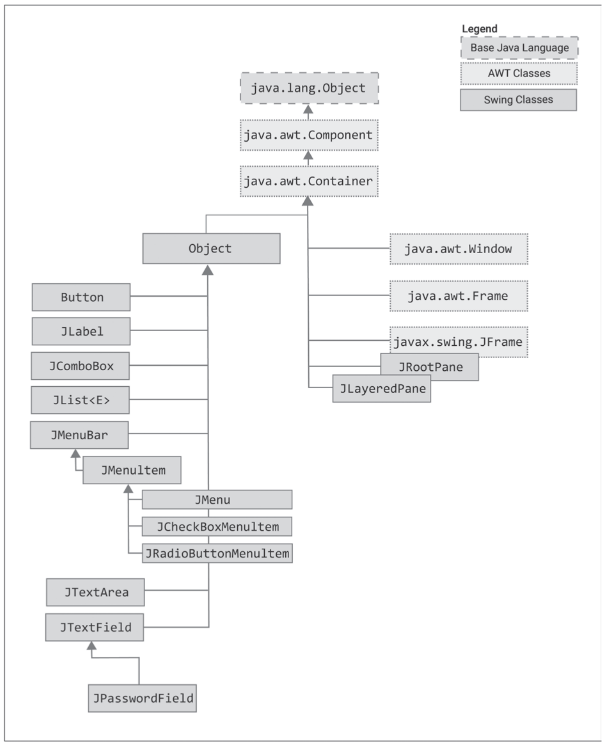


Figure CC1.5. Class Inheritance and Data Field Relationships for Selected Swing Classes

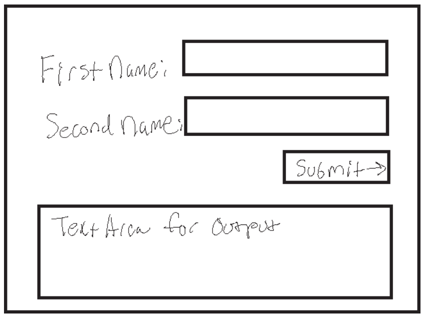


Figure CC1.6. Wireframe Diagram of Name Display Application

The containment hierarchy of this Swing application would look something like figure CC1.7. Notice the multiple layers, some of which are displayed visually and some of which are not.

Briefly, each of these layers of our Swing application works as follows:

- `JFrame`: In a Swing application, each window or display contains a **top-level container** that serves as the "root" or beginning of the containment hierarchy. In our simple application, our traditional main window will have as its top-level container a `JFrame` object. In a dialogue window, a `JDialog` object would serve as the top-level container. Every other layer is nested within this root layer, thus creating the hierarchy. The `JFrame` layer is not visually displayed.†
- `JRootPane`: A `JFrame` instance object will contain a `JRootPane` as a data field. The `JFrame` uses the `JRootPane` to contain all the pane layers beneath it. The `JRootPane` is not visually displayed, and developers rarely interact with it.‡
- `JLayeredPane`: A data field of the `JRootPane` class. Developers can use the `JLayeredPane` to provide a depth arrangement to components (*think:* third dimension, components overlapping one another, etc.) in

* https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html.

† https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html.

‡ https://docs.oracle.com/javase/7/docs/api/javax/swing/JRootPane.html.

a convenient way. Developers rarely need to interact with the `JLayeredPane` object. The `JLayeredPane` object will manage the two major objects that are displayed: the menu bar (if the developer adds one) and the content pane.*

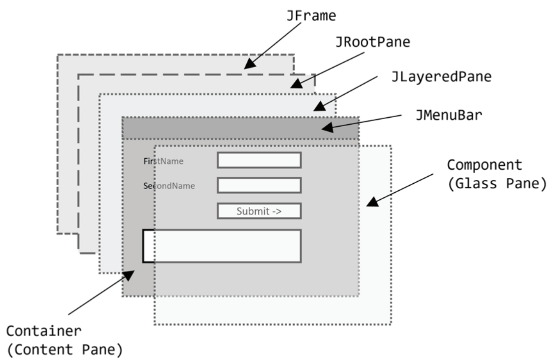


Figure CC1.7. Containment Hierarchy for a Simple `JFrame` Swing Application

- `JMenuBar` and `Container`: The two major layers that are visually displayed will be the `Container` data field of the `JLayeredPane` object called `contentPane` and the optional `JMenuBar` data field of `JLayeredPane` called `menuBar`. The `contentPane` will visually take up the full area of the displayed GUI window unless the developer adds a `JMenuBar` object to the display, which will then display along the top edge of the GUI window automatically, resizing the `contentPane` to make room. The Oracle documentation for the `JRootPane` class has a great discussion on this, and we will see this demonstrated later in this chapter.
- The `glassPane Component` data field: Finally, the `JRootPane` object has a `Component` data field called `glassPane`. Developers will rarely work with this layer. The `glassPane` can allow the developer to draw over the GUI and over individual components, and it can intercept actions/events and prevent them from reaching components in the application. By default, the `glassPane` is not visible.†

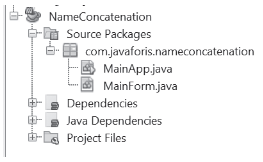


Figure CC1.8. Beginning Contents of the `NameConcatenation` Project in Apache NetBeans

Understanding the basic Swing application: At this point, you can begin working with the classes in the Swing API to build your first Swing GUI application. Since Swing is still (as of the time of this writing) a native part of the Java JDK, there is **nothing to install or configure** in order for you to begin using it. In your IDE of choice (we will develop in Apache NetBeans in this chapter), create a new Java project, and name it `NameConcatenation`. Once your project is created, create a new Java `main()` class file in the default package, and give it the name `MainApp.java` to indicate it is the starting point for the application and will (eventually) create and display the Swing form. Next, add to the default package a Java class (nonmain class), and call it MainForm.java. Once done, your project's file structure should look similar to that in figure CC1.8. Whether you are using NetBeans, Eclipse, or any other software, the two `.java` files should appear in the same package folder.

The design philosophy here is simple and uses OO techniques as cleanly as possible: `MainApp` will create an instance object of the `MainForm` class, which itself uses the Java Swing classes to display a form with controls and perform our name concatenation and display task. You certainly could place the code that we will eventually write in `MainForm` instead into `MainApp` (and many tutorials on the web do this very thing), but our approach, with a clean separation, will keep things simple. The app launches the form, and the form allows the user to carry out their task.

Focus first on `MainForm.java`. Add the following imports, which will define the Swing classes needed:

Code Snippet CC1.1

```
import java.awt.*;              // Import All
import java.awt.event.*;        // Import All
import javax.swing.*;           // Import All

import java.util.*;             // Import All
```

Since Swing expands and builds upon the AWT, some of the imports must come from that package and the rest from the Swing package.

* https://docs.oracle.com/javase/7/docs/api/javax/swing/JLayeredPane.html.

† https://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html#glasspane.

Working with Swing components: The components in a Swing application are displayed visually in the GUI, and these are what the user will interact with. For IS professionals, the development effort starts before the first line of code is written in the determination and clarification of requirements for the application. The look and feel, effectiveness, efficiency, and ease of use of a system as a whole must be hammered out prior to writing the first line of code. Part of this planning involves talking to stakeholders, clients, potential users, and others who might use or be impacted by the software that is planned to determine how it will be used and what it should do. Our wireframe diagram of the application in figure CC1.6 earlier is a key part of this planning process. Both the developer (and their team if a member of one) and the client will and should have input on wireframing of various parts of the GUI during the planning process.

For this simple application, there are two main functionality requirements:

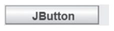
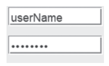
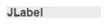
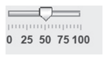
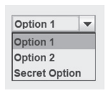
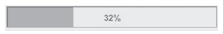
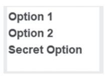
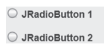
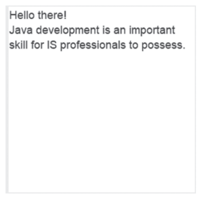
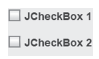
- The user can enter their first and last names.
- The user can click a button that will cause the two names to be concatenated and displayed in the output area in the application window.

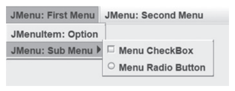
Pretty simple! Wireframing and initially elicited system requirements are crucial steps in the systems development process, and getting the client(s) and/or stakeholders involved in this process can help spur discussion and clarification of "fuzzy" requirements. Often this due diligence can yield more system features that may have been missed if the planning process had been neglected.

Table CC1.1 displays names and examples of many of the more commonly used Swing components. From the wireframe design in figure CC1.6, you have a good idea of what types of controls will be needed for the application. Table CC1.1 can provide guidance if an application's functionality needs expansion via additional controls. You will see examples of several of these table CC1.1 components later on in this chapter. Many characteristics of each component can be changed; for example, images can be added and displayed instead of text within a `JButton` or `JLabel`.

Manually sizing and positioning components: You will build the form in two ways. First, both the position and the size of the Swing components on the form will need to be specified. This will give you an idea of how to exert precise control over the positioning of your components if you need it. After this, the use of several `JPanel` components to group visual GUI elements and Swing layout managers to automatically configure the form's layout will be explored.

Table CC1.1. Some Common Java Swing Components and Example Displays of Each

Swing component class name	Visual example of component	Swing component class name	Visual example of component
`JButton`		`JTextField` and `JPasswordField`	
`JLabel`		`JSlider`	
`JComboBox`		`JProgressBar`	
`JList<E>` (can be generically typed to visually display many class instance object types)		`JRadioButton` (add these to a `ButtonGroup` to make them aware of each other—only one can be clicked)	
`JTextArea`	 Hello there! Java development is an important skill for IS professionals to possess.	`JCheckBox`	

		`JMenuBar`
		`JMenu`
		`JMenuItem`
			 JMenu: First Menu
			JMenu: Second Menu
			JMenuItem: Option
			JMenu: Sub Menu
			Menu CheckBox
			Menu Radio Button

From figure CC1.6, it appears the application needs two `JLabel` components, two `JTextField` components for the user to type into, a `JButton` component to (eventually) trigger the concatenation action, and a `JTextArea` to display output. Since the position and sizes of the components are being manually specified, the steps to display the GUI are relatively straightforward:

- Create and set the characteristics of components.
- Specify the position and size of the components.
- Create the `JFrame`, add the components to it, and set its characteristics.
- Instantiate our Java Swing form from the application.

Keep in mind that the application `MainApp.java` will instantiate the Java Swing form class `MainForm`, which will cause it to display. First, create the Swing components by adding them to `MainForm` as class-level data fields. The changes are in **bold** (see chapter 8 for more details on class-level data fields):

Code Snippet CC1.2

```
package com.javaforis.nameconcatenation;

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

import java.util.*;

// Class declaration
public class MainForm {

    // Class-level declarations of
    //  components
    JFrame mainFrame;
    JLabel lblFirstName;
    JLabel lblSecondName;
    JTextField txtFirstName;
    JTextField txtSecondName;
    JButton btnConcatName;
    JTextArea txtaOutput;


}
```

By adding Swing components at the class level, you will ensure that your code has access to them from anywhere in the `MainForm` class. If `MainApp` will instantiate `MainForm`, you will need a constructor that will handle the initialization of `MainForm` and cause it to display. The characteristics of the form's Swing components can be specified, the components positioned, and the `JFrame` set up from the constructor as well.

Add a zero-argument constructor for `MainForm`, and initialize the components in the following way (remember that constructors are placed inside the class declaration and usually after any class-level data fields):

Code Snippet CC1.3

```
// Zero-Arg Constructor for MainForm
public MainForm()
{
    // Create the JFrame
    mainFrame = new JFrame();

    // Create the components
    lblFirstName = new JLabel("First Name:");
    lblSecondName = new JLabel("Second Name:");
    txtFirstName = new JTextField();
    txtSecondName = new JTextField();
    btnConcatName = new JButton("Submit ->");
    txtaOutput = new JTextArea();


}
```

Notice that each Java Swing component is an instance object of a class. Swing is fully OO! Each component class is also part of the class hierarchy sampled in figure CC1.5. There are instance methods that can be invoked on each Swing component to interact with it and set its characteristics. Since all components inherit from the `JComponent` Swing class, many of the methods available for one component may be available for others. For example, several of the methods described for a `JTextField` instance object will also work with `JLabel`, `JButton`, and so on. Some of the more commonly invoked methods are as follows:

- **`JTextField`:** Inherits methods and data fields from the `JTextComponent` class, which inherits from `JComponent`.
  - `.getText()` and `.setText(String str)`: The reference to a `JTextField` instance object does not contain its text, but the text that is displayed in this component is stored in a data field of the `JTextField` instance object. As shown in chapter 8, many classes include accessor ("getter") and mutator ("setter") methods to access the values in an instance object's data fields. `JTextField` instance objects have a "`text`" data field, and the getter/setter methods can interact with it to retrieve or set what is visually displayed in the component.
  - `.setColumns(int colCount)`: You can set the width of the `JTextField` by specifying how many character columns it should display.
  - `.setEditable(boolean bln)`: Toggle the editability of the `JTextField` object by using a `boolean true` or `false` value. Useful when you need to display a value in the `JTextField` but prevent the user from changing it.
  - `.setHorizontalAlignment(int alignValue)`: You can set the justification of the text typed or displayed in a `JTextField` by invoking this method and providing it an int alignment value. The alignment values are enumerated types built into the `JTextField` class (a value represents an int number). These are the following:
    - `JTextField.LEFT`, `JTextField.RIGHT`, `JTextField.CENTER`, `JTextField.LEADING`, `JTextField.TRAILING`
  - `.setForeground(Color clr)`: Allows you to change the font color of the text displayed in a `JTextfield` easily. Provide the color by using the enumerated types built into the `Color` class (`java.awt.Color`). For example,
    - `txtSomeTextField.setForeground(Color.red);`
  - `.setFont(Font ft)`: Allows you to change the `Font` type, the style, and the size of the text in a `JTextField`. You will create a new instance object of the `Font` class (`java.awt.Font`) and provide its constructor with three values, passing the reference to the `Font` object to the `.setFont()` method. For example,
    - `txtSomeTextField`
    `.setFont(new Font("Consolas",Font.BOLD, 12));`
    Note that the font styles are enumerated types within the `Font` class.
  - `.setToolTipText(String str)`: Allows you to display a small pop-up note that can instruct the user as to what the component does and how to use it. For example,

```
txtSomeTextField.setToolTipText("Type in this JTextField");
```

This would display the following in the GUI:

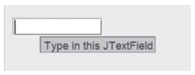
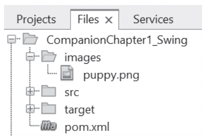


Figure CC1.9. ToolTip Text Box for a `JTextField` in Swing

- **`JLabel`:** Inherits methods and data fields from the `JComponent` class.
  - `.getText()` and `.setText(String str)`: These work the same way as they do for `JTextField` and other text-based components.
  - `.setVerticalAlignment(int alignValue)`: Similar to `.setHorizonalAlignment()`, this method allows you to align the text in the `JLabel` according to a vertical alignment. It uses the enumerated types in `SwingConstants` for the `int alignValue` required by the method's parameter:
    - `SwingConstants.TOP`, `SwingConstants.CENTER`, `SwingConstants.BOTTOM`
  - `.setIcon()`: In addition to having a text label that you can change using `.setText()`, the `.setIcon()` method allows you to use an image as the icon for a `JLabel` as well. This is an easy and indirect way of displaying an image in your Swing GUI application. This is a multistep process:
    - First, copy an image file into your Java project through NetBeans or another IDE of your choice. For example, an "images" folder has been added to my example project (not part of `NameConcatenation`), and the image of a pet has been copied into it like so:



```
Projects   Files ×   Services
CompanionChapter1_Swing
  images
    puppy.png
  src
  target
  pom.xml
```

Figure CC1.10. Location of "Images" Folder in NetBeans Java Swing Project

- Second, create an `ImageIcon` instance object, and use the relative (i.e., local in project) file path to the desired image in the constructor:

```
ImageIcon imgIcon = new ImageIcon("images/puppy.png");
```

- Finally, invoke the `.setIcon()` method on the appropriate `JLabel` to set the `ImageIcon` file as the `JLabel` icon. Later when the form displays, it will look like the below:

```
lblTestLabel.setIcon(imgIcon);
```

Figure CC1.11. Setting an Image in a `JLabel` Control in Java Swing

The same method can be invoked on other components like `JButton` to set icons along with their displayed text:

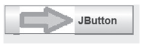


Figure CC1.12. Icon Used in `JButton` Control in Java Swing

You are now ready to manually set the position and sizes of the form's controls. Keep in mind that later usage of layout managers and `JPanel` components will be examined to "automate" this process. The `.setBounds()` method can be used for manual sizing and placement of components in the GUI. One overload version of the `.setBounds()` method looks like this:

```
.setBounds(int x, int y, int width, int height);
```

The `x` and `y` values are the number of pixels to the right (`x`) and the number of pixels down from the top (`y`) that the control will be placed at. This is relative to the component's parent container. Since you will be adding these to the `JFrame`, the `x` and `y` are relative to the top-left corner of the `JFrame` itself (at x = 0 and y = 0). The `.setBounds()` method is invoked for each component.

After the initializations in the constructor, add the following code:

Code Snippet CC1.4

```
    // Size the JFrame window
    mainFrame.setSize(300, 350);

    // Position and Size the Swing Components
    lblFirstName.setBounds(10, 10, 100, 20);
    txtFirstName.setBounds(150, 10, 100, 20);
    lblSecondName.setBounds(10, 40, 100, 20);
    txtSecondName.setBounds(150, 40, 100, 20);
    btnConcatName.setBounds(150, 70, 100, 20);
    txtaOutput.setBounds(10, 100, 250, 200);

    // Add components to the JFrame window
    mainFrame.add(lblFirstName);
    mainFrame.add(txtFirstName);
    mainFrame.add(lblSecondName);
    mainFrame.add(txtSecondName);
    mainFrame.add(btnConcatName);
    mainFrame.add(txtaOutput);

    // Cause text in the JtextArea to wrap
    txtaOutput.setLineWrap(true);

    // Specify what should happen when the main window is closed.
    mainFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    // Specify that we are manually laying out components.
    mainFrame.setLayout(null);
    // Indicate that the JFrame should display itself.
    mainFrame.setVisible(true);
```

A summary of the code has been added:

- First, the method `.setSize()` is invoked upon the `JFrame` instance object to indicate how large the GUI window should be when it eventually is displayed.
- Next, the `.setBounds()` method is invoked for each component to indicate where on the `JFrame` it should display and its size. This can be tedious, for sure, but you can use `.setBounds()` if you need really precise control over the positioning and sizing of components.
- After this, the components have been added to the `JFrame` so that they are "nested" inside its `contentPane` (discussed earlier in the chapter).
- The `JTextArea` has been set to wrap any text that is too long for one line.
- Finally, the Swing application is set so that it will close when the main `JFrame` window is also closed. The form's components are arranged by changing the `layout` data field of the `JFrame` to `null`. Lastly, the `JFrame` is commanded to make itself visible.

The `MainForm.java` class should be ready to run. As a last step, over in the main method of `MainApp.java`, create an instance object of the `MainForm` class, and invoke its constructor:

Full Program CC1.1

```
package com.javaforis.nameconcatenation;

public class MainApp {

    public static void main(String[] args) {

        // Create and Display a MainForm window.
        MainForm mainAppForm = new MainForm();
    }
}
```

When the project is run, the Swing GUI window should display as seen in figure CC1.13:

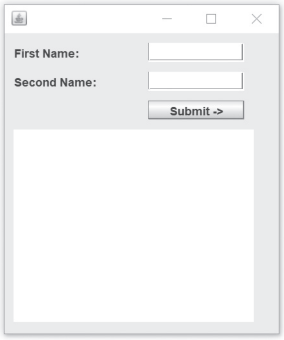


Figure CC1.13. `NameConcatenation` GUI Form Displayed

Using layout managers and `JPanel`: For a very small application like `NameConcatenation`, manually setting the positions and sizes of controls is not too difficult. As our applications grow and become more complex, this could become excessively tedious. It is easier to allow Swing's layout managers to arrange the components in a more automatic fashion. Oracle's documentation does a thorough job of introducing layout managers and discussing each in detail, but discussing a few in detail will be helpful.[*]

Using the `JPanel` class: The `JPanel` class is a great general-use container for other Swing components.[†] A `JPanel` itself can use a layout manager for the components nested inside of it, or the `JPanel` itself could be placed with other components inside another container that uses a layout manager. Think of a `JPanel` like a plate of food: you can place certain items on the plate, and then you can move the *plate itself* to another location. `JPanel` containers work just like this. If you need to and are clever with the way you design your forms, you can create several `JPanel` containers to group components together and arrange them using layout managers. Alter the code in the `MainForm` constructor to add a `JPanel`, place all components within, and add the `JPanel` to the `JFrame` (changes in **bold**):

Code Snippet CC1.5

```
// Zero-Arg Constructor for MainForm
public MainForm()
{
    // Create a JPanel
    JPanel mainPanel = new JPanel();
    mainPanel.setBorder(BorderFactory.createTitledBorder("Name Concatenation"));

    // Create the JFrame
    mainFrame = new JFrame();

    // Create the components
    lblFirstName = new JLabel("First Name:");
    lblSecondName = new JLabel("Second Name:");
    txtFirstName = new JTextField();
    txtSecondName = new JTextField();
    btnConcatName = new JButton("Submit ->");
    txtaOutput = new JTextArea();

    // Size the JFrame window
    mainFrame.setSize(300, 350);

    // Specify the Widths of text-components
    txtFirstName.setColumns(20);
    txtSecondName.setColumns(20);


    txtaOutput.setColumns(20);
    txtaOutput.setRows(10);

    // Add components to the JPanel container
    mainPanel.add(lblFirstName);
```

[*] https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html.
[†] https://docs.oracle.com/javase/tutorial/uiswing/components/panel.html.

```
    mainPanel.add(txtFirstName);
    mainPanel.add(lblSecondName);
    mainPanel.add(txtSecondName);
    mainPanel.add(btnConcatName);
    mainPanel.add(txtaOutput);

    // Add the JPanel to the JFrame
    mainFrame.add(mainPanel);

    // Cause text in the JTextArea to wrap
    txtaOutput.setLineWrap(true);

    // Specify what should happen when the main window is closed.
    mainFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    // Indicate that the JFrame should display itself.
    mainFrame.setVisible(true);

}
```

Notice that in addition to the changes indicated in **bold**, the `null` value for the `JFrame` has been removed. The `JPanel` now has a border specified using the `.setBorder()` method. In summary, this code does the following:

- Creates the individual components, `JFrame`, and `JPanel`
- Adds the components to the `JPanel`
- Adds the `JPanel` to `JFrame`
- Commands the `JFrame` to display

When `MainApp.java` is run, the GUI will display as shown in figure CC1.14. Figure CC1.14 also demonstrates the effect of resizing the form with the mouse:

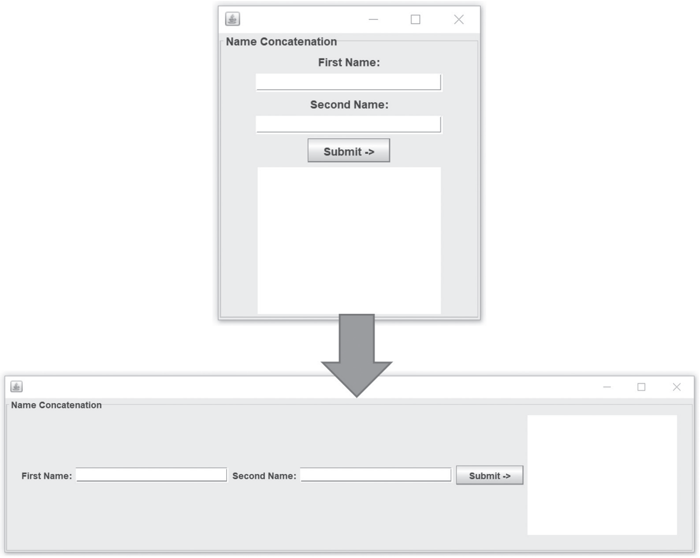


Figure CC1.14. `NameConcatenation` GUI Using a `JPanel` and Display after a Window Resize

Some of the various layout managers can be summarized as follows:

- **`FlowLayout`:** Notice that the components in Figure CC1.14's GUI rearranged themselves to fit the size of the window. This is the default behavior of `JPanel`, which is to use the `FlowLayout` layout manager. `FlowLayout` will arrange all components in a single row, moving down to the next row when there is no more room to display a component. To use `FlowLayout`, allow the default behavior of your `JPanel` container with no extra changes needed.

- **`BorderLayout`:** The `BorderLayout` layout manager allows you to arrange components or containers on a form directionally. For example, you can modify the `NameConcatenation` example in the following way (only the changed code is shown):

Code Snippet CC1.6

```
mainPanel.setLayout(new BorderLayout());

// . . .

// Add components to the JPanel container
mainPanel.add(lblFirstName, BorderLayout.NORTH);
mainPanel.add(txtFirstName, BorderLayout.WEST);
mainPanel.add(lblSecondName, BorderLayout.CENTER);
mainPanel.add(txtSecondName, BorderLayout.EAST);
mainPanel.add(btnConcatName, BorderLayout.SOUTH);
mainPanel.add(txtaOutput, BorderLayout.SOUTH);
```

This will yield the following GUI (shown after a little resizing):

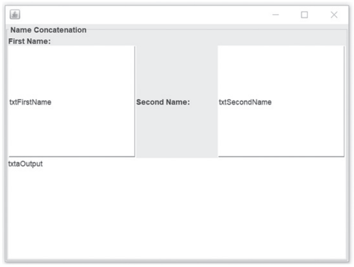


Figure CC1.15. Changes to the Swing Form after Specifying a `BorderLayout` in a `JPanel`

To understand the `BorderLayout` behavior better, the following uses `JButton` components to demonstrate the border directionality of this layout manager (the code is nearly identical to that shown above, just using `JButton` objects instead of the intended `NameConcatenation` components):

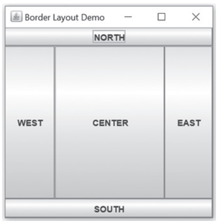


Figure CC1.16. Visually Demonstrating `BorderLayout` in `JPanel` Using `JButton` Controls

A few rules to keep in mind when using the `BorderLayout` layout manager:

- Only one component per directional "area" is allowed. If you place multiple controls, the last one placed will sit "stacked" over any others (you can see this in figure CC1.15: the `JTextArea` is covering the `JButton`).
- Each control or container placed in an area will be expanded in size to fill the space for that area. That is why both of the `JTextField` controls look so large in figure CC1.15 as well as the expanded buttons in figure CC1.16.
- **`CardLayout`:** The `CardLayout` layout manager is handy when you need the ability to quickly swap out one set of controls for another on the same form. It operates as if you were holding playing cards in your hands: You remove one card and pick up the next, swapping one for the other. The Oracle documentation has a great tutorial for using the `CardLayout` layout manager.*

* https://docs.oracle.com/javase/tutorial/uiswing/layout/card.html.

- **BoxLayout:** The BoxLayout manager will arrange components or containers in either a single row or a single column, with no wrapping to another line/column. Consider the following code (note the alternative directionality commented out), and see figure CC1.17 for how this will impact the arrangement of the form's components:

Code Snippet CC1.7

```
    mainPanel.setLayout(
new BoxLayout(mainPanel, BoxLayout.X_AXIS));
    //mainPanel.setLayout(
new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
    // . . .

    // Add them to the JPanel container
    mainPanel.add(lblFirstName);
    mainPanel.add(txtFirstName);
    mainPanel.add(lblSecondName);
    mainPanel.add(txtSecondName);
    mainPanel.add(btnConcatName);
    mainPanel.add(txtaOutput);
```

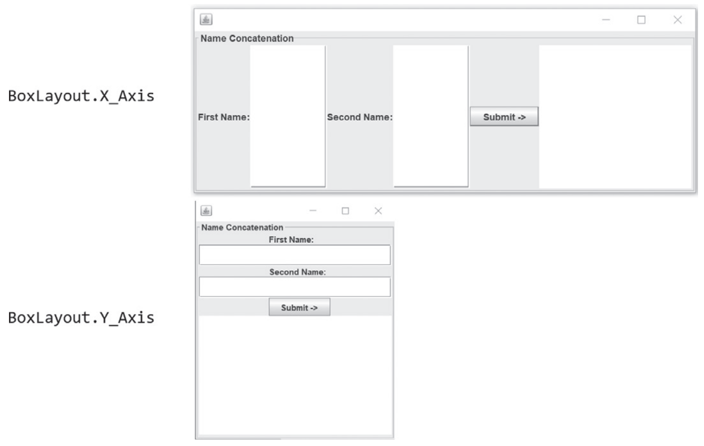


Figure CC1.17. The Impact of BoxLayout Settings on the `NameConcatenation` Components

- **GridLayout:** The GridLayout layout manager allows you to specify a number of columns and rows, and the components will be arranged in a grid-like fashion. The first added component will be placed in the first column, first row; the next in the same row, second column; and so on. For example, consider the following code:

Code Snippet CC1.8

```
    mainPanel.setLayout(new GridLayout(4,2));
    // . . .
    // Add them to the JPanel container
    mainPanel.add(lblFirstName);
    mainPanel.add(txtFirstName);
    mainPanel.add(lblSecondName);
    mainPanel.add(txtSecondName);
    mainPanel.add(btnConcatName);
    mainPanel.add(txtaOutput);
```

This would yield a GUI arrangement of components as follows (the window was resized to show the components better):

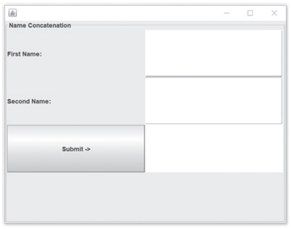


Figure CC1.18. Swing Controls Arranged Using a `GridLayout` Layout Manager

- **`GridBagLayout`:** Similar to `GridBag`, the `GridBagLayout` also arranges components in a grid-like fashion, but `GridBagLayout` gives precise control over the row and column where each component should go. A `GridBagLayout` object is paired with a `GridBagConstraints` object, which defines the rows, columns, any spanning across those (for big components), and other settings. Note in the following example how the row and column are specified for each component added:

Code Snippet CC1.9

```
// Create a JPanel
JPanel mainPanel = new JPanel();
mainPanel.setBorder(BorderFactory.createTitledBorder("Name Concatenation"));

mainPanel.setLayout(new GridBagLayout());

// Create the GridBagConstraints object
GridBagConstraints gbSettings = new GridBagConstraints();

// . . .

// Add components to the JPanel container

// Set the GridBag constraints and specify
//  grid positions of each component.
gbSettings.fill = GridBagConstraints.HORIZONTAL;
gbSettings.gridx = 0; // What row? Indexing starting at 0
gbSettings.gridy = 0; // What col? Indexing starting at 0
mainPanel.add(lblFirstName, gbSettings);

gbSettings.gridx = 1;
gbSettings.gridy = 0;
mainPanel.add(txtFirstName, gbSettings);

gbSettings.fill = GridBagConstraints.HORIZONTAL;
gbSettings.gridx = 0;
gbSettings.gridy = 1;
mainPanel.add(lblSecondName, gbSettings);

gbSettings.gridx = 1;
gbSettings.gridy = 1;
mainPanel.add(txtSecondName, gbSettings);

gbSettings.fill = GridBagConstraints.HORIZONTAL;
gbSettings.gridx = 1;
gbSettings.gridy = 2;
mainPanel.add(btnConcatName, gbSettings);

// Positioning the JtextArea
gbSettings.fill = GridBagConstraints.HORIZONTAL;
gbSettings.gridx = 0;
gbSettings.gridy = 3;
gbSettings.gridwidth = 2; // Span two columns
gbSettings.ipady = 100; // Make this component tall
mainPanel.add(txtaOutput, gbSettings);
```

This will arrange the components on our GUI in the following manner:

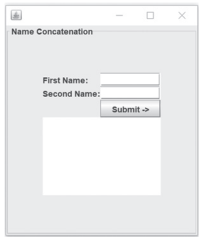


Figure CC1.19. Swing Controls Arranged Using a `GridBagLayout` Layout Manager

Notice that with `GridBagLayout`, the grid is arranged in rows and columns, with the first row and first column starting count using index notation at `0`. So like with a two-dimensional array, the fourth row would be index `3`, the eighth column would be index `7`, and so on. Oracle's documentation for both `GridBagLayout` and `GridBagConstraints` walks through the many additional options for sizing, filling, and positioning as well as absolute versus relative positioning of components using `GridBagLayout`.[*,†]

Final design for `NameConcatenation`: So what is the best layout manager design for the `NameConcatenation` GUI? There are many choices. Using a `BoxLayout` for the overall form in vertical orientation and adding two `JPanel` containers each using a `FlowLayout` manager could work best. The `GridBagLayout` offers the most concise arrangement but could be just as tedious as manually setting sizes and positions. There are lots of options. Here is the full `MainForm` class code listing with changes (in **bold**) to highlight this chosen layout:

Code Snippet CC1.10

```
package com.javaforis.nameconcatenation;

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

import java.util.*;

// Class Declaration
public class MainForm {

    // Class-level declarations of
    //  components
    JFrame mainFrame;
    JLabel lblFirstName;
    JLabel lblSecondName;
    JTextField txtFirstName;
    JTextField txtSecondName;
    JButton btnConcatName;
    JTextArea txtaOutput;

    // Zero-Arg Constructor for MainForm
    public MainForm()
    {
        // Create the JFrame
        mainFrame = new JFrame();
```

[*] https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html.
[†] https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagConstraints.html.

```
        // Create two JPanel containers

        // Input Container
        JPanel inputPanel = new JPanel();
        inputPanel.setBorder(
             BorderFactory
                     .createTitledBorder("Enter Name Information:"));
        // Output Container
        JPanel outputPanel = new JPanel();
        outputPanel.setBorder(
                BorderFactory
                        .createTitledBorder("Application Output:"));

        // Create the components
        lblFirstName = new JLabel("First Name:");
        lblSecondName = new JLabel("Second Name:");
        txtFirstName = new JTextField();
        txtSecondName = new JTextField();
        btnConcatName = new JButton("Submit ->");
        txtaOutput = new JTextArea();

        // Size the JFrame window
        mainFrame.setSize(350, 500);

        txtFirstName.setColumns(20);
        txtSecondName.setColumns(20);

        txtaOutput.setColumns(30);
        txtaOutput.setRows(15);

        // Set the layout manager for the JFrame
        // Box Layout with Vertical orientation.

        // Since we are mixing layout managers, we pass
        //  a reference to the contentPane for the JFrame
        //  to prevent "sharing" errors.
        mainFrame.setLayout(
                new BoxLayout(
                        mainFrame.getContentPane(),
                        BoxLayout.Y_AXIS));

        // Set the Layout Managers for the Input and Output
        //  JPanel containers

        // Overloaded version of FlowLayout constructor used
        inputPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
        outputPanel.setLayout(new FlowLayout(FlowLayout.CENTER));

        // Add components to the inputPanel
        inputPanel.add(lblFirstName);
        inputPanel.add(txtFirstName);
        inputPanel.add(lblSecondName);
        inputPanel.add(txtSecondName);
        inputPanel.add(btnConcatName);

        // Add components to the outputPanel
        outputPanel.add(txtaOutput);

        // Add the JPanel to the JFrame
        mainFrame.add(inputPanel);
        mainFrame.add(outputPanel);

        // Cause text in the JtextArea to wrap
        txtaOutput.setLineWrap(true);

        // Specify what should happen when the main window is closed.
        mainFrame
            .setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        // Indicate that the JFrame should display itself.
        mainFrame.setVisible(true);

        } // End of Constructor MainForm();

} // End of Class MainForm
```

The Java Swing form will display like so:

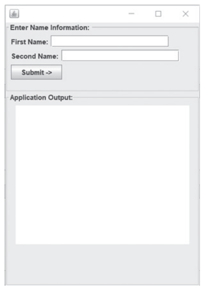


Figure CC1.20. Final Layout for the `NameConcatenation` Java Swing Form

This design is fairly close to the wireframe design from figure CC1.6. Use of the `GridBagLayout` as seen earlier would give an exact re-creation of the design but at the expense of tedious positioning of components. For small applications, this could be an acceptable trade-off, but you will have to decide which design to follow. The first example using simply a `FlowLayout` manager for all the components would work as well but not match the wireframe design too closely. Swing's layout managers provide flexibility!

SUMMARY POINTS

- The Swing framework is used to build Java-based GUI applications using object-oriented syntax that is similar to that of the Java language itself.
- Swing was a useful evolution of Java GUI frameworks after AWT, containing user interface components that would display and behave the same across execution platforms.
- Swing uses a containment hierarchy to manage the relationship between components and containers in memory and to specify how they will be visually displayed.
- Each `JFrame` contains a `JLayeredPane`, `Container`, and `Component` data field to represent and provide access to the various layers displayed on a Swing form.
- Because many of the Swing components are subclasses of the `JComponent` class, they share many methods, particularly methods that can change the look, behavior, and contents of components.
- Developers can manually set the location and sizes of components on a Swing form, or they can use various layout managers to automatically arrange and size components according to layout manager behavior.

QUICK PROBLEMS

1. **Coding:** Build a small Swing application UI that has a `JTextField` and a `JPasswordField`. Add a `JButton` with the caption "Login" (no logic is needed for this problem).
2. **Think:** Why would developers choose to use a layout manager instead of specifying the location and sizes of Swing components in the UI themselves?
3. **Coding:** Build a small Swing application UI that allows a user to build a pizza order. Include `JRadioButton` controls for the selection of cheese type (include a "None" option to account for customer health concerns). Include `JCheckBox` controls for various toppings. Include a `JTextField` for the quantity and a final `JCheckBox` control for pizza size. Finally, add a `JButton` with the caption "Add to Order ->" (no logic is needed for this problem).

CC1.3 Understanding Event Handling in Swing

Learning about the nature of Java Swing, how components work, the layout managers that Swing provides for styling your GUI forms, and the customizations to all of these that can be made are merely half the battle. In order to be useful, our GUI forms must be capable of actions and of executing the logic and information processing tasks. Enabling your GUI-based application to respond to user interaction events by executing code is known as **event handling**. Classes and objects are involved in the event-handling process, giving the developer considerable flexibility in "wiring up" events. Oracle's official event-handling documentation goes into considerable detail, but the basics will be covered in this chapter.*

Event delegation model: Chapter 10 discusses how runtime errors can be mitigated through error handling. Instance objects of class `Exception`, and its subclasses are "thrown" when a specific issue arises. This concept of "throwing" and "catching" objects generated by some event is the same for event handling. Swing's event-handling approach follows the event delegation model. In a Swing form, when a `JButton` is clicked, a letter is typed in a `JTextField`, a mouse cursor hovers over a component or container (or hovers out/leaves), and so on—all these actions will generate an event. In Swing, instance objects of the `EventObject` class and its subclasses will be created and thrown because of user-generated actions. With event delegation, the **component that generates the event does not contain the code that handles the event**. The developer can specify (i.e., delegate) that some other **target** object's code will handle the event and that the **source** object's job (the component that generated the event) is simply to pass the `EventObject` off to the delegated target. This keeps things flexible because in Swing, you can conceivably change how an event for a component is handled on the fly (during runtime) if you wanted. Figure CC1.21 presents an example of event delegation in action.

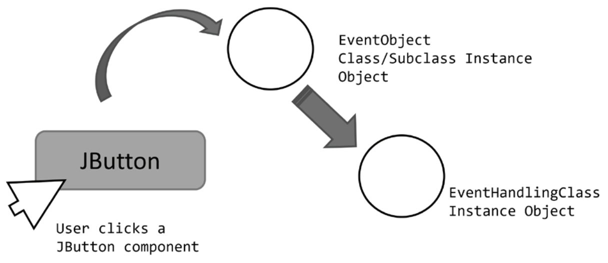


Figure CC1.21. Event Delegation When a `JButton` Is Clicked

In Swing, `JComponent` objects and their subclasses can generate `EventObject` objects through user activity. On a Swing form, controls such as `JButton`, `JTextField`, `JSlider`, and the `JFrame` itself are commonly where user interaction causes event actions to be fired off. Many of these component classes have methods implemented that allow for the **registration of an event handler**, which is how event delegation occurs in Swing. For example, when a user clicks a `JButton` on a form, it will create and throw (or "fire off") an `EventObject` instance object. You get to delegate "who" will catch and handle the thrown `EventObject` object. Event handling for a `JComponent` object is accomplished in two overall steps:

* https://docs.oracle.com/javase/tutorial/uiswing/events/index.html.

- Based on the event to the handled, invoke the appropriate event handling method on the `JComponent` object (e.g., a `JButton`).
- Include a reference to an event-handling class object as a parameter to the invoked method. This is the object you will delegate the event handling task to.

Three ways of handling events in Swing: A simple example using a `JButton` will help. You can "wire up" the "`Submit ->`" `JButton` on the `NameConcatenation` Swing form so that when a user clicks it, test output is printed to the `JTextArea` on the form (we will implement the actual concatenation logic afterward). To handle events, we need several things to be in place:

1. A **Source** object: In this case, the `JButton` instance object that is clicked is our source.
2. A **Target** object: This will be the instance object of a class that can handle `EventObject` objects. You will need to create this class and instantiate an object from it. There are three ways to do this:
   a. An inner (i.e., nested) event-handling class
   b. An anonymous inner event-handling class
   c. The use of a lambda expression (*think:* shortcut syntax)

   A class can handle events when the class implements an event-listening **interface** (see companion chapter 4 for more detail on interfaces in Java).
3. Invoking the **event registration method**: Choose and invoke the appropriate event-handling method upon the **source** object and pass a reference to the **target** object as a parameter. This will "register" an instance object of the event-handling class you have created to be a listener for specific types of `EventObject` objects when they are thrown.

Best practice: Any GUI components that the event-handling object may change (like changing text, color, etc.) should be defined at the data field level due to scope.

Recall the `JButton` object on the form. It has an event registration method defined for `JButton` clicks, called `.addActionListener()`, that can be invoked when needed. So all that is needed is to define an event-handling class and instantiate one of its objects.

There are three ways to create this class:

Event-handling technique #1—inner event-handling class: Java classes allow for another class to be defined within their class body (but outside of any method). These are considered nested or inner classes. Since an object must handle events and classes are the definitions for objects, we must define an event-handling method. In our code listing, after the end of the `MainForm` constructor, write the code for the following class (you will want to make sure this code is inside the `MainForm` class):

Code Snippet CC1.11

```
// . . .
} // End of MainForm() Constructor

// Inner Event-Handling Class Definition
class ButtonClickHandler implements ActionListener{
    public void actionPerformed(ActionEvent e)
    {
        txtaOutput.append("Event Handled!");
    }
}
```

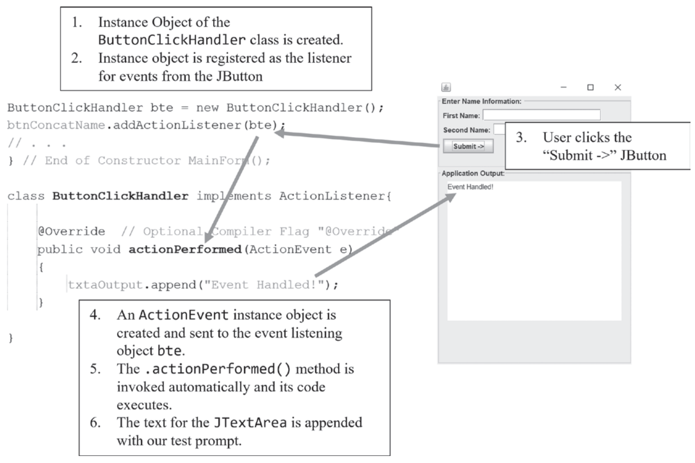
A couple of things to note here:

- The `implements ActionListener` part of the class header tells Java that this class implements an interface known as `ActionListener`.[*] If you want an inner class to handle events in your GUI, you must have that class definition implement one of the event listener interfaces. An **interface** (companion chapter 4) is simply a way of relating classes together that are not normally related to one another at all (not in an inheritance relationship).

[*] https://docs.oracle.com/javase/7/docs/api/java/awt/event/ActionListener.html.

- If a class implements an interface, it is usually **required** to include a definition for any methods that are part of the interface class's definition. Notice we have a `.actionPerformed()` method. This is defined in the `ActionListener` interface. Since our class implements this interface, you are forced (i.e., Java law!) to implement a **local, concrete definition of this method (more on this in companion chapter 4)**.
- Notice that after all this, the code we want to have execute (placing some text in the `JTextArea`) is the code within the `.actionPerformed()` method. When an `EventObject` object is passed to the instance object of this class, the `.actionPerformed()` method will **automatically** be called to handle the event.

Now back in the `MainForm` constructor, you can create an instance object of this event-handling inner class and register that object as the listener for any click events from the `JButton` control:



1. Instance Object of the `ButtonClickHandler` class is created.
2. Instance object is registered as the listener for events from the JButton

```
ButtonClickHandler bte = new ButtonClickHandler();
btnConcatName.addActionListener(bte);
// . . .
} // End of Constructor MainForm();

class ButtonClickHandler implements ActionListener{

    @Override  // Optional Compiler Flag "@Override"
    public void actionPerformed(ActionEvent e)
    {
        txtaOutput.append("Event Handled!");
    }

}
```

3. User clicks the "Submit ->" JButton

4. An `ActionEvent` instance object is created and sent to the event listening object `bte`.
5. The `.actionPerformed()` method is invoked automatically and its code executes.
6. The text for the `JTextArea` is appended with our test prompt.

Figure CC1.22. Event-Handling Sequence Using an Inner Listener Class

Code Snippet CC1.12

```
ButtonClickHandler bte = new ButtonClickHandler();
btnConcatName.addActionListener(bte);
```

Figure CC1.22 details the sequence of events that happens when the user clicks the `JButton`.

Here is the full code listing for `MainForm` reference (with the event-handling changes in **bold**):

Code Snippet CC1.13

```
package com.javaforis.nameconcatenation;

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

import java.util.*;

// Class Declaration
public class MainForm {

    // Class-level declarations of
    //  components
    JFrame mainFrame;
    JLabel lblFirstName;
    JLabel lblSecondName;
    JTextField txtFirstName;
    JTextField txtSecondName;
    JButton btnConcatName;
    JTextArea txtaOutput;

    // Zero-Arg Constructor for MainForm
    public MainForm()
    {
        // Create the JFrame
        mainFrame = new JFrame();

        // Create two JPanel containers

        // Input Container
        JPanel inputPanel = new JPanel();
        inputPanel.setBorder(
                BorderFactory
                     .createTitledBorder("Enter Name Information: "));
        // Output Container
        JPanel outputPanel = new JPanel();
        outputPanel.setBorder(
                BorderFactory
                        .createTitledBorder("Application Output:"));

        // Create the components
        lblFirstName = new JLabel("First Name:");
        lblSecondName = new JLabel("Second Name:");
        txtFirstName = new JTextField();
        txtSecondName = new JTextField();
        btnConcatName = new JButton("Submit ->");
        txtaOutput = new JtextArea();

        // Size the JFrame window
        mainFrame.setSize(350, 500);

        txtFirstName.setColumns(20);
        txtSecondName.setColumns(20);

        txtaOutput.setColumns(30);
        txtaOutput.setRows(15);

        // Set the layout manager for the JFrame
        // Box Layout with Vertical orientation.

        // Since we are mixing layout managers, we pass
        //  a reference to the contentPane for the JFrame
        //  to prevent "sharing" errors.
        mainFrame.setLayout(
                new BoxLayout(
                        mainFrame.getContentPane(),
                        BoxLayout.Y_AXIS));

        // Set the Layout Managers for the Input and Output
        //  JPanel containers

        // Overloaded version of FlowLayout constructor used
        inputPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
        outputPanel.setLayout(new FlowLayout(FlowLayout.CENTER));

        // Add components to the inputPanel
        inputPanel.add(lblFirstName);
        inputPanel.add(txtFirstName);
        inputPanel.add(lblSecondName);
        inputPanel.add(txtSecondName);
        inputPanel.add(btnConcatName);

        // Add components to the outputPanel
        outputPanel.add(txtaOutput);

        // Add the JPanel to the JFrame
        mainFrame.add(inputPanel);
        mainFrame.add(outputPanel);

        // Cause text in the JtextArea to wrap
        txtaOutput.setLineWrap(true);

        // Specify what should happen when the main window is closed.
        mainFrame
                .setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        // Indicate that the JFrame should display itself.
        mainFrame.setVisible(true);

        ButtonClickHandler bte = new ButtonClickHandler();
        btnConcatName.addActionListener(bte);

} // End of Constructor MainForm();

    class ButtonClickHandler implements ActionListener{

        @Override  // Optional Compiler Flag
        public void actionPerformed(ActionEvent e)
        {
            txtaOutput.append("Event Handled!");
        }
    }

} // End of Class MainForm
```

Event-handling technique #2—anonymous inner event-handling class: Swing's event-handling syntax simplifies technique #1 a bit by combining both the class definition and event-handling object instantiation into one step and placing this in-line within the event registration method parameter of the `JButton` object. In this example, you do not need to add an inner class placed outside the `MainForm` constructor. Instead, define and use the class all in one step, and **do this within the parameter list** of the `.addActionListener()` method like so:

Code Snippet CC1.14

```
btnConcatName.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        txtaOutput.append("Mischief Managed!");
    }
}); // end of the .addActionListener parentheses
```

Java knows what you are trying to do and allows this shortcut. Notice that you use the interface `ActionListener`, but the "inner" class is not named as it was in technique #1. This is known as an **anonymous inner class**—anonymous because it is nameless. Notice also that this occurs within the parameter list of the `.addActionListener()` method. A new, nameless class is being defined, one that implements the `ActionListener` interface, and a new instance object of that anonymous class is being instantiated and registered as the event handler for `JButton` clicks . . . all in one step! The full class we defined outside the constructor earlier has been removed and is no longer needed. The end of the constructor and class now looks like this:

Code Snippet CC1.15

```
        // Inside MainForm's constructor:
        // . . .
        btnConcatName.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                txtaOutput.append("Mischief Managed!");
            }
        }); // end of the .addActionListener parentheses

    } // End of Constructor MainForm();

} // End of Class MainForm
```

Event-handling technique #3—event handling using a lambda expression: This technique is even easier. In Java (and in most modern OO languages), **lambda expressions** have become very popular because of their very short syntax and ease of use. The easiest way to understand them is to think about the parameter inputs and code body of any Java method. A lambda expression simplifies a method into one line of code:

```
( input parameter(s) ) -> { expression }
```

Notice that the arrow is actually part of the syntax; it is a **lambda operator**. Lambda expressions in Java are normally passed to another method as a parameter. With event handling, a lambda expression can be used where a reference to an instance object is expected and where that instance object is one of a class that implements an interface. Sound familiar? You can use a lambda expression to replace the anonymous inner event-handling class with a **much** simpler syntax:

Code Snippet CC1.16

```
btnConcatName.addActionListener(e -> {
    txtaOutput.append("Event Handled!");
});
```

That's it! Java knows that you are accepting a single parameter and code that is executing. You are using this lambda expression where a reference to an event-handling class instance object (that implements the `ActionListener` interface) is expected and where that class implements the `.actionPerformed()` method. The `e` parameter is understood to be the `EventObject` instance object "caught" by the implied `.actionPerformed()` method. Java figures out the rest and executes the code in the lambda expression when events are generated by `JButton` clicks! This is by far the easiest way to handle events in Swing.

When each event-handling technique should be used: For some of the event-listening interface classes like `ActionListener`, there is only one method defined in the interface that will execute your code to "handle" the method. For example, with ActionListener, the method `.actionPerformed()` is the only listener method defined in the code. Since this works well with clicks of `JButton`, then whenever you need to handle `JButton` clicks by the user, a **lambda expression** will work well in your code. Anytime a listener interface class has only one listener method defined, you can use the lambda expression.

In most other cases, listener interface classes may have more than one listener method defined. Table CC1.2 gives an example of some of the other listeners that can be used with other `JComponent` objects in a Swing GUI. The official Oracle documentation gives a full listing of all the listener interfaces and their defined methods as well as a listing of various components and their common listeners.[*,†] For example, if you are handling events like a mouse moving over a component or moving away, these would be two different events. Table CC1.2 shows that there is a listener method defined for each as well as for other mouse events.

A rule of interface classes in Java is that a local, fully coded definition for all abstract methods defined within that interface must be implemented. Since, for example, the `MouseListener` interface contains multiple listener methods, we have to define each and every one of these. Since they all contain an event parameter "`e`," the lambda expression technique will not work here, and we will need to use **technique #1** and add an **inner event-handling class**. In the following code, a mouse event-handling class is defined, and an instance object of the class is created, registering it with the `JButton`'s `MouseListener` event registration method:

[*] https://docs.oracle.com/javase/tutorial/uiswing/events/api.html.

[†] https://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html.

Table CC1.2. Some Common UI Events and How to Handle Them in Swing

Common events	Event object thrown	Registration method, parameter interface, and listener methods for interface

Button click	`ActionEvent`	`.addActionListener(ActionListener lis)`
		`.actionPerformed(ActionEvent ae)`

Mouse events	`MouseEvent`	`.addMouseListener(MouseListener mls)`
		`.mouseClicked(MouseEvent me)`
		`.mouseEntered(MouseEvent me)`
		`.mouseExited(MouseEvent me)`
		`.mousePressed(MouseEvent me)`
		`.mouseReleased(MouseEvent me)`

Keyboard key pressed	`KeyEvent`	`.addKeyListener(KeyListener kl)`
		`.keyPressed(KeyEvent ke)`
		`.keyReleased(KeyEvent ke)`
		`.keyTyped(KeyEvent ke)`

List item clicked/selected	`ListSelectionEvent`	`.addListSelectionListener(ListSelectionListener lsl)`
		`.valueChanged(ListSelectionEvent lse)`

Code Snippet CC1.17

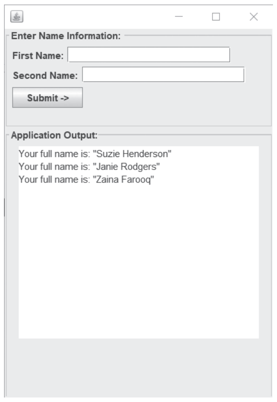
```
// . . .
btnConcatName.addMouseListener(
    new MouseMotionHandler());

} // End of Constructor MainForm();

// Inner Mouse Event-Handling Class
class MouseMotionHandler implements MouseListener
{
    public void mouseClicked(MouseEvent e)
    {
        // Do nothing.
    }

    public void mouseEntered(MouseEvent e)
    {
        btnConcatName.setBackground(Color.ORANGE);
    }

    public void mouseExited(MouseEvent e)
    {
        btnConcatName.setBackground(Color.LIGHT_GRAY);
    }

    public void mousePressed(MouseEvent e)
    {
        // Do nothing.
    }

    public void mouseReleased(MouseEvent e)
    {
        // Do nothing.
    }

} // End of MouseMotionHandler class

} // End of Class MainForm
```

Finishing basic functionality for `NameConcatenation`: Now that you have a good overview of event handling, the test print to the `JTextArea` can be reimplemented with the actual purpose of this application. The code listing for the event-handling registration method would look like the following:

Code Snippet CC1.18

```
btnConcatName.addActionListener(e -> {
    String fullName = "Your full name is: \"";
    fullName += txtFirstName.getText();
    fullName += " " + txtSecondName.getText() + "\"";
    if (txtaOutput.getText().length() == 0)
        txtaOutput.append(fullName);
    else
        txtaOutput.append("\n" + fullName);

    txtFirstName.setText("");
    txtSecondName.setText("");
});
```

Notice that quotation marks are escaped within the displayed String, the result is appended to the `JTextArea`, and the `JTextField` components are cleared out after each use. This gives the user the visual cue that the GUI is ready for their next interaction/usage. Figure CC1.23 shows the GUI after a few names are entered.



```
Your full name is: "Suzie Henderson"
Your full name is: "Janie Rodgers"
Your full name is: "Zaina Farooq"
```

Figure CC1.23. The Name Concatenation GUI Application after Example Usage

Handling events thrown by keyboard events: An extensive discussion of how to handle events of all types from all components would be beyond the scope of this chapter. The official Oracle documentation and many exemplary tutorials online can take you through those examples thoroughly.[*,†,‡] That said, you can now leverage table CC1.2 and your knowledge of event-listening interfaces, their methods, how to create classes that use these interfaces, and how to register them to handle events. With some clever coding, the `JButton` can be removed altogether, and the concatenation action occurs when the user hits the Enter key on their keyboard when typing is completed. The modified code for this would look like this (with changes in **bold**):

Code Snippet CC1.19

```
package com.javaforis.nameconcatenation;

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

import java.util.*;

// Class Declaration
public class MainForm {

    // Class-level declarations of
    //  components
    JFrame mainFrame;
    JLabel lblFirstName;
    JLabel lblSecondName;
    JTextField txtFirstName;
    JTextField txtSecondName;
    //JButton btnConcatName;
```

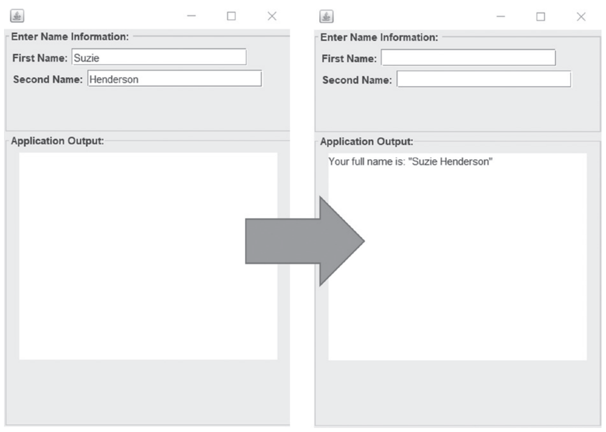
[*] https://docs.oracle.com/javase/tutorial/uiswing/events/handling.html.
[†] https://www.javatpoint.com/event-handling-in-java.
[‡] https://www.tutorialspoint.com/swing/swing_event_handling.htm.

```
    JTextArea txtaOutput;

    // Zero-Arg Constructor for MainForm
    public MainForm()
    {
        // Create the JFrame
        mainFrame = new JFrame();

        // Create two JPanel containers

        // Input Container
        JPanel inputPanel = new JPanel();
        inputPanel.setBorder(
                BorderFactory
                     .createTitledBorder("Enter Name Information: "));
        // Output Container
        JPanel outputPanel = new JPanel();
        outputPanel.setBorder(
                BorderFactory
                        .createTitledBorder("Application Output:"));

        // Create the components
        lblFirstName = new JLabel("First Name:");
        lblSecondName = new JLabel("Second Name:");
        txtFirstName = new JTextField();
        txtSecondName = new JTextField();
        //btnConcatName = new JButton("Submit ->");
        txtaOutput = new JtextArea();

        // Size the JFrame window
        mainFrame.setSize(350, 500);

        txtFirstName.setColumns(20);
        txtSecondName.setColumns(20);

        txtaOutput.setColumns(30);
        txtaOutput.setRows(15);

        // Set the layout manager for the JFrame
        // Box Layout with Vertical orientation.

        // Since we are mixing layout managers, we pass
        //  a reference to the contentPane for the JFrame
        //  to prevent "sharing" errors.
        mainFrame.setLayout(
                new BoxLayout(
                        mainFrame.getContentPane(),
                        BoxLayout.Y_AXIS));

        // Set the Layout Managers for the Input and Output
        //  JPanel containers

        // Overloaded version of FlowLayout constructor used
        inputPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
        outputPanel.setLayout(new FlowLayout(FlowLayout.CENTER));

        // Add components to the inputPanel
        inputPanel.add(lblFirstName);
        inputPanel.add(txtFirstName);
        inputPanel.add(lblSecondName);
        inputPanel.add(txtSecondName);
        //inputPanel.add(btnConcatName);

        // Add components to the outputPanel
        outputPanel.add(txtaOutput);

        // Add the JPanel to the JFrame
        mainFrame.add(inputPanel);
        mainFrame.add(outputPanel);

        // Cause text in the JtextArea to wrap
        txtaOutput.setLineWrap(true);

        // Specify what should happen when the main window is closed.
        mainFrame
                .setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        // Indicate that the JFrame should display itself.
        mainFrame.setVisible(true);

        // Register a KeyEventHandler instance
        //  object for txtSecondName
        txtSecondName.addKeyListener(
                new KeyEventHandler());

    } // End of Constructor MainForm();

    class KeyEventHandler implements KeyListener
    {
        public void keyPressed(KeyEvent e)
        {
            if (e.getKeyCode() == KeyEvent.VK_ENTER)
            {
            String fullName = "Your full name is: \"";
            fullName += txtFirstName.getText();
            fullName += " " + txtSecondName.getText() + "\"";
            if (txtaOutput.getText().length() == 0)
                txtaOutput.append(fullName);
            else
                txtaOutput.append("\n" + fullName);

            txtFirstName.setText("");
            txtSecondName.setText("");
            }

        }

        public void keyReleased(KeyEvent e)
        {
            // Do Nothing
        }

        public void keyTyped(KeyEvent e)
        {
            // Do Nothing
        }
    }

} // End of Class MainForm
```

Notice that the `JButton` declaration and settings have been commented out and its associated lambda expression, mouse listener registration, and mouse event-handling classes have been removed. The code now handles a `KeyEvent` event object for `txtSecondName`. In the event-handling logic, `e` is the parameter passed and provides a reference to the `KeyEvent` instance object thrown by the press of a key in the `JTextField`. The event is then handled in the following manner:

- For each key press, the code checks to see if the specific key pressed is the Enter key. First, the `.getKeyCode()` method is invoked on the caught `KeyEvent` object, which returns an `int` value.
- Next, this `int` value is compared to an enumerated type in the `KeyEvent` class. If the value `VK_ENTER` matches the key code from the `KeyEvent` object, the names are concatenated and printed to the `JTextArea`. Simple!

Figure CC1.24 shows the modified form both before and after user interaction. Though this is a clever way to kick off the concatenation action, you probably should always have a button somewhere on your Swing form. Users typically expect them, and buttons visually send users a clear "What should I do?" signal in your application.


Enter Name Information:
First Name: Suzie
Second Name: Henderson
Application Output:

Enter Name Information:
First Name:
Second Name:
Application Output:
Your full name is: "Suzie Henderson"

Figure CC1.24. Modified Swing GUI Form—before and after User Interaction

SUMMARY POINTS

- Swing uses the event delegation model, which passes the responsibility for handling an event off to code other than the object/code that caused the event to be thrown.
- Similar to the `Exception` classes in Java, the `EventObject` class has specific event-related subclasses to represent various actions that can occur in a Swing GUI.
- In an event-handling scenario, some component or object will serve as the source and cause an `EventObject` instance object to be "thrown." This event-related instance object will then be "caught" and handled by the instance object of a class listening for objects of that event type coming from a specific source.
- Some event-related interface classes have multiple abstract methods that must be defined by an event-handling class you create. For these, it is useful to use an inner event-handling class (technique #1).
- An inline, anonymous inner event-handling class can be useful in defining and instantiating an event-handling class using more concise syntax (technique #2).
- A lambda expression can "shortcut" the syntax for defining, instantiating, and registering an event-handling class to a source component in a very concise manner. Lambda expressions are handy when the event-related interface class has only one abstract method to be defined.

QUICK PROBLEMS

1. **Coding:** Implement the logic for quick problem #1 in section CC1.2. Use a String for the username and password of your choosing, and compare what the user types into these values. Display the success or failure of the login in a `JLabel` on the form.
2. **Think:** What is the relationship between an event-handling class that you write, the event-related interface class used for various types of events, and the `EventObject` class and its subclasses?
3. **Coding:** Implement the logic for quick problem #3 in section CC1.2. Use a `JTextArea` to display a summary of the pizza order placed. Determine costs for each of the cheese, topping, and size options, and display the order total as configured on the form.

Summary

In this chapter, we have covered both the basics and some advanced techniques in using the Swing GUI framework to build a modern, functional GUI-based Java application. The skills used in this chapter were highly integrative, using concepts from all the chapters you have studied throughout this text. Swing is still a widely used GUI framework in Java and is particularly in use in many corporate development environments today. Your studies and practice in this area of Java will put you ahead of student peers in the market who may not be familiar with the many features of Swing. Additionally, as information systems professionals, experience gained in concepts involving the "front-end" design of an information system is invaluable toward your becoming a well-rounded expert in your field. How easy users find your system to learn and use, how the system fulfills business requirements, and how helpful users perceive the system to be will all be heavily influenced by the design of the form, handling of events, and variety of controls plus the manner in which information is displayed in the application. Spending time in this area will help your future IS projects succeed! In the next and last chapter of this textbook, you will explore how to integrate database technology into your Java application to fully finish out the functionality of a contemporary, business-ready system.

Practice Problems

Terminology

Match the following terms from the chapter with their most appropriate definition:

1. Graphical user interface	a. The structured relationship defined for containers and components in a Java Swing application. Helps define the location in memory for visibly displayed UI components.
2. Swing	b. Method defined for a Java Swing `JComponent` class that allows the developer to specify which event-handling class instance object should listen for and handle events generated by the `JComponent`.
3. Abstract Window Toolkit	c. A visualization developed by information systems analysts and developers to help in the planning of the look, feel, and functionality of a to-be system.
4. API	d. The base object in a containment hierarchy.
5. Containment hierarchy	e. A hierarchical relationship between Swing objects in memory where the reference to one object is stored within another, forming a chain of containment.
6. `JComponent`	f. A Java Swing layout manager that will display components in a single row until the horizontal space is exhausted, moving the next down to another row below the previous.
7. Root	g. Instance object of a class that implements an event-handling interface. This object receives and processes the `ActionEvent` object generated by a control or process in a Java Swing application.
8. `JLayeredPane`	h. Object in the Java Swing containment hierarchy that enables component nesting.
9. Glass pane	i. A Java Swing layout manager that allows the developer to add Swing components to a container in grid-like fashion after specifying the number of rows and columns in the grid.
10. Container	j. The oldest of the official Java GUI frameworks. Was considered "heavy" and resource inefficient. Often spurred additional work by developers using it who wanted to ensure universal look and feel for their applications across platforms.
11. Wireframe model	k. A Java Swing layout manager that allows the developer to choose within which specific grid-located row and column a Swing component will be placed.
12. Nesting	l. A nested component in the containment hierarchy that allows for a depth aspect of Swing GUI components to be specified.
13. `FlowLayout`	m. The visual display of controls and output that a user can interact with (mouse, keyboard, touch), allowing them to execute the logic of an application without the need for typed commands or execution syntax.

14. `BorderLayout`	n. "Shortcut" syntax that allows the developer to, among other syntax-summarizing tasks, define and instantiate an event-handling class along with its abstract method in a short, simple syntax style.
15. `GridBagLayout`	o. Top-level class from which data-type classes for Java Swing events are defined.
16. `GridLayout`	p. The process where systems analysts and developers allow targeted and appropriate end users to use a system still in development and provide development regarding topics such as ease of use and efficacy.
17. Event delegation model	q. A component data field of the `JRootPane` class. Overlays the entire visible `JFrame` area and is not visible at first and allows for interception of events and overlaid drawing of the GUI.
18. Event handler	r. The top-level Java Swing class for all UI components, excluding the container classes.
19. Event registration method	s. Stands for application programming interface, a defined set of functions/methods/commands for building upon and interacting with a development language platform.
20. `ActionEvent` class	t. A Swing component that allows dynamic display of different components and containers depending on which tab the user has selected.
21. Lambda expression	u. A popular GUI framework for Java developed by Oracle and still native to the Java language. Extends and improves upon the AWT and defines its own controls for universal look and feel across execution platforms.
22. User testing	v. An event-handling approach implemented by Java Swing that passes responsibility for handling an event to code located elsewhere apart from the object that generated the event.
23. `JTabbedPane`	w. A Java Swing layout manager that allows the developer to specify directional positioning ("north," "south," "east," etc.) of controls in a container.

Find the Error

In each of the following problems, carefully examine the code given, and determine the error(s)/issue(s) with each. Keep in mind, the error(s) could be syntax (code) or logic (intended outcome) based or both! Assume any Swing code is listed within an appropriate constructor or other valid location unless stated otherwise. Where ellipses (. . .) are indicated, assume that necessary but unrelated code has been omitted for clarity.

1.

```
JFrame mainframe = new JFrame();
JTextField tOne = new JTextField();
JTextField tTwo = new JTextField();
mainPanel.setLayout(new GridLayout());
gP.add(tOne, 0, 0);
gP.add(tOne, 0, 1);
. . .
mainFrame.setVisible(true);
```

2.

```
JTextField firstNum = new JTextField ();
JTextField secondNum = new JTextField ();
JButton btnAdd = new Button("Concat Names");
. . .
btnAdd(e -> (
    int numA = firstNum;
    int numB = secondNum;
    System.out.println(numA * numB);

));
```

3.

```
TextField tOne = new JTextField();
TextField tTwo = new JTextField();
JButton btnAdd = new JButton{};
JPanel somePanel = new JPanel();
GridBagConstraints gp = new GridBagConstraints(GridBagLayout new);
gP.add(JTextField(), 1, 0);
gP.add(JTextField(), 0, 1);
```

4.

```
JComboBox cmboNames = new ComboBox();
JTextArea txtNames = new JTextField();
cmboNames.add("Suzie");
cmboNames.add("Breyea");
. . .
for (int i>0;i<10;i++)
{
    txtNames.setText(name.toString());
}
```

5.

```
jButton btnFinish = new jButton();
jLabel lblStatus = new Label("");
. . .
btnFinish.setOnKeyListener(new interface handle(
   Label.getText("Button Finished Clicked!");
 });
```

6.

```
AnchorPane aPane = new AnchorPane(Pos.Center);
aPane.add(btnSubmit, 0, 0);
AnchorPane.setAlignment(btnSubmit, 0.0);
gPane.add(aPane, anchorRight);
```

7. (Use Companion Chapter 1 Opening Scenario Application Supplemental for this problem)

```
JButton btnUpdateList = new JButton();
JComboBox cmboList = new JcmboBox();
ArrayList<Integer> numberList = new ArrayList<>();
. . .
// code that gathers numbers from the UI and
// adds them to numberList
. . .
btnUpdateList.addActionListener(e -> {
    for (Integer int : numberList)
    {
        cmboList.add(int);
    }
});
```

8. (Use Companion Chapter 1 Opening Scenario Application Supplemental for this problem)

```
JTabbedPane classroomTabs = new JTabbedPane()
classroomTabs.newTab("Students");
classroomTabs.newTab("Faculty");
somePanel.setContent(new JTabPane());
```

9. (Use Companion Chapter 1 Opening Scenario Application Supplemental for this problem)

```
// Assume the Person class is included and
// any methods invoked upon its objects are
// defined.
Person personOne = new Person();
Person personTwo = new Person();
JLabel lblStatusArea = new JLabel("");
JComboBox personList = new JComboBox();
personList.addItem(personOne);
personList.addItem(personTwo);
. . .
lblStatusArea.setText(personList
        .getSelectionModel()
        .getSelectedIndex()
        (getPersonName()));
```

10. (Use Companion Chapter 1 Opening Scenario Application Supplemental for this problem)

```
JButton addTextFields = new JButton("Add New Field");
int col = 0, row = 0;
somePanel.setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();
// . . .
gbc.gridx = col;
gbc.gridy = row;
addTextFields.addActionListener(e -> {
    somePanel.add(new JTextField(), gbc);
    TextField().setText("TextField #" + row);
});
```

Think about It

1. What advantages does a GUI application have over a console-based application? What advantages does a console-based application have over a GUI one?
2. Why was AWT's lack of portability between platforms a hindrance for developers?
3. What characteristics does the Swing Java GUI library share with JavaFX? What differentiates JavaFX from Swing?
4. How is Swing an object-oriented GUI framework?
5. How are components related to one another in the Swing container hierarchy? How does this help make development of Swing applications easier to understand?
6. What is the root component in a Swing application, and why is it important? What relationship does it have to the rest of the components and containers in the application?
7. Where does code execution start in a Swing application?
8. What is the difference between a `JTextField` and a `JTextArea` in a Swing application?
9. What do layout managers do in a Swing application?
10. What is the data type returned by the `.getSelectedItem()` method in `JComboBox`? What is the relationship between the referenced object and the `JComboBox` itself?
11. What is the general approach that Swing uses to handle GUI events?
12. How is event handling in Swing similar to error handling in Java?
13. Why is it beneficial to have subclasses for the `EventObject` class?
14. What type of parameters do event-handling registration methods for components accept? Where does the parameter come from?
15. What are the benefits of using a lambda expression in an event-handling method?
16. How is creating a second `JFrame` in your Swing application related to the object-oriented nature of Swing?
17. What is important to keep in mind when pulling data from and putting values into Swing `JTextField` controls?
18. What is the difference between a `JComboBox` and a `JList` control?

Short Syntax Problems

1. Develop a Swing application that will accept the entry of a user's name. Include a `JButton` that when clicked, the user's name will be displayed in a `JLabel` on the form, but with the letters printed in reverse.
2. Develop a Swing application that will allow a user to type in a long paragraph in a `JTextArea`. Ensure that the text wraps within the `JTextArea`. Include a `JButton` that when clicked will pop up a secondary form that will report the number of each of the English vowels (`a`, `e`, `i`, `o`, `u`, and `y`) found in the paragraph entered.
3. Develop a Swing application that will ask the user to enter two numbers. The application will generate a quantity of random numbers between those two values. Include the quantities five through ten in a `JComboBox` on the form. Include a `JButton` that when clicked will read the quantity selected in the `JComboBox` and generate that quantity of random numbers between the first and second entered somewhere on the form. Include a small bit of logic making sure the second number is larger than the first. Include a warning on the form if this is **not** the case, and do not let number generation occur if so.
4. Develop a Swing application that allows the user to enter a paragraph in a `JTextArea`. In a `JComboBox`, include three text editing options: "Remove all blank spaces," "Convert to lowercase," "Remove all vowels." Include a `JButton` that when clicked will carry out the currently selected command and edit the contents of the `JTextArea` accordingly.
5. Develop a Swing application that includes a single `JButton` control in the center. When the `JButton` is clicked, it will move to a random x and y location on the form. *Hint:* Use a `JPanel`, get the current width and height of the `JFrame`, and use the `.setBounds()` method.

Full Problems

1. Complete the functionality for the "`Grocery Inventory Management 0.1`" Swing application by implementing the following:
   a. Research how to programmatically exit the application, and enable the App -> Quit App `JMenuItem` to close the application out.
   b. Add a new `Menu` called "Summary" with a `MenuItem` called "Generate Summary." When the user clicks this `MenuItem`, a secondary form will pop up and list the total revenue that would occur if every item in the grocery store was sold. Make sure to format the information appropriately as a monetary amount.
2. Researchers in the information systems discipline generally include references to their research that follow the style detailed in the *Publication Manual of the American Psychological Association*. A template for the reference style for a regular academic journal article looks like the following (minus any alignment requirements):

   Author, A. A., Author, B. B., & Author, C. C. (Year). Title of article. *Title of Periodical, volume number*(issue number), pages.

   Develop a Swing application that prompts for the following information:
   - Year of article
   - Title of article
   - Title of the periodical
   - Volume number
   - Issue number
   - Pages (in style ##–##)
   - First author's name

   Include a `JButton` that allows the user to dynamically add another `JTextField` if there is a second author, another if there is a third, up to six authors. When the "Add Author" button is clicked, a new `JTextField` should be dynamically instantiated and displayed. A final `JButton`, "Generate Reference," should display all the entered information according to the reference style above in a `JTextArea` so that the user can copy and paste it elsewhere. *Hint:* You'll need to keep references to all the new `JTextField` objects generated so that you can gather author names from them.