Chapter: 11

**Applet, Event, Swing and JavaFx**

Applet, Delegation event, Swing components, JavaFx with application and miscellaneous topics.

**11.0 AWT, Swing and JavaFx**

Although *console-based* programs are excellent for learning Java and for server-side code, most real-world applications will be GUI-based.

* Java’s original GUI framework was the AWT. It was soon followed by Swing, which offered a far superior approach.
* Swing defines a *collection of* ***classes*** *and* ***interfaces*** that support a rich *set of visual components*, such as *buttons*, *text fields*, *scroll* *panes*, *check* *boxes*, *trees*, and *tables*, to name a few. Which are used to construct graphical interfaces.
* JavaFX is Java’s next-generation GUI framework: to better handle the demands of the *modern GUI and advances in GUI design*. JavaFX ***designed as a replacement for*** Swing***.***

[The original JavaFX was based on a scripting language called JavaFX Script. However, JavaFX Script has been discontinued. Beginning with the release of JavaFX 2.0, JavaFX has been programmed in Java itself and provides a comprehensive API. JavaFX 8 represents the latest version of JavaFX, it is the version of JavaFX discussed here. Furthermore, when the term JavaFX is used, it refers to JavaFX 8.]

**11.1 Applet fundamentals**

Applets are *small programs* that are designed for transmission over the Internet and run within a browser. Applets offer a reasonably secure way to ***dynamically*** download ***and*** execute ***programs over the*** Web.

* AWT and Swing applets: Both the Abstract Window Toolkit (AWT) and Swing support the creation of a graphical user interface (GUI). The AWT is the ***original GUI toolkit*** and Swing is a *lightweight* alternative.
* Swing-based *applets* are built upon the same basic architecture as AWT-based *applets*. Also, Swing is built on top of the AWT.
* A simple AWT-based applet step by step: following applet displays the string "Java makes applets easy." inside a window.

**import java.awt.\*;**

**import java.applet.\*;**

**public class** SimpleApplet **extends** **Applet** {

**public void paint**(**Graphics** g) { g.**drawString**("*Java makes applets easy.*", 20, 20); }

}

* ***import java.awt.\*;*** imports the AWT classes. AWT-based applets interact with the user through the AWT, not through the *console-based I/O classes*. The AWT contains support for a ***limited window-based, GUI***.
* ***import java.applet.\*;*** imports the *applet package*. This package contains the class Applet. Every AWT-based applet must be a subclass of ***Applet***.
* The class ***SimpleApplet*** must be declared as ***public*** because it will be ***accessed by outside code***.
* Inside ***SimpleApplet***, ***paint()*** is declared. This method is defined by the AWT Component class *(which is a superclass of Applet)* and is ***overridden by the applet***. ***paint()*** is called each time the applet must *redisplay its output*.

[It can occur for several reasons. Eg: The window in which the applet is running can be overwritten by another window and then uncovered. Or the applet window can be minimized and then restored. Or when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, *paint()* is called. ]

* ***paint()*** has one parameter of type ***Graphics***. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
* Inside ***paint()***, there is a call to ***drawString()***, which is a member of the Graphics class. This method ***outputs a string beginning at the specified X,Y location***. It's general form: ***void drawString(String message, int x, int y)***
* Here, ***message*** is the *string to be output* beginning at ***x,y***. In a Java window, the *upper-left corner* is location ***0,0***. The call to ***drawString()*** in the applet causes the message to be displayed beginning at location ***20,20***.
* Notice that the applet does not have a ***main()*** method. Applets do not begin execution at ***main()***. An *applet begins execution* when the *name of its class* is passed to a *browser* or other *applet-enabled program*.

|  |  |
| --- | --- |
| * Let’s review an applet’s key points: * All AWT-based applets are *subclasses* of Applet. * Applets do not need a ***main()*** method. | * Applets run under an *applet viewer* or a *Java-compatible browser*. * User I/O is not accomplished with Java’s stream I/O classes. Instead, applets use the interface provided by a GUI framework. |

* Running a simple applet: An applet can be run in two ways: inside a ***browser*** or ***appletviewer*** the tool provided with JDK.
* appletviewer is a *special development tool* that displays applets, and it is much easier to use during development.
* One way to execute an applet *(in either a Web browser or the appletviewer)* is to write a short HTML text file that contains an APPLET tag that loads the applet. (The OBJECT tag can also be used, and other deployment strategies are available):

<**applet** **code**="SimpleApplet" **width**=200 **height**=60> </**applet**>

* The ***width*** and ***height*** statements specify the dimensions of the display area used by the applet.
* To execute ***SimpleApplet*** with an applet viewer, for example, if the preceding HTML file is called StartApp.html, then the command line will run ***SimpleApplet***: **C:\>appletviewer StartApp.html**
* Another easier way: Simply include a comment near the top of your applet’s source code file that contains the ***APPLET tag***. For example, the ***SimpleApplet*** source file looks like:

**import java.awt.\*;**

**import java.applet.\*;**

/\* <**applet code**="SimpleApplet" **width**=200 **height**=60> </**apple**t> \*/

**public class** SimpleApplet **extends** **Applet** { **public void paint**(**Graphics** g) { g.**drawString**("*Java makes applets easy.*", 20, 20); } }

* Compile the ***.java*** file. Execute the applet by *passing the name of its source file* to appletviewer. Use command line:

***C:>appletviewer SimpleApplet.java***

[keep in mind that it provides the window frame. Applets run in a browser will not have a visible frame.]

NOTE

Java applets must be signed to prevent security warnings when run in a browser. In fact, in some cases, the applet may be prevented from running.

Adjust the security settings in the Java Control Panel to run a local applet in a browser. Furthermore, unsigned local applets may be blocked from execution in the future.

The concepts and techniques required to sign applets (and other types of Java programs) are beyond the scope of this CODEX. Information is found on Oracle’s website.

**11.2 How an Applet works and applet Skeleton**

* Applets are event driven, and an applet resembles a set of interrupt service routines. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the system. For the most part, your applet should not enter a “mode” of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution.
* It is the user who initiates interaction with an applet—not the other way around. In a console-based program, when the program needs input, it will prompt the user and then call some input method. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks a mouse inside the applet’s window, a mouse-clicked event is generated. If the user presses a key while the applet’s window has input focus, a keypress event is generated. Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.
* A Complete Applet Skeleton: Most applets override a set of methods that provide the basic mechanism by which the *browser or applet viewer interfaces to the applet* and controls its execution. These life-cycle methods are ***init()***, ***start()***, ***stop()***, and ***destroy()***, and they are defined by ***Applet***.
* A fifth method, ***paint()***, is commonly overridden by AWT-based applets even though it is not a life-cycle method. It is inherited from the ***AWT Component*** class. Overriding ***paint()*** applies mostly to AWT-based applets. Swing applets use a ***different painting mechanism***. These four life-cycle methods plus ***paint()*** can be assembled into the skeleton :

[Since default implementations for all of these methods are provided, applets do not need to override those methods they do not use. ]

***import java.awt.\*;***

***import java.applet.\*;***

/\* <applet code="AppletSkel" width=300 height=100> </applet> \*/

**public class** AppletSkel **extends** **Applet** {

**public void init**() { */\* initialization : Called first. \*/* }

**public void start**() { */\* start or resume execution : Called second, after init(). Also called whenever the applet is restarted \*/* }

**public void stop**() { */\* suspends execution : Called when the applet is stopped.\*/*  }

**public void destroy**() { */\* perform shutdown activities : Called when applet is terminated. This is the last method executed \*/*  }

**public void paint**(**Graphics** g) { */\* redisplay contents of window : Called when an AWT-based applet's window must be restored \*/* }

} */\*Although this skeleton does not do anything, it can be* ***compiled*** *and* ***run****.\*/*

* When an applet begins, following methods are called in this sequence:
* ***init()*** : It is called first. In ***init()*** your applet will *initialize variables* and perform any other startup activities
* ***start()*** : It is called after ***init()***. It also calls to *restart an applet* after it has been *stopped*, [Eg: user returns to a previously displayed web page that contains an applet]. Thus, ***start()*** might be called more than once during the *life cycle* of an applet.
* ***paint()*** : It is called each time an AWT-based applet’s output must be *redrawn*.
* When an *applet* is *terminated*, the following sequence of method calls takes place:
* ***stop()*** : When the page containing applet is left, the ***stop()*** method is called. ***stop()*** is used to *suspend any child threads* created by the applet and to perform any other activities required to put the ***applet in a safe, idle state***.
* A call to ***stop()*** does not mean that the applet should be terminated because it might be restarted with a call to ***start()*** if the user returns to the page.
* ***destroy()*** : It is called when the applet is no longer needed. It is used to perform any shutdown operations for applet.

**11.3 repaint(), update() and getGraphics( )**

An AWT-based applet writes to its window only when its ***paint()*** is called by the run-time system. Using ***repaint()***, applet updates its window itself when its information changes. Eg: moving banner. [Since applet must quickly return control to the Java run-time system. It cannot create a loop inside paint( ) that repeatedly scrolls the banner. This would prevent control from passing back to the run-time system. ]

* repaint(): The ***repaint()*** is defined by the AWT’s ***Component*** class. It causes the run-time system to execute a call to applet’s ***paint()***. [Thus, for another part of your applet to output to its window, simply store the output and then call repaint(). This causes a call to paint(), which can display the stored information.]
* The simplest version of ***repaint()*** is: ***void repaint()*** This version causes the ***entire*** *window to be* ***repainted***.
* This version specifies a *region* that will be *repainted:* ***void repaint(int left, int top, int width, int height)***
* Here, the coordinates of the *upper-left corner* of the region are specified by ***left*** and ***top***, and the *width* and *height* of the region are passed in ***width*** and ***height***. These dimensions are specified in pixels.

[If you only need to update a small portion of the window, it is more efficient to repaint only that region.]

* update(): ***update()*** is defined by the Component class, and it is called when applet has requested that a *portion of its window* be redrawn. The default version of ***update()*** simply calls ***paint()***. [However, update( ) can be overriden so that it performs more subtle repainting.]
* getGraphics(): To output to an applet’s window, you must obtain a *graphics context* by calling ***getGraphics()*** *(defined by* ***Component****)* and then use this *context* to output to the window. However, for most AWT-based applications, it is better and easier to route window output through ***paint()*** and to call ***repaint()*** when the ***contents of the window change***.
* Example 1: A simple banner applet that use ***repaint()***. This applet creates a thread that scrolls the message contained in ***msg*** right to left across the *applet's window*.

|  |  |
| --- | --- |
| ***import java.awt.\*;***  ***import java.applet.\*;***  /\* <**applet** code="Banner" width=300 height=50> </**applet**> \*/  **public class** Banner **extends** **Applet implements Runnable** {  **String** msg = " Java Rules the Web ";  **Thread** t;  **boolean** stopFlag;  **public void** ***init***() { t = **null**; } *// Initialize t to null.*  **public void** ***start***() { t = **new** **Thread**(**this**); *// Start thread*  stopFlag = **false**;  t.**start**(); } | **public** **void** **run**() { *// Entry point for the thread that runs the banner.*  **for**( ; ; ){ **try** { **repaint**(); *// Redisplay banner*  **Thread.sleep**(250);  **if**(stopFlag)  **break**; }  **catch**(***InterruptedException*** exc){} } }  **public** **void** ***stop***() { stopFlag = **true**; *// Pause the banner.*  t = null; }  **public void** ***paint***(**Graphics** g) { **char** ch;  ch = msg.**charAt**(0);  msg = msg.**substring**(1, msg.**length**());  msg += ch;  g.**drawString**(msg, 50, 30); }} |

* Since the *scrolling of the message is a repetitive task*, it is performed by a separate thread, created by the applet when it is initialized.
* Banner extends Applet, but it also implements Runnable. Since the applet will be creating a second thread of execution that will be used to scroll the banner.
* The scroll message is contained in msg. A *reference to the thread* that runs the applet is stored in t. The Boolean variable stopFlag is used to stop the applet. Inside ***init()***, the thread reference variable t is set to null.
* The *run-time system* calls ***start()*** to start the applet running. Inside ***start()***, a *new* *thread of execution* is created and assigned to the *Thread variable* t. Then, stopFlag is set to false. Next, the thread is started by a call to ***t.start()***. Here ***t.start()*** calls a method defined by Thread, which causes ***run()*** to begin executing. It does not cause a call to the version of ***start()*** defined by Applet. These are two separate methods.
* In ***run()***, a call to ***repaint()*** is made. This eventually causes the ***paint(*** ***)*** to be called, and the rotated contents of msg are displayed. Between each iteration, ***run()*** sleeps for a quarter of a second. The net effect of ***run()*** is that the contents of msg are scrolled right to left in a constantly moving display. The stopFlag variable is checked on each iteration. When it is true, the ***run()*** method terminates.
* When a new page is viewed during scrolling, ***stop()***is called, which sets stopFlag to true, causing ***run()*** to terminate. It also sets t to null. Thus, there is no longer a reference to the Thread object, and it can be recycled the next time the garbage collector runs. *This mechanism used to stop the thread when its page is no longer in view*. When the *applet is brought back into view*, ***start()*** is once again called, which starts a new thread to execute the banner. Inside ***paint()***, the message is rotated and then displayed.

**11.4 Using the Status Window**

To output a message to the *status window of the browser* or *applet viewer*, call ***showStatus()***, defined by Applet, with the string that you want displayed. The general form: **void showStatus(String msg)**  Here, ***msg*** is the string to be displayed.

* The status window displays current state of applet, suggest options, or possibly report some types of errors. It is also an excellent debugging aid.
* Example 2: The following applet demonstrates showStatus( ):

***import java.awt.\*; import java.applet.\*;*** */\* <applet code="StatusWindow" width=300 height=50> </applet> \*/*

**public class** StatusWindow **extends** **Applet**{ **public void paint**(**Graphics** g) { g.**drawString**("*This is in the applet window*.", 10, 20);

*/\* Display msg in applet window. \*/*  ***showStatus***("*This is shown in the status window*."); }}

**11.5 Passing Parameters to Applets**

* To *pass* parameters to an applet, use the *PARAM* attribute of the *APPLET* *tag*, specifying the parameter’s name and value.
* To *retrieve* a parameter, use ***getParameter( )***, defined by Applet. General form: ***String getParameter(String paramName)***
* Here, ***paramName*** is the *name of the parameter*. It returns the value of the specified parameter in the form of a ***String*** object.
* Thus, for *numeric* and *boolean* values, you will need to convert their *string representations* into their *internal* *formats*.
* If the specified parameter not found, ***null*** is returned. i.e. confirm that the value returned by ***getParameter( )*** is valid.
* Also, check any parameter that is converted into a numeric value, confirming that a valid conversion took place.
* Example 3: Following demonstrates passing parameters:

|  |  |
| --- | --- |
| ***import java.awt.\*; import java.applet.\*;***  /\* <**applet** code="Param" **width**=300 **height**=80>  <**param** **name**=author value="Herb Schildt">  <**param** **name**=purpose value="Demonstrate Parameters">  <**param** **name**=version value=2>  </**applet**> \*/ */\* HTML parameters are passed to the applet\*/*  **public class** Param **extends** **Applet** { **String** author;  **String** purpose;  **int** ver; | **public void start**() { **String** temp; */\* check that the parameter exists\*/*  author = **getParameter**("author"); **if**(author == **null**) author = "not found";  purpose = **getParameter**("purpose"); **if**(purpose == **null**) purpose = "not found";  temp = **getParameter**("version");  **try** { **if**(temp != **null**) ver = **Integer**.**parseInt**(temp);  **else** ver = 0; } */\* make sure that numeric conversions succeed\*/*  **catch**(***NumberFormatException*** exc) { ver = -1; */\* error code \*/* } }  **public** **void** **paint**(**Graphics** g){g.**drawString**("Purpose: " + purpose, 10, 20);  g.**drawString**("By: " + author, 10, 40);  g.**drawString**("Version: " + ver, 10, 60); }} |

**11.6 The Applet Class**

All AWT-based applets are subclasses of the Applet class. Applet inherits the following *superclasses* defined by the AWT: ***Component***, ***Container***, and ***Panel***. Applet contains several methods that give us detailed control over the execution of an applet. *All of the methods* defined by Applet are shown in following Table.

|  |  |  |
| --- | --- | --- |
| The Methods Defined by Applet | | |
| Method | | Description |
| ***void destroy( )*** | | Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction. |
| ***AccessibleContext getAccessibleContext( )*** | | Returns the accessibility context for the invoking object. |
| ***AppletContext getAppletContext( )*** | | Returns the context associated with the applet. |
| ***String getAppletInfo( )*** | | Returns a string that describes the applet. |
| ***AudioClip getAudioClip(URL url)*** | | Returns an AudioClip object that encapsulates the audio clip found at the location specified by url. |
| ***AudioClip getAudioClip(URL url, String clipName)*** | | Returns an AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName. |
| ***URL getCodeBase( )*** | | Returns the URL associated with the invoking applet. |
| ***URL getDocumentBase( )*** | | Returns the URL of the HTML document that invokes the applet. |
| ***Image getImage(URL url)*** | | Returns an Image object that encapsulates the image found at the location specified by url. |
| ***Image getImage(URL url, String imageName)*** | | Returns an Image object that encapsulates the image found at the location specified by url and having the name specified by imageName. |
| ***Locale getLocale( )*** | | Returns a Locale object that is used by various localesensitive classes and methods. |
| ***String getParameter(String paramName)*** | | Returns the parameter associated with paramName. Null is returned if the specified parameter is not found. |
| ***String[ ] [ ] getParameterInfo( )*** | | Overrides of this method should return a String table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose. The default implementation returns null. |
| ***void init( )*** | | This method is called when an applet begins execution. It is the first method called for any applet. |
| ***boolean isActive( )*** | | Returns true if the applet has been started. It returns false if the applet has been stopped. |
| ***boolean isValidateRoot( )*** | | Returns true, which indicates that an applet is a validate root. |
| ***static final AudioClip newAudioClip(URL url)*** | | Returns an AudioClip object that encapsulates the audio clip found at the location specified by url. This method is similar to getAudioClip( ) except that it is static and can be executed without the need for an Applet object. |
| ***void play(URL url)*** | | If an audio clip is found at the location specified by url, the clip is played. |
| ***void play(URL url, String clipName)*** | | If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played. |
| ***void resize(Dimension dim)*** | | Resizes the applet according to the dimensions specified by dim. Dimension is a class stored inside java.awt. It contains two integer fields: width and height. |
| ***void resize(int width, int height)*** | | Resizes the applet according to the dimensions specified by width and height. |
| ***final void setStub(AppletStub stubObj)*** | | Makes stubObj the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A stub is a small piece of code that provides the linkage between your applet and the browser. |
| ***void showStatus(String str)*** | | Displays str in the status window of the browser or applet viewer. If the browser does *not support* a status window, then no action takes place |
| ***void start( )*** | Called by the browser when an applet should start (or resume) execution. It is automatically called after init( ) when an applet first begins. | |
| ***void stop( )*** | Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls start( ). | |

**11.7 Event Handling**

GUI programs, such as applets, are event driven. Most events to which your program will respond are generated by the user. These events are passed to your program in a variety of ways, with the specific method depending upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various controls, such as a push button. AWT-based events are supported by the java.awt.event *package*.

* The Delegation Event Model: A source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

[The logic that processes events is cleanly separated from the UI logic that generates those events. A UI element is able to “delegate” the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification.]

* Events: In the delegation model, an *event* is an object that describes a *state change* in a *source*. An event can be generated in a GUI by pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
* Event Sources: An *event source* is an *object* that *generates an event*.
* Event-Registration Method: A *source* must register *listeners* in order for the *listener* to receive notifications about a *specific type* of *event*. *Each type of* ***event*** *has its own* ***registration*** *method*. Example: the method that registers *a keyboard event listener* is called ***addKeyListener()***. The method that registers a *mouse motion listener* is called ***addMouseMotionListener()***. Here is the general form:

***public void addTypeListener(TypeListener el)***

* Here, ***Type*** is the *name of the event*, and ***el*** is a *reference to the event listener*.
* When an ***event occurs***, all ***registered listeners*** are notified and receive a copy of the ***event object***.
* Event-Unregistration Method: A *source* must also provide a *method* that allows a listener to unregister an *interest* in a specific type of *event*. The general form of the method:

***public void removeTypeListener(TypeListener el )***

* Here, ***Type*** is the name of the event, and ***el*** is a reference to the event listener.
* Eg: To remove a *keyboard listener*, you would call ***removeKeyListener()***. The methods that *add/remove listeners* are provided by the *source that generates events*. [***Component*** provides methods to add-remove *keyboard/mouse* event listeners.]
* Event Listeners: A listener is an object that is notified when an ***event occurs***. It has two major requirements.
* First, it must have been *registered* with one or more ***sources*** to receive notifications about *specific types of events*.
* Second, it must *implement* ***methods*** to *receive and process these notifications*.
* The methods that receive and process AWT events are defined in a *set of interfaces*, such as those found in ***java.awt.event***. Example, the MouseMotionListener interface defines methods that receive notifications when the mouse is dragged or moved. Any object that provides an *implementation* of this *interface* can *receive and process these events*.
* Event Classes: The classes that represent events are at the *core of Java’s event handling mechanism.*
* At the root of the Java event class hierarchy is ***EventObject***, which is in ***java.util***. It is the superclass for all events.
* The class ***AWTEvent***, defined within the ***java.awt*** package, is a *subclass* of ***EventObject***. It is the *superclass* (either directly or indirectly) for all *AWT-based events* used by the *delegation event model*.
* The package ***java.awt.event*** defines several types of events that are generated by various UI elements. Following Table enumerates several commonly used UI elements and provides a brief description of when they are generated.

|  |  |
| --- | --- |
| ***Commonly Used Event Classes in* java.awt.event** | |
| Event Class | Description |
| ***ActionEvent*** | Generated when a button is pressed, a list item is double-clicked, or a menu item is selected. |
| ***AdjustmentEvent*** | Generated when a scroll bar is manipulated. |
| ***ComponentEvent*** | Generated when a component is hidden, moved, resized, or becomes visible. |
| ***ContainerEvent*** | Generated when a component is added to or removed from a container. |
| ***FocusEvent*** | Generated when a component gains or loses keyboard focus. |
| ***InputEvent*** | Abstract superclass for all component input event classes. |
| ***ItemEvent*** | Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected. |
| ***KeyEvent*** | Generated when input is received from the keyboard. |
| ***MouseEvent*** | Generated when the mouse is dragged or moved, clicked, pressed, or released; also generated when the *mouse enters or exits a component*. |
| ***MouseWheelEvent*** | Generated when the mouse wheel is moved. |
| ***TextEvent*** | Generated when the value of a text area or text field is changed. |
| ***WindowEvent*** | Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

* Event Listener Interfaces: Event listeners receive *event notifications*. Listeners for AWT-based events are created by *implementing one or more of the* ***interfaces*** defined by the **java.awt.event** *package*. When an event occurs, the ***event source*** invokes the ***appropriate method*** defined by the ***listener*** and provides an *event object* as its *argument*.

|  |  |
| --- | --- |
| ***Commonly Used Event Listener Interfaces*** | |
| Interface | Description |
| ***ActionListener*** | Defines one method to receive action events. Action events are generated by such things as push buttons and menus. |
| ***AdjustmentListener*** | Defines one method to receive adjustment events, such as those produced by a scroll bar. |
| ***ComponentListener*** | Defines four methods to recognize when a component is hidden, moved, resized, or shown. |
| ***ContainerListener*** | Defines two methods to recognize when a component is added to or removed from a container. |
| ***FocusListener*** | Defines two methods to recognize when a component gains or loses keyboard focus. |
| ***ItemListener*** | Defines one method to recognize when the state of an item changes. An item event is generated by a check box, for example. |
| ***KeyListener*** | Defines three methods to recognize when a key is pressed, released, or typed. |
| ***MouseListener*** | Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released. |
| ***MouseMotionListener*** | Defines two methods to recognize when the mouse is dragged or moved. |
| ***MouseWheelListener*** | Defines one method to recognize when the mouse wheel is moved. |
| ***TextListener*** | Defines one method to recognize when a text value changes. |
| ***WindowListener*** | Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**11.8 Using the Delegation Event Model**

1. Implement the appropriate ***interface*** in the ***listener*** so that it will receive the ***type of event*** desired.
2. Implement ***code*** to ***register*** and ***unregister*** (if necessary) the listener as a ***recipient*** for the ***event notifications***.

* A source may generate ***several types of events***. Each event must be registered separately.
* An object may *register* to *receive* several types of events, but it must ***implement all of the interfaces*** that are required to receive these events.
* Handling Mouse and Mouse Motion Events: To handle ***mouse*** and ***mouse motion*** events, implement the ***MouseListener*** and the ***MouseMotionListener*** interfaces.

|  |  |  |
| --- | --- | --- |
| The ***MouseListener*** interface defines five methods: | The general forms of these methods are: | |
| If a mouse button is clicked, *mouseClicked( )* is invoked. | **void mouseClicked(MouseEvent me)** | |
| When the mouse enters a component, the *mouseEntered( )* method is called. | **void mouseEntered(MouseEvent me)** | |
| When mouse leaves, *mouseExited( )* is called. | **void mouseExited(MouseEvent me)** | |
| The *mousePressed( )* and *mouseReleased( )* methods are invoked when a mouse button is pressed and released, respectively. | **void mousePressed(MouseEvent me)**  **void mouseReleased(MouseEvent me)** | |
| The ***MouseMotionListener*** interface defines two methods | |  |
| The ***mouseDragged( )*** method is called multiple times as the mouse is ***dragged***. | **void mouseDragged(MouseEvent me)** | |
| The ***mouseMoved( )*** method is called multiple times as the mouse is ***moved***. | **void mouseMoved(MouseEvent me)** | |

* The ***MouseEvent*** object passed in me describes the event. ***MouseEvent*** defines a number of methods that you can use to get information about what happened.

|  |  |  |
| --- | --- | --- |
| * The most commonly used methods in ***MouseEvent*** are ***getX()*** and ***getY()***. These return the ***X*** and ***Y*** *coordinates* *of the mouse* (relative to the window) when the event occurred. Their forms are: | | ***int getX()***  ***int getY()*** |
| * ***Screen-relative coordinates of the mouse (that is, absolute) location:*** ***MouseEvent*** defines methods that obtain the X and Y coordinates of the mouse relative to the screen. | ***int getXOnScreen()***  ***int getYOnScreen()*** | |

* Example 4: Following example will show how to handle the basic mouse and mouse motion events.
* It displays the current coordinates of the mouse in the applet’s status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked." is displayed in the upper-left corner of the applet display area.
* As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a \* is shown, which tracks with the mouse pointer as it is dragged.
* Notice that the two variables, mouseX and mouseY, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by paint( ) to display output at the point of these occurrences.

*import* ***java.awt.\*****; import* ***java.awt.event.\*****; import* ***java.applet.\*****;* */\* <applet code="MouseEvents" width=300 height=100> </applet> \*/*

**public class** MouseEvents **extends Applet implements MouseListener, MouseMotionListener** { **String** msg = "";

*/\* coordinates of mouse \*/* **int** mouseX = 0, mouseY = 0;

**public** **void init()** { **addMouseListener**(this); **addMouseMotionListener**(this); } *// Register this class as a listener for mouse events.*

**public void** ***mouseClicked***(**MouseEvent** me) { mouseX = 0; mouseY = 10; msg = "Mouse clicked."; **repaint**(); } *//Handle mouse click*

*// This, and the other event handlers, respond to mouse events.*

**public void** ***mouseEntered***(**MouseEvent** me) { mouseX = 0; mouseY = 10; msg = "Mouse entered."; **repaint**(); } *// Handle mouse entered.*

**public void** ***mouseExited***(**MouseEvent** me) { mouseX = 0; mouseY = 10; msg = "Mouse exited."; **repaint**(); } *// Handle mouse exited.*

**public void** ***mousePressed***(**MouseEvent** me) { mouseX = me.**getX**(); mouseY = me.**getY**(); *// save coordinates*

msg = "Down"; repaint(); } *// Handle button pressed.*

**public void** ***mouseReleased***(**MouseEvent** me) { mouseX = me.**getX**(); mouseY = me.**getY**(); *// save coordinates*

msg = "Up"; repaint(); } *// Handle button released.*

**public void** ***mouseDragged***(**MouseEvent** me) { mouseX = me.**getX**(); mouseY = me.**getY**(); *// save coordinates*

msg = "\*";

**showStatus**("Dragging mouse at " + mouseX + ", " + mouseY);

**repaint**(); } *// Handle mouse dragged.*

**public void *mouseMoved***(**MouseEvent** me) { **showStatus**("Moving mouse at " + me.**getX**() + ", " + me.**getY**()); } *// Handle mouse moved.*

**public void** ***paint***(***Graphics*** g) { g.**drawString**(msg, mouseX, mouseY); } *// Display msg in applet window at current X,Y location.*

}

* The MouseEvents class extends Applet and implements both the MouseListener and MouseMotionListener ***interfaces***.
* Notice that the applet is both the source and the listener for these events. This works because Component, which supplies the addMouseListener() and addMouseMotionListener( ) methods, is a superclass of Applet. *Being both the source and the listener for events is a common situation for applets.*
* The applet then implements all the methods defined by the MouseListener and MouseMotionListener interfaces. Each method handles its event and then returns.
* Inside ***init()***, the applet registers itself as a ***listener*** for mouse events. This is done by using ***addMouseListener()*** and ***addMouseMotionListener()***, which are members of ***Component***. They are shown here:

***void addMouseListener(MouseListener ml)***

***void addMouseMotionListener(MouseMotionListener mml)***

* Here, ml is a *reference* to the object *receiving* ***mouse events***, and
* mml is a *reference* to the object *receiving* ***mouse motion events***. In this program, the same object is used for both.

**11.9 More Java Keywords and more on "this"**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. ***transient*** | 1. ***volatile*** | 1. ***instanceof*** | 1. ***native*** | 1. ***strictfp*** | 1. ***assert*** |

* transient and volatile Modifiers:

transient: When an instance variable is declared as transient, then its value need not persist when an object is stored. Thus, a transient field is one that does not affect the persisted state of an object.

Volatile: volatile tells the compiler that a variable can be changed unexpectedly by other parts of program. One of these situations involves multithreaded programs: sometimes two or more threads will share the same variable. For efficiency, each thread can keep its own, private copy of such a shared variable, possibly in a register of the CPU. The real (or master) copy of the variable is updated at various times, such as when a synchronized method is entered. However sometimes it is inappropriate. In some cases, the master copy of a variable always reflects the current state, and that this current state is used by all threads. For this, declare the variable as volatile.

* Instanceof operator: Sometimes it is useful to know the type of an object during run time. For example, you might have one thread of execution that generates various types of objects and another thread that processes these objects.

Another situation is to obtain an object’s type at run time which involves casting. In Java, an invalid cast causes a run-time error. Most of them can be caught at compile time. But, casts involving class hierarchies can produce invalid casts that can only be detected at run time [Because a superclass reference can refer to subclass objects]. The instanceof keyword addresses these types of situations. The instanceof operator has this general form: ***objref instanceof type***

* Here, objref is a *reference* to an instance of a class, and type is a class or interface type.
* If *object referred* to by objref *is of* the specified type or can be cast into the specified type, then instanceof evaluates to true. Otherwise, is false.
* strictfp: In old version of Java the floating-point computation model was relaxed slightly. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. This prevents overflow or underflow in some cases.
* By modifying a class, method, or interface with ***strictfp***, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. When a class is modified by strictfp, all of the methods in the class are also strictfp *automatically*.
* assert: The assert keyword is used in assertion, which is a condition that is expected to be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an assert statement. At run time, if the condition actually is true, no other action takes place. However, if the condition is false, then an AssertionError is thrown. assert keyword has two forms.

|  |  |
| --- | --- |
| ***assert condition;*** | condition is a Boolean expression. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default AssertionError object is thrown. Example: assert n > 0; If n is less than or equal to zero, then an AssertionError is thrown. Otherwise, no action takes place. |
| ***assert condition : expr;*** | In this version, expr is a value that is passed to the AssertionError constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for expr, but any *non-void expression* is allowed as long as it defines a reasonable string conversion. To *enable assertion checking at run time*, you must specify the -ea option. For example, to enable assertions for Sample, execute it using this line: ***java -ea Sample*** |

* Native Methods: To call a subroutine that is written in a language other than Java [Typically, such a subroutine will exist as executable code for the CPU and environment in which you are working—that is, native code. For example, you may wish to call a native code subroutine in order to achieve faster execution time.] use the native keyword, which is used to declare native code methods. These methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the native modifier, but *do not define any body* for the method. For example: public native int meth() ; Once you have declared a native method, you must provide the native method and follow a rather *complex series of steps in order to link it with your Java code*.

* this with parentheses **this(x);** : This form of ***this*** enables ***one constructor to invoke another constructor*** within the same class. The general form of this use of ***this*** : ***this(arg-list)***
* When ***this()*** is executed, the overloaded constructor that matches the *parameter list* specified by ***arg-list*** is *executed first*. Then, *if there are any* ***statements*** *inside the* ***original constructor****, they are* ***executed***. The call to ***this()*** must be the *first statement* within the constructor.

**class** MyClass { **int** a, b;

MyClass(**int** i, **int** j) { a = i; b = j; } *// Initialize a and b individually.*

MyClass(**int** i) { **this**(i, i); */\* invokes MyClass(i, i)\*/* } *// Use this() to initialize a and b to the same value.*

}

* In MyClass, only the first constructor actually assigns a value to a and b. The second constructor simply invokes the first. Therefore, when this statement executes: ***MyClass mc = new MyClass(8);*** the call to MyClass(8) causes this(8, 8) to be executed, which translates into a call to MyClass(8, 8).
* Restriction 1: ***this()*** can be useful because it can prevent the *unnecessary duplication of code*. However, constructors that call ***this()*** will execute a *bit slower* than those that contain all of their *initialization code in-line*.
* Restriction 2: you cannot use any instance variable of the constructor’s class in a call to ***this()***.
* Restriction 3: you cannot use ***super()*** and ***this()*** in the same constructor because each must be the first statement in the constructor.

**11.10 Swing Intro**

* In computer networking, ***peer*** refers to another computer on a network that has the same or similar rights as another computer.

Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). With very few exceptions, Swing components are lightweight. This means that a component is written entirely in Java. They do not rely on platform-specific peers. It is possible to separate the look and feel of a component from the logic of the component and it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component. Java provides look-and-feels, such as metal (Java look and feel) and Nimbus, that are available to all *Swing users*.

* *Swing’s pluggable look and feel* is made possible because Swing uses a modified version of the classic model-view-controller (MVC) architecture. In MVC terminology,
* The model corresponds to the *state information* associated with the *component*. For example, in the case of a *check box*, the model contains a field that indicates if the box is *checked* or *unchecked*.
* The view determines how the *component* is *displayed on the screen*, including any aspects of the view that are *affected by the current state of the* model.
* The controller determines how the *component reacts* to the *user*. For example, when the user clicks a check box, the controller reacts by *changing the model* to reflect the *user’s choice (checked or unchecked)*. This then results in the *view* being *updated*.
* By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two.
* Swing uses a *modified version* of MVC that combines the view and the controller into a *single logical entity* called the UI delegate. For this reason, Swing’s approach is called either the model-delegate architecture or the separable model architecture. Therefore, although Swing’s component architecture is based on MVC, it does not use a *classical implementation* of it.
* Components and Containers: A component is an independent visual control, such as a push button or text field. In order for a component to be displayed, it must be held within a container
* A *container* holds a *group of components*. Thus, a *container* is a special type of *component* that is designed to hold *other components*. A container can also hold other containers. All Swing GUIs will have at least one container.
* Swing can define a *containment hierarchy*, at the top of which must be a *top-level container*.

**11.10.1 Components**

* Swing components are derived from the JComponent class (Except the four top-level containers). JComponent inherits the AWT classes Container and Component.
* All of Swing’s components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components (including those used as containers): *Notice that all component classes begin with the letter* J.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***JApplet*** | ***JDesktopPane*** | ***JLabel*** | ***JOptionPane*** | ***JRootPane*** | ***JTabbedPane*** | ***JToolTip*** |
| ***JButton*** | ***JDialog*** | ***JLayer*** | ***JPanel*** | ***JScrollBar*** | ***JTable*** | ***JTree*** |
| ***JCheckBox*** | ***JEditorPane*** | ***JLayeredPane*** | ***JPasswordField*** | ***JScrollPane*** | ***JTextArea*** | ***JViewport*** |
| ***JCheckBoxMenuItem*** | ***JFileChooser*** | ***JList*** | ***JPopupMenu*** | ***JSeparator*** | ***JTextField*** | ***JWindow*** |
| ***JColorChooser*** | ***JFormattedTextField*** | ***JMenu*** | ***JProgressBar*** | ***JSlider*** | ***JTextPane*** |  |
| ***JComboBox*** | ***JFrame*** | ***JMenuBar*** | ***JRadioButton*** | ***JSpinner*** | ***JTogglebutton*** |  |
| ***JComponent*** | ***JInternalFrame*** | ***JMenuItem*** | ***JRadioButtonMenuItem*** | ***JSplitPane*** | ***JToolBar*** |  |

**11.10.2 Containers: Swing defines two types of containers.**

* top-level or Heavy-weight containers: *JFrame*, *JApplet*, *JWindow*, and *JDialog*. These containers do not inherit *JComponent*. They inherit the *AWT* classes Component and Container. *Top-level containers are heavyweight*.

|  |  |
| --- | --- |
| * A top-level container must be at the top of a containment hierarchy. * A top-level container is not contained within any other container. | * Every containment hierarchy must begin with a top-level container. * JFrame is used for applications, JApplet is used for applets is. |

* The lightweight container: *Lightweight containers* inherit JComponent. Eg: JPanel, JScrollPane, and JRootPane. Lightweight containers are often used to collectively organize and manage *groups of related components* because a lightweight container can be contained within another container.
* Lightweight containers can be used to create subgroups of related controls that are contained within an outer container.
* The Top-Level Container Panes: Each top-level container defines a set of panes. At the top of the hierarchy is an instance of JRootPane.
* JRootPane is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that compose the root pane are called the glass pane, the content pane, and the layered pane.
* The glass pane is the *top-level pane*. The *glass pane* enables you to manage mouse events that affect the *entire container* (rather than an individual control) or to *paint* over any other *component*.
* The layered pane allows components to be given a ***depth value***. This value determines which component ***overlays another***. (Thus, the layered pane lets you specify a ***Z-order*** for a component). The ***layered pane*** holds the ***content pane*** and the (optional) ***menu bar***.
* The *application* will *interact* most with the content pane, because this is the pane to which you will ***add visual components***. In other words, when you ***add*** a ***component***, such as a ***button***, to a ***top-level container***, you will add it to the ***content pane***.
* Layout Managers: The ***layout manager*** controls the position of components within a container. Java offers several layout managers. Most are provided by the AWT (within java.awt), but Swing adds a few of its own.
* All ***layout managers*** are instances of a class that implements the LayoutManager interface. (Some will also implement the LayoutManager2 interface.)

|  |  |  |  |
| --- | --- | --- | --- |
| ***FlowLayout*** | A simple layout that positions components left-to-right, top-to-bottom. (Positions components right-to-left for some cultural settings.) | | |
| ***BorderLayout*** | Positions components within the center or the borders of the container. This is the default layout for a content pane. | | |
| ***GridLayout*** | Lays out components within a grid. | ***BoxLayout*** | Lays out components vertically or horizontally within a box. |
| ***GridBagLayout*** | Lays out different size components within a flexible grid. | ***SpringLayout*** | Lays out components subject to a set of constraints. |

* BorderLayout: ***BorderLayout*** is the ***default layout manager*** for the ***content pane***. It implements a layout style that defines five locations to which a component can be added. The first is the ***center***. The other ***four*** are the ***sides*** (i.e., borders), which are called ***north***, ***south***, ***east***, and ***west***. By default, when you add a component to the ***content pane***, you are ***adding*** the component to the ***center***. To add a component to one of the other regions, ***specify its name***.
* FlowLayout: A ***flow layout*** lays out components ***one row at a time***, top to bottom. When one row is full, layout advances to the next row. Be aware that if you resize the frame, the position of the components will change.

|  |  |
| --- | --- |
| Note: | Swing programs differ from the console-based programs and AWT-based applets. Swing programs use the Swing component set to handle user interaction, but they also have special requirements that relate to threading. |

* Example 5: Swing is typically used in *desktop application* and in *applet building*. Following uses Swing components: JFrame and JLabel to build a swing application
* JFrame is the top-level container that is commonly used for Swing applications. The program uses a JFrame container to hold an *instance* of a JLabel. The label displays a *short text message*.
* JLabel is the Swing component that creates a *label*, *which is a component* that displays information. The label is Swing’s simplest component because it is passive. That is, a ***label does not respond to user input. It just displays output***.

***import javax.swing.\*;*** *// Swing programs must import* ***javax.swing****.*

**class** SwingDemo { SwingDemo() **{**  **JFrame** jfrm = **new** JFrame("A Simple Swing Application"); *// Create a new JFrame container.*

jfrm.**setSize**(275, 100); *// Give the frame an initial size.*

jfrm.**setDefaultCloseOperation**(JFrame.EXIT\_ON\_CLOSE); *// Terminate when the user closes the application.*

**JLabel** jlab = **new** JLabel(" Swing defines the modern Java GUI."); *// Create a text-based label.*

jfrm.**add**(jlab); *// Add the label to the content pane.*

jfrm.**setVisible**(**true**); */\* Display the frame.\*/*  **}**

**public static void main(String args[])** {

**SwingUtilities.invokeLater**( **new** Runnable() { **public** **void** **run**() { **new** SwingDemo(); } }); *// Create the frame on the event dispatching thread.*

}}

* Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, use: **javac SwingDemo.java**
* The First Swing Example Line by Line: The program begins by importing the package: ***import javax.swing.\*;*** This package contains the components and models defined by Swing. It defines classes that implement labels, buttons, edit controls, and menus. This package will be included in all programs that use Swing.
* SwingDemo class is then declared and a constructor for that class. Inside the constructor a JFrame is created, using: ***JFrame jfrm = new JFrame("A Simp…");***
* This creates a container called jfrm that defines a *rectangular window* complete with a *title bar*; *close*, *minimize*, *maximize*, and *restore* buttons; and a *system menu*. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.
* Next, the *window is sized* using: ***jfrm.setSize(275, 100);*** The setSize( ) sets the dimensions of the window, which are specified in pixels. Its general form:

***void setSize(int width, int height)***

* Terminating: By default, when a *top-level window* is closed (user clicks the close box), the window is removed from the screen, but application isn't terminated.
* To terminate the entire application when its top-level window is closed, the easiest way is to call ***setDefaultCloseOperation()***:

***jfrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);***

After this call executes, closing the window causes the entire application to terminate.

* The general form of ***setDefaultCloseOperation()*** is: ***void setDefaultCloseOperation(int what)***
* The value passed in ***what*** determines what happens when the window is closed. There are several options :

|  |  |  |  |
| --- | --- | --- | --- |
| Names reflect their actions. These constants are declared in WindowConstants, which is an interface declared in javax.swing that is implemented by JFrame. | | | |
| ***JFrame.EXIT\_ON\_CLOSE*** | ***JFrame.DISPOSE\_ON\_CLOSE*** | ***JFrame.HIDE\_ON\_CLOSE*** | ***JFrame.DO\_NOTHING\_ON\_CLOSE*** |

* The next line of code creates a JLabel component: ***JLabel jlab = new JLabel(" Swing defines the modern Java GUI.");***
* JLabel does not accept user input. It simply displays information, which can consist of text, an icon, or a combination of the two.
* The line: ***jfrm.add(jlab);*** adds the label to the content pane of the frame. Since,all top-level containers have a content pane in which components are stored. Thus, to add a component to a frame, you must add it to the frame’s content pane. This is accomplished by calling add( ) on the JFrame reference (jfrm in this case). ***add()*** has several versions. The general form of which we used is: ***Component add(Component comp)***
* By default, the content pane associated with a JFrame uses a border layout. This version of add( ) adds the component to the center location. Other versions of add( ) specify one of the border regions. When a component is added to the center, its size is automatically adjusted to fit the size of the center.
* ***jfrm.setVisible(true);*** is the last statement in the SwingDemo constructor causes the window to become visible. The ***setVisible()*** method has this general form: ***void setVisible(boolean flag)***
* If flag is true, the window will be displayed. Otherwise, hidden. By default, a JFrame is invisible, so setVisible(true) must be called to show it.
* Inside ***main()***, by creating SwingDemo object, window and the label is displayed. Notice, SwingDemo constructor is invoked using:

***SwingUtilities.invokeLater(new Runnable() { public void run() { new SwingDemo(); } });***

* It causes a SwingDemo object to be created on the event-dispatching thread rather than on the main thread. Because in general, Swing programs are event-driven ***(event is generated by user interaction with component).*** *event* is passed to *application* by calling an event handler defined by the application.
* Event handler is executed on the event-dispatching thread provided by Swing . So, two different threads (main thread and event-dispatching thread) trying to update the ***same component*** at the ***same time***. Hence, all Swing GUI components *must be created and updated* from the event-dispatching thread, not the main thread of the application.
* main() is executed on main thread , it can't directly instantiate a SwingDemo object. Instead, it create a Runnable object that executes on event-dispatching thread.
* To enable GUI code on the event-dispatching thread, use one of two methods defined by the SwingUtilities class. These are invokeLater( ) and invokeAndWait( ). ***static void invokeLater(Runnable obj)***

***static void invokeAndWait(Runnable obj) throws InterruptedException, InvocationTargetException***

* Here, obj is a Runnable object that will have its ***run()***  method called by the event-dispatching thread. The difference between the two methods is that ***invokeLater()*** returns immediately, but ***invokeAndWait()*** waits until ***obj.run()*** returns.
* Normally use ***invokeLater()***. However, when constructing the initial GUI for an applet, use ***invokeAndWait()***.
* Add a component to the other regions of a border layout by using an overloaded version of add( ): To specify one of the other locations, use following form of add( ):

***void add(Component comp, Object loc)***

Here, comp is the *component* to add and loc specifies the *location* to which it is added. The loc value is typically one of the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***BorderLayout.CENTER*** | ***BorderLayout.EAST*** | ***BorderLayout.NORTH*** | ***BorderLayout.SOUTH*** | ***BorderLayout.WEST*** |

|  |  |
| --- | --- |
| Note: | 1. In the past, you had to use the following statement to add jlab to jfrm: ***jfrm.getContentPane().add(jlab);*** *// old-style*  * Here, getContentPane( ) first obtains a reference to the content pane, and then add( ) adds the *component* to the *container* linked to this *pane*. * This same procedure was required to invoke remove( ) to *remove a component* and setLayout( ) to *set the layout manager* for the *content pane*.   *[You will see explicit calls to getContentPane( ) frequently throughout pre-5.0 code. ]*   * Today, the use of ***getContentPane()*** is no longer necessary. You can simply call ***add()***, ***remove()***, and ***setLayout()***  directly on ***JFrame*** because these methods have been changed so that they automatically operate on the content pane. |
|  | 1. The preceding program does not respond to any events, because JLabel is a passive component. In other words, a JLabel ***does not generate any events***. Hence, the program *does not include* any ***event handlers***. |

**11.11 JButton, JTextField, JCheckBox and JList**

|  |  |
| --- | --- |
| **11.11.1 JButton:** | JButton inherits the abstract class AbstractButton, which defines the ***functionality common to all buttons***. *Swing push buttons* are instances of JButton, can contain text, an image, or both. JButton supplies several constructors. The one used here is: ***JButton(String msg)*** |

* msg specifies the string that will be displayed inside the button.
* ActionEvent generated if *push button* is *pressed*. ***ActionEvent*** is defined by the ***AWT*** and also used by ***Swing***. JButton provides the following methods, to add or remove action listener: ***void addActionListener(ActionListener al)***

***void removeActionListener(ActionListener al)***

* Here, al specifies an object that will *receive event notifications*. This object must be an *instance of a class* that implements the ActionListener *interface*.
* The ***ActionListener*** *interface* defines only one method: actionPerformed( ), It is: ***void actionPerformed(ActionEvent ae)***
* It is the event handler that is called when a button press event has occurred. Implementation of actionPerformed( ) must quickly respond to that event and return. As a general rule, event handlers must not engage in long operations (cause slow application). [*Use* ***separate thread*** *if* ***TIME-CONSUMPTION*** *needed*]
* Using the ActionEvent object passed to ***actionPerformed()***, ***button-press event related information*** can be obtained. For example the action command string associated with the button can be obtained by calling ***getActionCommand()*** on the event object. It is declared as: ***String getActionCommand()***
* The action command identifies the button. By default, this is the string displayed inside the button. Thus, when using *two* or *more buttons* within the *same application*, the action command gives an easy way to determine *which* button was *pressed*.
* Example 6: The following program demonstrates how to create a push button and respond to button- press events.

|  |  |
| --- | --- |
| * Notice that both the java.awt and java.awt.event packages are included. * java.awt is needed because it contains the FlowLayout class, which supports the flow layout manager. * java.awt.event is needed because it defines the ActionListener interface and the ActionEvent class. * Next, the class ButtonDemo is declared. Notice that it implements ActionListener. This means that ButtonDemo objects can be used to receive action events. * Next, a JLabel reference jlab is declared. jlab will be used within the actionPerformed() to display which button has been pressed. * The ButtonDemo constructor begins by creating a JFrame called jfrm. It then sets the *layout* *manager* for the *content* *pane* of jfrm to FlowLayout, as:   ***jfrm.setLayout(newFlowLayout());*** | ***import java.awt.\*;***  ***import java.awt.event.\*;***  ***import javax.swing.\*;***  **class** ButtonDemo **implements** **ActionListener** {  **JLabel** jlab;  ButtonDemo() {  **JFrame** jfrm = **new** JFrame("A Button Example"); *// Create a new JFrame container.*  jfrm.**setLayout**(***new*** ***FlowLayout***()); *// Specify FlowLayout for the layout manager.*  jfrm.**setSize**(220, 90); *// Give the frame an initial size.*  jfrm.**setDefaultCloseOperation**(***JFrame***.***EXIT\_ON\_CLOSE***); *// Exit the program when user closes application.*  **JButton** jbtnUp = **new** JButton("Up"); **JButton** jbtnDown = **new** JButton("Down"); *// Make two buttons.*  jbtnUp.**addActionListener**(**this**); jbtnDown.**addActionListener**(**this**); *// Add action listeners for the buttons.*  jfrm.**add**(jbtnUp); jfrm.**add**(jbtnDown); *// Add the buttons to the content pane.*  jlab = ***new*** ***JLabel***("Press a button."); *// Create a label.*  jfrm.**add**(jlab); *// Add the label to the frame.*  jfrm.**setVisible**(**true**); */\* Display the frame.\*/* }  **public void actionPerformed**(**ActionEvent** ae) { *// Handle button events.*  */\* Use the action command to determine which button was pressed. \*/*  **if**(ae.**getActionCommand**().**equals**("Up")) jlab.**setText**("You pressed Up.");  **else** jlab.**setText**("You pressed down. "); }  **public static void main(String args[])** { *// Create the frame on the event dispatching thread.*  **SwingUtilities.invokeLater**(**new** **Runnable**() { **public** **void** ***run()*** { ***new*** ButtonDemo(); } }); }} |

* After setting size and default close operation, ButtonDemo() creates two buttons : ***JButton jbtnUp = new JButton("Up"); JButton jbtnDown = new JButton("Down");***
* The first button will contain the text "Up", and the second will contain "Down".
* Next, the instance of ButtonDemo referred to via this is added as an action listener for the buttons by these two lines:

***jbtnUp.addActionListener(this); jbtnDown.addActionListener(this);***

* This approach means that the *object that creates the buttons* will also *receive notifications* when a button is pressed.
* Each time a button is pressed, it generates an action event and all registered listeners are notified by calling the ***actionPerformed()*** method.
* The ActionEvent object representing the button event is passed as a parameter. In the case of ButtonDemo, this event is passed to actionPerformed( ) as following way:

***public void actionPerformed(ActionEvent ae) { if(ae.getActionCommand().equals("Up")) jlab.setText("You pressed Up.");***

***else jlab.setText("You pressed down. "); }***

* event is passed via ae. Inside the method, the action command from button is obtained by calling ***getActionCommand()***. ***(by default, the action command is the same as the text displayed by the button).*** Based on the *contents of that string*, the *text in the label* *is set to show* which button was pressed.
* actionPerformed( ) is called on the event-dispatching thread.
* Use ***setActionCommand()*** with a push button: To set the action command to a different value, you can use the ***setActionCommand()***. It works the same for JButton as it does for JTextField.

|  |  |
| --- | --- |
| **11.11.2 JTextField:** | JTextField enables the user to enter a line of text. JTextField inherits the abstract class JTextComponent, which is the superclass of all text components. JTextField defines several constructors. The one we will use is: ***JTextField(int cols)*** |

cols specifies the *width of the text field in* columns. A string can longer than the number of columns. The physical size of the text field on the screen will be *cols columns* wide.

* When you press enter when inputting into a text field, an ActionEvent is generated. Therefore, JTextField provides the ***addActionListener()*** and ***removeActionListener()***. So implement ***actionPerformed()*** method defined by the ActionListener interface. The process is similar button.
* Like a JButton, a JTextField has an action command string associated with it. By default, the action command is the *current content of the text field*.
* To set the action command to a fixed value call the ***setActionCommand()*** method: ***void setActionCommand(String cmd)***
* The string passed in cmd becomes the *new action command*. The text in the *text field is* unaffected. Once you set the action command string, it remains the same no matter what is entered into the text field.
* One reason for explicit setting of the action command is to provide a way to recognize the text field as the ***source of an*** action event.
* Also, if you don’t set the action command associated with a text field, then the contents of the text field might match the action command of another component.
* To obtain the string that is ***currently displayed*** in the text field, call getText( ) on the JTextField instance. It is declared as: **String getText()**
* You can set the text in a JTextField by calling setText( ) : **void setText(String text)** Here, text is the *string* that will be put into the *text field*.
* Example 7: The following program demonstrates JTextField. It contains one text field, one push button, and two labels. One label prompts the user to enter text into the text field. When the user presses ENTER while focus is within the text field, the *contents of the text field are obtained* and *displayed within a second label*. The push button is called Reverse. When pressed, it reverses the contents of the text field.

|  |  |
| --- | --- |
| ***import java.awt.\*; import java.awt.event.\*; import javax.swing.\*;***  **class** TFDemo **implements ActionListener** { **JTextField** jtf;  **JButton** jbtnRev;  **JLabel** jlabPrompt, jlabContents;  TFDemo() { */\* constructor \*/*  **JFrame** jfrm = **new** JFrame("Use a Text Field"); *// Create a new JFrame container.*  jfrm.**setLayout**(new FlowLayout()); *// Specify FlowLayout for the layout manager.*  jfrm.**setSize**(240, 120); *// Give the frame an initial size.*  jfrm.**setDefaultCloseOperation**(***JFrame.EXIT\_ON\_CLOSE***); *// Terminate when application closed.*  jtf = **new** ***JTextField***(10); *// Create a text field of 10 columns wide.*  jtf.***setActionCommand***("myTF"); *// Set the action commands for the text field*.  **JButton** jbtnRev = **new** ***JButton***("Reverse"); *// Create the Reverse button.*  *// Add action listeners for both the text field and the button*  jtf.**addActionListener**(***this***); jbtnRev.**addActionListener**(***this***);  jlabPrompt = **new** ***JLabel***("Enter text: "); jlabContents = **new** ***JLabel***(""); *// Create the labels.*  *// Add the components to the content pane.*  jfrm.**add**(jlabPrompt); jfrm.**add**(jtf); jfrm.**add**(jbtnRev); jfrm.**add**(jlabContents);  jfrm.**setVisible**(***true***); */\* Display the frame.\*/* }  *// Handle action events.*  **public void** **actionPerformed**(***ActionEvent*** ae) { | * Notice that the action command associated with the text field is set to "myTF" by the line:   ***jtf.setActionCommand("myTF");***  After this line executes, the action command string will always be "myTF" no matter what text is currently held in the text field.   * ***actionPerformed()*** use this fact to determine if the action command string is "Reverse", it can mean only one thing: that the Reverse push button has been pressed. Otherwise, the action command was generated by the user pressing ENTER while the text field had input focus. * Notice within actionPerformed() under else:   **jlabContents.setText("You...);**  when ENTER is pressed while focus is inside the text field, an ActionEvent is generated and sent to all registered action listeners, through ***actionPerformed()***. *[Here, this method obtains the current text in the text field by calling* ***getText( )*** *on* ***jtf****. then displays the text through the label referred to by* ***jlabContents****.]* |
| **if**(ae.***getActionCommand***().***equals***("Reverse")) { **String** orgStr = jtf.***getText***(); **String** resStr = ""; *// Reverse button pressed.*  **for**(**int** i=orgStr.***length***()-1; i >=0; i--) resStr += orgStr.***charAt***(i); *// Reverse the string*  jtf.***setText***(resStr); */\* Store the reversed string in the text field.\*/* }  **else** jlabContents.***setText***("You pressed ENTER. Text is: " + jtf.***getText***()); *// Enter pressed*  }    **public static void main(String args[])** { **SwingUtilities.invokeLater**(**new Runnable**() { **public void run**() { **new** TFDemo(); } }); }} | |

|  |  |
| --- | --- |
| **11.11.3 JCheckBox:** | In Swing, a check box is an object of type JCheckBox. JCheckBox inherits AbstractButton and JToggleButton. Thus, a check box is, essentially, a special type of button. JCheckBox defines several constructors. The one used here is: ***JCheckBox(String str)*** |

* It creates a check box that has the text specified by str as a label.
* When a check box is *selected/ deselected (i.e. checked/unchecked)*, an item event is generated. Item events are represented by the ItemEvent class. ***Item events*** are handled by classes that implement the ItemListener *interface*. This interface specifies only one method: ***itemStateChanged()*** , which is shown here:

***void itemStateChanged(ItemEvent ie)***  The item event is received in ie.

* To obtain a reference to the ***item that changed***, call ***getItem()*** on the ***ItemEvent*** object. This method is: ***Object getItem()***

The reference returned must be cast to the ***component class*** being handled, which in this case is JCheckBox.

* You can obtain the text associated with a check box by calling ***getText()***. You can set the text after a check box is created by calling ***setText()***. These methods work the same as they do for JButton.
* The easiest way to determine the state of a check box is to call the ***isSelected()*** method. It is shown here: ***boolean isSelected()*** It returns true if the check box is selected and false otherwise.
* Example 8: The following program demonstrates check boxes. It creates ***three check boxes*** called Alpha, Beta, and Gamma. Each time the state of a box is changed, the current action is displayed. Also, the list of all currently selected check boxes is displayed.

|  |  |
| --- | --- |
| ***import java.awt.\*; import java.awt.event.\*; import javax.swing.\*;***  ***class*** CBDemo ***implements ItemListener*** { **JLabel**  jlabSelected, jlabChanged;  **JCheckBox**  jcbAlpha, jcbBeta, jcbGamma;  CBDemo() {  **JFrame** jfrm = **new** JFrame("Demonstrate Check Boxes"); *// new JFrame container.*  jfrm.**setLayout**(**new** ***FlowLayout***()); *// Specify FlowLayout for the layout manager.*  jfrm.**setSize**(280, 120); *// Give the frame an initial size.*  jfrm.**setDefaultCloseOperation**(***JFrame.EXIT\_ON\_CLOSE***); *// Terminate when application closed*  jlabSelected = **new** ***JLabel***(""); jlabChanged = **new** ***JLabel***(""); *// Create empty labels.* | * ***itemStateChanged()*** performs two functions. First, it reports whether the ***check box has been selected or cleared***. Second, it ***displays all selected check boxes***. It begins by obtaining a reference to the check box that generated the ItemEvent:   ***JCheckBox cb = (JCheckBox) ie.getItem();*** |
| jcbAlpha = **new** ***JCheckBox***("Alpha"); jcbBeta = **new** ***JCheckBox***("Beta"); jcbGamma = **new** ***JCheckBox***("Gamma"); *// Make check boxes.*  *// Events generated by the check boxes are handled in common by the itemStateChanged() method implemented by CBDemo.*  jcbAlpha.**addItemListener**(*this*); jcbBeta.**addItemListener**(*this*); jcbGamma.**addItemListener**(*this*);  *// Add check boxes and labels to the content pane.*  jfrm.**add**(jcbAlpha); jfrm.**add**(jcbBeta); jfrm.**add**(jcbGamma); jfrm.**add**(jlabChanged); jfrm.**add**(jlabSelected); | |
| jfrm.***setVisible***(***true***); */\* Display the frame.\*/* }  *// This is the handler for the check boxes.*  **public void itemStateChanged**(***ItemEvent*** ie) { String str = ""; */\* notice the following cast \*/*  */\* Obtain a reference to the check box that caused the event.\*/* JCheckBox cb = (JCheckBox) ie.getItem();  *// Determine what happened : what check box changed.*  **if**(cb.**isSelected**()) jlabChanged.**setText**(cb.**getText**() + " was just selected.");  **else** jlabChanged.**setText**(cb.**getText**() + " was just cleared.");  *// Report all selected boxes.*  **if**(jcbAlpha.**isSelected**()) { str += "Alpha "; }  **if**(jcbBeta.**isSelected**()) { str += "Beta "; }  **if**(jcbGamma.**isSelected**()) { str += "Gamma"; }  jlabSelected.**setText**("Selected check boxes: " + str); } | * The cast to JCheckBox is necessary because ***getItem()*** returns a reference of type Object. * Next, ***itemStateChanged()*** calls ***isSelected()*** on cb to determine the current state of the check box. If ***isSelected()*** returns true, it means that the user selected the check box. Otherwise, the check box was cleared. It then sets the ***jlabChanged*** label to reflect states. * ***itemStateChanged()*** checks the selected state of each check box, building a string that contains the names of those that are selected. It displays this string in the ***jlabSelected*** label. |

**public static void main(String args[])** {

**SwingUtilities.invokeLater**(**new Runnable()** { **public void run()** { **new** CBDemo(); } }); }}

|  |  |
| --- | --- |
| **11.11.4 JList:** | JList component is Swing’s basic list class. It supports the selection of one or more items from a list. Although often the ***list consists of*** strings, it is possible to create a ***list of just about any object*** that can be ***displayed***. In the past, the items in a JList were represented as Object references. Now JList is generic, declared like: ***class JList<E>*** Here, E represents the type of the items in the list. As a result, JList is now type-safe. JList has several constructors. The one used here is: ***JList(E[] items)*** creates a JList, contains the items in the array specified by items. |

* JScrollPane, which is a container that automatically provides scrolling for its contents is used to wrap a JList. The constructor: ***JScrollPane(Component comp)***

Here, comp specifies the component to be scrolled, which in this case will be a JList. When you wrap a JList in a JScrollPane, long lists will automatically be scrollable.

* A JList generates a ListSelectionEvent when the user makes/changes a selection, deselects an item. It is handled by ListSelectionListener, interface in javax.swing.event. This listener specifies only one method: ***void valueChanged(ListSelectionEvent le)*** Here, le is a reference to the object that generated the event.
* Although ListSelectionEvent does provide some methods of its own, often you will interrogate the JList object itself to determine what has occurred. ListSelectionEvent is also packaged in javax.swing.event.
* By default, a JList allows to select multiple ranges of items within the list, but you can change this behavior by calling ***setSelectionMode( )***, which is defined by JList. It is: ***void setSelectionMode(int mode)*** Here, mode specifies the selection mode. It must be one of these values defined by the ListSelectionModel interface (which is packaged in javax.swing):

|  |  |  |
| --- | --- | --- |
| 1. **SINGLE\_SELECTION** | 1. **SINGLE\_INTERVAL\_SELECTION** | 1. **MULTIPLE\_INTERVAL\_SELECTION** |

* The default, multiple-interval selection lets the user ***select multiple ranges of items*** within a list. With single-interval selection, the user can ***select one range*** ***of*** ***items***. With single selection, the user can select ***only a single item***. Of course, a single item can be selected in the other two modes, too.
* Call ***getSelectedIndex()***  to obtain the index of the first selected item or only selected item of single-selection mode: ***int getSelectedIndex()***
* Indexing begins at zero. So, if the first item is selected ***0*** is returned. If no item is selected, ***–1*** is returned.
* You can obtain an array ***containing all selected items*** by calling ***getSelectedIndices()***: ***int[ ] getSelectedIndices( )*** In the returned array, the indices are ordered from smallest to largest. If a zero-length array is returned, it means that no items are selected.
* Example 9: Following creates a simple JList, which holds a list of names. Each time a name is selected in the list, a ***ListSelectionEvent*** is generated, which is handled by the ***valueChanged()*** method defined by ListSelectionListener. It responds by obtaining the index of the selected item and showing corresponding name.

|  |  |
| --- | --- |
| * Notice the names array near the top of the program. Inside ListDemo( ), a JList called jlst is constructed using the names array. When array constructor is used *(as it is in this case)*, a JList instance is automatically created that contains the contents of the array. i.e, the list will contain the names in names. * Next, the selection mode is set to single selection. Then, jlst is wrapped inside a JScrollPane, with size of the scroll 120 by 90. In Swing, the ***setPreferredSize()*** method sets the ideal size of a component.   [ Some layout managers are free to ignore this request. ]   * Inside the ***valueChanged()*** *event handler*, the index of the item selected is obtained by calling ***getSelectedIndex()***. Because the list has been set to single-selection mode, this is also the index of the only item selected. This index is then used to index the names array to obtain the selected name. * Notice that this index value is tested against ***–1***. Since, if no item has been selected ***-1*** returned. This will be the case when the selection event handler is called if the user has deselected an item. * A selection event is generated when the user selects or deselects an item. | ***import javax.swing.\*; import javax.swing.event.\*; import java.awt.\*; import java.awt.event.\*;***  **class** ListDemo **implements ListSelectionListener** { **JList<String>** jlst;  **JLabel** jlab;  **JScrollPane** jscrlp;  *// Create an array of names.*  **String** names[] = { "Sherry", "Jon", "Rachel", "Sasha", "Josselyn", "Randy",  "Tom", "Mary", "Ken", "Andrew", "Matt", "Todd" };  ListDemo() {  **JFrame** jfrm = **new** JFrame("JList Demo"); *// Create a new JFrame container.*  jfrm.**setLayout**(**new** FlowLayout()); *// Specify a flow Layout.*  jfrm.**setSize**(200, 160); *// Give the frame an initial size.*  jfrm.**setDefaultCloseOperation**(***JFrame.EXIT\_ON\_CLOSE***); *// Terminate when application closed*  jlst = **new** **JList<String>**(names); *// Create a JList.*  jlst.**setSelectionMode**(***ListSelectionModel.SINGLE\_SELECTION***); *// single-selection. mode*  jscrlp = **new** ***JScrollPane***(jlst); *// Wrap the list in a scroll pane*  jscrlp.**setPreferredSize**(**new** ***Dimension***(120, 90)); *// Set the preferred size of the scroll pane.*  jlab = **new *JLabel***("Please choose a name"); *// Make a label that displays the selection.*  jlst.***addListSelectionListener***(***this***); *// Add list selection handler : Listen for list selection events.*  jfrm.**add**(jscrlp); jfrm.**add**(jlab); *// Add the list and label to the content pane.*  jfrm.**setVisible**(***true***); */\* Display the frame \*/*  }  *// Handle list selection events.*  **public void valueChanged**(***ListSelectionEvent*** le) {  **int** idx = jlst.**getSelectedIndex**(); *// Get the index of the selected-deselected item.*  **if**(idx != -1) jlab.**setText**("Selected: " + names[idx]); *// Display selection, if item selected.*  **else** jlab.**setText**("Please choose a name"); */\* Otherwise, reprompt.\*/*  }  **public static void main(String args[])** {  **SwingUtilities.invokeLater**(**new Runnable**() { **public void run**() { **new** ListDemo(); } }); }} |

**11.12 Use *Anonymous INNER CLASSES* or *LE* to Handle Events**

Different classes can handle different events and these classes would be separate from the main class of the application (use separate listener classes). However, two other approaches offer powerful alternatives.

|  |  |
| --- | --- |
| * First, you can implement listeners through the use of anonymous inner classes. | * Second, in some cases, you can use a LE to handle an event. |

* Anonymous inner classes: Anonymous inner classes are *inner classes that don’t have a name*. Instead, an instance of the class is simply generated “on the fly” as needed. They used to implement some types of event handlers. Eg:, given a JButton called jbtn, you could implement an action listener for it like this:

jbtn.**addActionListener**(**new** ***ActionListener***() { **public** **void** ***actionPerformed***(***ActionEvent*** ae) { */\* Handle action event here. \*/* } } );

* Here, an anonymous inner class is created that implements the ActionListener interface. The *body of the inner class* begins after the **{** that follows

**new ActionListener()**. Also notice that the call to **addActionListener()** ends with a **)** and a **;** just like normal.

* The same basic syntax and approach is used to create an anonymous inner class for any event handler. Of course, for different events, you specify different event listeners and implement different methods.
* One advantage to using an anonymous inner class is that the component that invokes the *class’ methods* is *already known*. Eg: there is no need to call ***getActionCommand()*** because this implementation of ***actionPerformed()*** will only be called by events generated by ***jbtn***.
* FI and LE : In the case of an event whose listener defines a FI, you can handle the event by use of a LE. For example, action events can be handled with a LE because ActionListener defines only one abstract method, ***actionPerformed()***. Using a LE to implement ActionListener provides a *compact alternative* to explicitly declaring an *anonymous inner class*. Eg: again assuming a JButton called jbtn, you could implement the action listener like this:

**jbtn.addActionListener((ae) -> {** */\* Handle action event here. \*/* **});**

* As was the case with the anonymous inner class approach, the object that generates the event is known. In this case, the LE applies only to the jbtn button. Of course, in cases in which an event can be handled by use of a single expression, it is not necessary to use a block lambda.
* In general, you can use a LE to handle an event when its listener defines a FI. Eg: ItemListener is also a FI.

**11.13 Create a Swing Applet**

Swing-based applets are similar to AWT-based applets, but: A Swing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for Swing.

* JApplet is a top-level Swing container. Therefore, it includes the various panes described earlier. As a result, all components are added to JApplet***’s content pane*** in the same way that components are added to JFrame***’s content pane***.
* Swing applets use the same four life-cycle methods: ***init()***, ***start()***, ***stop()***, and ***destroy()***. [Override only those methods that are needed by the applet.] In general, painting is accomplished differently in Swing than it is in the AWT. Thus, a Swing applet will not usually override the ***paint()*** method.

Note: All interaction with components in a Swing applet must take place on the event-dispatching thread.

* Example 10: Following provides the same functionality as the push-button example. It also uses anonymous inner classes to implement the action event handlers.

|  |  |
| --- | --- |
| ***import javax.swing.\*; import java.awt.\*; import java.awt.event.\*;***  */\* <applet code="MySwingApplet" width=200 height=80> </applet> \*/*  **public class** MySwingApplet **extends** **JApplet** { **JButton** jbtnUp;  */\* Swing applets must extend JApplet \*/* **JButton** jbtnDown;  **JLabel** jlab; | * First, MySwingApplet extends JApplet. As explained, all Swing-based applets extend JApplet rather than Applet. * Second, the ***init()*** method initializes the Swing components on the event-dispatching thread by setting up a call to ***makeGUI()***. Notice that this is accomplished through the use of ***invokeAndWait()*** rather ***than invokeLater()***. Applets must use ***invokeAndWait()*** because the ***init()*** must not return until the entire initialization process has been completed. In essence, the ***start()*** method cannot be called until after initialization, which means that the *GUI must be fully constructed*. |
| *// Initialize the applet.*  **public** **void** **init()** { **try** { **SwingUtilities.invokeAndWait**(***new Runnable*** () { **public void run**() { **makeGUI**(); */\* initialize the GUI \*/* } }); }  **catch**(***Exception***exc) { **System.out.println**("Can't create because of "+ exc); }  } | |
| *// This applet does not need to override start(), stop(), or destroy().*  **private** **void** **makeGUI**() { *// Set up and initialize the GUI.*  **setLayout**(**new** **FlowLayout**()); *// Set the applet to use flow layout.* | * Inside ***makeGUI()***, the two buttons and label are created, and the action listeners are added to the buttons. Notice that anonymous inner classes are used to implement the action event handlers. |
| jbtnUp = **new** ***JButton***("Up"); jbtnDown = **new** ***JButton***("Down"); *// Make two buttons.*  *// Add action listener for Up button.*  jbtnUp.**addActionListener**(**new** **ActionListener**() { **public void actionPerformed**(**ActionEvent** ae) { jlab.**setText**("Pressed Up."); } });  *// Add action listener for Down button.*  jbtnDown.**addActionListener**(**new** **ActionListener**() { **public void actionPerformed**(**ActionEvent** ae) { jlab.**setText**("Pressed down."); } }); | |
| **add**(jbtnUp); **add**(jbtnDown); *// Add the buttons to the content pane*  jlab = **new** ***JLabel***("Press a button."); *// Create a text-based label*  **add**(jlab); *// Add the label to the content pane*  }} | * Here the object that causes the event is known because it is the object on which the anonymous inner class is instantiated. So, it is not necessary to obtain the action command to determine which button generated the event. (Using a LE would also provide the same advantage.) * Finally, the components are added to the content pane. |

**11.14 JavaFX fundamentals**

The JavaFX framework is contained in packages that begin with the javafx prefix. The central metaphor implemented by JavaFX is the stage. A stage contains a scene. Thus, a stage defines a ***space*** and a scene defines ***what goes in that space***. Or, put another way, a stage is a container for scenes and a scene is a container for the items that comprise the scene.

* All JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the Stage and Scene classes. To create a JavaFX application, you will, at minimum, add at least one Scene object to a Stage.
* Stage: Stage is a top-level container. All JavaFX applications automatically have access to one Stage, called the ***primary stage***. The ***primary stage*** is supplied by the ***run-time system*** when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.
* Scene: Scene is a container for the items that construct the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, add those elements to an instance of Scene.
* Nodes and Scene Graphs Nodes, child, parent/branch and leaves/terminal nodes: The individual elements of a scene are called nodes. For example, a push button control is a node. However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a parent node or branch node. Nodes without children are terminal nodes and are called leaves.
* Tree and root: The collection of all nodes in a scene creates what is referred to as a scene graph, which comprises a tree. There is one special type of node in the scene graph, called the root node. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.
* The base class for all nodes is Node. There are other classes that are, either directly or indirectly, subclasses of Node. These include Parent, Group, Region, and Control, to name a few.
* Layouts: JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the FlowPane class provides a flow layout and the GridPane class supports a row/column grid-based layout. Several other layouts, such as BorderPane (similar to the AWT’s BorderLayout), are available. Each inherits Node. The layouts are packaged in javafx.scene.layout.
* The Application Class and the Life-cycle Methods: A JavaFX application must be a subclass of the Application class (i.e. ***extend Application***), which is packaged in javafx.application. The Application class defines three ***life-cycle methods*** that your application can override. These are called ***init()***, ***start()***, and ***stop()*** [Recall Java/C# 11.2: Applet defines four life-cycle-methods] in the order in which they are called:
* ***void init()*** : called when the application begins execution. It is used to perform various initializations. However, it cannot be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.
* ***abstract void start(Stage primaryStage)*** : The ***start()*** method is called after ***init()***. This is where your application begins and it can be used to construct and set the scene. Notice that it is passed a reference to a Stage object. This is the stage provided by the run-time system and is the primary stage. Notice that this method is abstract. Thus, it must be overridden by your application.
* ***void stop()*** : When your application is terminated, the ***stop( )*** is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.
* Launching a JavaFX Application: To start a free-standing JavaFX application, must call the ***launch()*** defined by Application. It has two forms. One form is:

***public static void launch(String ... args)***

* Here, args is a possibly empty list of strings that typically specify command-line arguments. When called, ***launch()*** causes the application to be constructed, followed by calls to ***init()*** and ***start()***. The ***launch()*** method will not return until after the ***application*** has ***terminated***. This version of ***launch()*** starts the subclass of Application from which ***launch()*** is called.
* The second form of ***launch()*** lets you specify a class other than the enclosing class to start.
* Note: JavaFX applications that have been packaged by using the javafxpackager tool (or its equivalent in an IDE) do not need to include a call to ***launch()***. However, its inclusion often simplifies the test/debug cycle, and it lets you use the program without creating a JAR file.
* A JavaFX Application Skeleton: A message noting when each life-cycle method is called is displayed on the console. The complete skeleton is shown here:

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*;***

**public class** JavaFXSkel **extends** **Application** {

**public static void main(String[] args)** { **System.out.println**("Launching JavaFX application.");

**launch**(args); */\* Start the JavaFX application by calling launch().\*/* }

**public void init**() { **System.out.println**("Inside the init() method."); } *// Override the init() method.*

**public void start**(**Stage** myStage) { **System.out.println**("Inside the start() method."); *// Override the start() method.*

myStage.**setTitle**("JavaFX Skeleton."); *// Give the stage a title.*

**FlowPane** rootNode = **new** FlowPane(); *// Create a root node. In this case, a flow layout is used*

**Scene** myScene = **new** Scene(rootNode, 300, 200); *// Create a scene.*

myStage.**setScene**(myScene); *// Set the scene on the stage.*

myStage.**show**(); */\* Show the stage and its scene \*/.* }

**public void stop**() { **System.out.println**("Inside the stop() method."); } */\* Override the stop() method.\*/*  }

* Override the ***init()*** and ***stop()***only if your application must perform special startup or shutdown actions. Otherwise use the default implementations.
* Program begins by importing four packages. The first is javafx.application, which contains the Application class. The Scene class is packaged in javafx.scene, and Stage is packaged in javafx.stage. The javafx.scene.layout package provides several layout panes. The one used by the program is FlowPane.
* Next, the ***application class*** JavaFXSkel is created. Notice that it extends Application. JavaFXSkel contains four methods: ***main()***, ***init()***, ***start()*** and ***stop()***.
* ***main()*** is used to launch the application via a call to ***launch()***. Notice that the args parameter to ***main()*** is passed to the ***launch()***. Although this is a common approach, you can pass a different set of parameters to ***launch()***, or none at all.
* Note: ***launch()*** is required by a free-standing application, but not in other cases. When it is not needed, ***main()*** is also not needed.
* When the application begins, the ***init()*** is called first by the JavaFX ***run-time system*** (it would normally be used to initialize some aspect of the application). If no initialization is required, it is not necessary to override ***init()*** because an empty, default implementation is provided.
* ***init()*** cannot be used to create the stage or scene portions of a GUI. Rather, these items should be constructed and displayed by the ***start()*** method.
* After ***init()*** finishes, the ***start()*** executes. It is here that the initial scene is created and set to the primary stage. First, notice that ***start()*** has a parameter of type Stage. When ***start()*** is called, this parameter will receive a reference to the primary stage of the application. It is to this stage that you will set a scene for the application.
* After displaying a message on the console that ***start()*** has begun execution, it sets the title of the stage using this call to ***setTitle()***:

***myStage.setTitle("JavaFX Skeleton.");*** This title becomes the name of the main application window.

* Next, a root node for a scene is created. In this case, a FlowPane is used for the root node, but there are several other classes that can be used for the root.

***FlowPane rootNode = new FlowPane();*** A FlowPane uses a ***flow layout***, in which elements are positioned ***line-by-line***, with lines ***wrapping*** as needed. (Thus, it works much like the FlowLayout class used by the AWT and Swing.) In this case, a ***horizontal flow*** is used, but it is possible to specify a ***vertical*** ***flow*** (vertical and horizontal gap between elements, and an alignment also possible).

* The line uses the root node to construct a Scene: ***Scene myScene = new Scene(rootNode, 300, 200);*** Scene has several constructor. The one used here creates a scene that has the specified root with width and height: ***Scene(Parent rootnode, double width, double height)***
* Notice that the type of rootnode is Parent. It is a subclass of Node and encapsulates nodes that can have children.
* The next line in the program sets myScene as the scene for myStage: ***myStage.setScene(myScene);*** Here, ***setScene()*** is a method defined by Stage that sets the scene to that specified by its argument.
* If you don’t make further use of the scene, you can combine the previous two steps, as: ***myStage.setScene(new Scene(rootNode, 300, 200));***
* The last line in ***start()*** displays the stage and its scene: ***myStage.show();*** ***show()*** shows the window that was created by the stage and scene.
* Closing the application removes its window from the screen and stop( ) is called by the JavaFX run-time system. In this case, stop( ) simply displays a message on the console. [stop( ) would not normally display anything. If your application does not need to handle any shutdown actions, there is no reason to override stop( ) .]
* Compiling and Running a JavaFX Program: You can run a JavaFX program as a ***stand-alone desktop application***, inside a web browser, or as a Web Start application. However, different ancillary files may be needed in some cases, such as an HTML file or a Java Network Launch Protocol (JNLP) file.
* The easiest way to compile a JavaFX application is to use an IDE that fully supports JavaFX programming. Here use the command-line tools. Just compile and run the application in the normal way, using javac and java. This creates a ***stand-alone application that runs on the desktop***.
* The Application Thread: Since, you cannot use the ***init()*** to construct a stage or scene. You also cannot create these items inside the application’s constructor. The reason is that a stage or scene must be constructed on the application thread.
* However, the application’s constructor and the ***init()*** are called on the main thread, also called the launcher thread. Thus, they can’t be used to construct a stage or scene. Instead, you must use the ***start()*** to create the initial GUI because ***start()*** is called on the application thread.
* Any changes to the GUI currently displayed must be made from the application thread. In JavaFX, events are sent to a program on the application thread. Therefore, event handlers can be used to interact with the GUI. The ***stop()*** method is also called on the application thread.

**11.15 JavaFX Label**

The simplest control is the label because it just displays a message or an image. The label is a good way to introduce the techniques needed to begin building a scene graph.

* The JavaFX label is an ***instance*** of the Label class, which is ***packaged*** in javafx.scene.control. Label inherits Labeled and Control, among other ***classes***. The Labeled class defines several features that are common to all ***labeled elements*** (that is, those that can contain text), and Control defines features related to ***all controls***.
* The Label constructor that we will use is: ***Label(String str)*** The string that is displayed is specified by str.
* Once you have created a label (or any other control) it must be added to the scene’s content, which means adding it to the scene graph. To do this, you will first call ***getChildren( )*** on the root node of the scene graph. It returns a list of the child nodes in the form of an ***ObservableList<Node>***.
* ObservableList is packaged in javafx.collections, and it inherits java.util.List, which is part of Java’s Collections Framework. List defines a collection that represents a ***list of objects***.
* It is easy o use ObservableList to add child nodes. Simply call ***add()*** on the list of child nodes returned by ***getChildren()***, passing in a reference to the node to add, which in this case is a label.
* addAll( ) : ObservableList provides a method called ***addAll()*** that can be used to add two or more children to the scene graph in a single call.
* Remove a node: To remove a control from the scene graph, call ***remove()*** on the ObservableList. Eg: ***rootNode.getChildren().remove(myLabel);***

removes myLabel from the scene (see next example).

* In general, ObservableList supports a wide range of list-management methods. Here are two examples. You can determine if the list is empty by calling ***isEmpty()***. You can obtain the number of ***nodes*** in the list by ***calling size()***.
* Example 11: A simple JavaFX application that displays a label:

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*; import javafx.scene.control.\*;***

***public class*** JavaFXLabelDemo ***extends*** ***Application***{ **public static void main(String[] args)** { **launch**(args); */\* Start the JavaFX \*/* }

**public void start**(**Stage** myStage) { *// Override the start()*

|  |  |
| --- | --- |
| * In the program, pay special attention to this line:   ***rootNode.getChildren().add(myLabel);***  It adds the label to the list of children for which rootNode is the parent. Although this line could be separated into its individual pieces if necessary. | myStage.**setTitle**("Use a JavaFX label."); *// Give the stage a title.*  **FlowPane** rootNode = **new** ***FlowPane***(); *// Use a FlowPane for the root node.*  **Scene** myScene = **new** ***Scene***(rootNode, 300, 200); *// Create a scene.*  myStage.**setScene**(myScene); *// Set the scene on the stage.*  **Label** myLabel = **new** ***Label***("JavaFX is a powerful GUI"); *// Create a label.*  rootNode.***getChildren***().***add***(myLabel); *// Add the label to the scene graph*.  myStage.**show**(); */\*Show the stage and its scene.\*/* }} |

**11.16Events handling: Buttons, CheckBox, ListView and TextField**

* Event Basics: The base class for JavaFX events is the Event class, which is packaged in javafx.event. Event inherits java.util.EventObject. Several subclasses of Event are defined. The one that we will use here is ActionEvent. It encapsulates action events generated by a button.
* In general, JavaFX uses the delegation event model approach to event handling. To handle an event, you must first register the handler that acts as a listener for the event. When the event occurs, the listener is called. It must then respond to the event and return.
* Events are handled by implementing the EventHandler interface, which is also in javafx.event. It is a generic interface with the following form: Interface ***EventHandler<T extends Event>*** Here, T specifies the type of event that the handler will handle. It defines one method, called ***handle()***, which receives the event object as a parameter. It is shown here: ***void handle(T eventObj)***
* In this case, eventObj is the event that was generated. Typically, event handlers are implemented through anonymous inner classes or lambda expressions, but you can use stand-alone classes ( if one ***event handler*** will handle events from ***more than one source***).
* Button Control: In JavaFX, the ***push button control*** is provided by the Button class, which is in javafx.scene.control. Button inherits a fairly long list of ***base classes*** that include ButtonBase, Labeled, Region, Control, Parent, and Node. If you examine the API ***documentation*** for Button, you will see that much of its functionality comes from its base classes.
* The Button constructor we will use is shown here: ***Button(String str)*** In this case, str is the message that is displayed in the button.
* When a button is pressed, an ActionEvent is generated. ActionEvent is packaged in javafx.event. You can *register a listener* for this event by calling ***setOnAction()*** on the button. It has this general form: ***final void setOnAction(EventHandler<ActionEvent> handler)***
* Here, handler is the handler being registered. The ***setOnAction()*** sets the property onAction, which stores a ***reference*** to the ***handler***.
* As with all other Java event handling, your handler must respond to the event ***as fast as possible*** and then return. If your handler consumes too much time, it will noticeably ***slow down the application***. For ***lengthy operations***, you must use a ***separate thread of execution***.
* Example 12: The following program demonstrates event handling and the Button control. It uses two buttons and a label. The buttons are called Up and Down. Each time a button is pressed, the content of the label is set to display which button was pressed. Thus, it functions similarly to the JButton example in Swing.

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*;***

***import javafx.scene.control.\*; import javafx.event.\*; import javafx.geometry.\*;***

**public** **class** JavaFXEventDemo **extends** **Application** { **Label** response;

**public static void main(String[] args)** { **launch**(args); /\* Start the JavaFX application \*/ }

**public void start**(**Stage** myStage) { */\* Override the start() method.\*/*

|  |  |
| --- | --- |
| * Buttons are created by these two lines:   ***Button btnUp = new Button("Up");***  ***Button btnDown = new Button("Down");***   * Next, an action event handler is set for each of these buttons. The sequence for the Up button is shown here:   ***btnUp.setOnAction(new EventHandler<ActionEvent>()***  ***{ public void handle(ActionEvent ae) {***  ***response.setText("You pressed Up."); } });***  Buttons respond to events of type ActionEvent. To register a handler for these events, ***setOnAction( )*** is called on the button. It uses an anonymous inner class to implement the EventHandler interface. (Recall that EventHandler defines only ***handle()***.) Inside ***handle( )***, the text in the response label is set to reflect the fact that the Up button was pressed. Notice that this is done by calling ***setText( )*** on the label. | myStage.**setTitle**("Use JavaFX Buttons and Events."); *// Give the stage a title.*  *// Use a FlowPane for the root node. In this case, vertical and horizontal gaps of 10.*  **FlowPane** rootNode = **new** ***FlowPane***(10, 10);  rootNode.**setAlignment**(***Pos.CENTER***); *// Center the controls in the scene.*  **Scene** myScene = **new** ***Scene***(rootNode, 300, 100); *// Create a scene.*  myStage.**setScene**(myScene); *// Set the scene on the stage.*  response = **new** ***Label***("Push a Button"); *// Create a label.*  **Button** btnUp = **new** ***Button***("Up"); *// Create push button*  **Button** btnDown = **new** ***Button***("Down"); *// Create push button*  *// Handle the action events for the Up button.*  btnUp.**setOnAction**(**new** **EventHandler<ActionEvent>()** {  **public void handle**(***ActionEvent*** ae) { response.***setText***("*You pressed Up.*"); } });  *// Handle the action events for the Down button.*  btnDown.**setOnAction**(**new** **EventHandler<ActionEvent>()** {  **public void handle**(***ActionEvent*** ae) { response.***setText***("*You pressed Down.*"); } }); |

*/\* Add the label and buttons to the scene graph \*/* rootNode.***getChildren***().***addAll***(btnUp, btnDown, response);

*/\*Show the stage and its scene.\*/* myStage.***show***(); }}

* After the event handlers have been set, the response label and the buttons btnUp and btnDown are added to the scene graph by using a call to ***addAll()***:

***rootNode.getChildren().addAll(btnUp, btnDown, response);***

* ***addAll()*** adds a list of nodes to the invoking parent node. Of course, these nodes could have been added by three separate calls to ***add()***.
* When the root node is created, this statement is used: ***FlowPane rootNode = new FlowPane(10, 10);***
* Here, the FlowPane constructor is passed two values. These specify the horizontal and vertical gap that will be left around elements in the scene. If these gaps are not specified, then two elements (such as two buttons) would be positioned in such a way that no space was between them.
* To sets the alignment of the elements in the FlowPane: ***rootNode.setAlignment(Pos.CENTER);***
* Here, the ***alignment of the elements*** is centered. By calling ***setAlignment()*** on the FlowPane. The value ***Pos.CENTER*** specifies that both a vertical and horizontal center will be used. Other alignments are possible. Pos is an enumeration that specifies alignment constants. It is packaged in javafx.geometry.
* The preceding program used anonymous inner classes to handle button events. However, because the EventHandler interface defines only ***one abstract method***, ***handle()***, a LE could have passed to ***setOnAction()***, instead. Eg:  ***btnUp.setOnAction( (ae) -> response.setText("You pressed Up.") );***
* CheckBox: In JavaFX, the check box is encapsulated by the CheckBox class. Its immediate superclass is ButtonBase. Thus it is a ***special type of button***. In JavaFX, CheckBox supports three states. The first two are checked or unchecked, as default behavior. The third state is indeterminate (also called undefined), it is used to indicate that the state of the check box has not been set. To use the indeterminate state, you will need to explicitly enable it.
* Here is the CheckBox constructor that we will use:  **CheckBox(String str)**  It creates a check box that has the text specified by str as a label. As with other buttons, a CheckBox generates an *action event* when it is selected.
* Example 13: Following displays four check boxes that represent different types of computers. They are labeled Smartphone, Tablet, Notebook, and Desktop. Each time a check-box state changes, an action event is generated. It is handled by displaying the new state (selected or cleared) and by displaying a list of all selected boxes.

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*;***

***import javafx.scene.control.\*; import javafx.event.\*; import javafx.geometry.\*;***

**public class** CheckboxDemo **extends Application** { **CheckBox** cbSmartphone, cbTablet, cbNotebook, cbDesktop;

**Label** response, selected;

**String** computers;

**public static void main(String[] args)** { **launch**(args); }

**public void start**(**Stage** myStage) { myStage.**setTitle**("Demonstrate Check Boxes"); *// Give the stage a title.*

|  |  |
| --- | --- |
| * Each time a check box is changed, an ActionEvent is generated. The handlers for these events first report whether the check box was selected or cleared. To do this, they call the isSelected( ) method on the event source. It returns true if the check box was just selected, and false if it was just cleared. Next, the showAll( ) method is called, which displays all selected check boxes. * Notice that it uses a vertical flow pane for the layout, as: *FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);* * By default, FlowPane flows horizontally. A vertical flow is created by passing the value Orientation.VERTICAL as the first argument to the FlowPane constructor. | **FlowPane** rootNode = **new** FlowPane(Orientation.VERTICAL, 10, 10);  rootNode.***setAlignment***(Pos.CENTER); *// Center the controls in the scene*.  **Scene** myScene = **new** ***Scene***(rootNode, 230, 200); *// Create a scene.*  myStage.***setScene***(myScene); *// Set the scene on the stage.*  **Label** heading = **new** ***Label***("What Computers Do You Own?");  response = **new** ***Label***(""); *// Create a label that will report the state change of a check box.*  selected = **new** ***Label***(""); *// Create a label that will report all selected check boxes.*  *// Create the check boxes.*  cbSmartphone = **new** ***CheckBox***("Smartphone"); cbTablet = **new** ***CheckBox***("Tablet");  cbNotebook = **new** ***CheckBox***("Notebook"); cbDesktop = **new** ***CheckBox***("Desktop"); |

*// Handle action events for the check boxes.*

cbSmartphone.**setOnAction**(

**new EventHandler<ActionEvent>**() {**public void handle**(***ActionEvent*** ae) { **if**(cbSmartphone.**isSelected**()) response.**setText**("*Smartphone selected.*");

**else** response.**setText**("*Smartphone cleared*."); **showAll**(); } });

cbTablet.**setOnAction**(

**new EventHandler<ActionEvent>**() { **public void handle**(***ActionEvent*** ae) { **if**(cbTablet.**isSelected**()) response.**setText**("*Tablet selected*.");

**else** response.**setText**("*Tablet cleared*."); **showAll**(); } });

cbNotebook.**setOnAction**(

**new EventHandler<ActionEvent>**() { **public void handle**(***ActionEvent*** ae) { **if**(cbNotebook.**isSelected**()) response.**setText**("*Notebook selected*.");

**else** response.**setText**("*Notebook cleared*."); **showAll**(); } });

cbDesktop.**setOnAction**(

**new EventHandler<ActionEvent>**() { **public void handle**(***ActionEvent*** ae) { **if**(cbDesktop.**isSelected**()) response.**setText**("*Desktop selected*.");

**else** response.**setText**("*Desktop cleared*."); **showAll**(); } });

rootNode.**getChildren**().**addAll**(heading, cbSmartphone, cbTablet, cbNotebook, cbDesktop, response, selected); *// Add controls to the scene graph.*

*/\* Show the stage and its scene.\*/* myStage.**show**(); **showAll**(); } *// start() ends here*

*// Update and show the selections. Use isSelected( ) to determine the state of the check boxes.*

**void** showAll() { computers = "";

**if**(cbSmartphone.**isSelected**()) computers = "Smartphone "; **if**(cbTablet.**isSelected**()) computers += "Tablet ";

**if**(cbNotebook.**isSelected**()) computers += "Notebook "; **if**(cbDesktop.**isSelected**()) computers += "Desktop";

selected.**setText**("Computers selected: " + computers); } }

* indeterminate state: CheckBox also supports a third, indeterminate state, which can be used to indicate that the state of the box has ***not yet been set*** or that an option is ***not applicable*** to a situation. The indeterminate state for a check box must be explicitly enabled. It is ***not provided by default***. Also, the event handler for the check box must also handle the indeterminate state.
* To enable the indeterminate state in a check box, call setAllowIndeterminate(): ***final void setAllowIndeterminate(boolean enable)***
* If enable is true, the indeterminate state is enabled. Otherwise, it is disabled. When the indeterminate state is enabled, the user can select between checked, unchecked, and indeterminate.
* To determine if a check box is in the indeterminate state, call ***isIndeterminate()***: ***final boolean isIndeterminate( )***
* It returns true if the check box state is indeterminate and false otherwise.
* The event handler for the check box will need to test for the indeterminate state.
* Example 14: Following enables indeterminate state to the Smartphone check box in CheckboxDemo program, just shown. To enable the indeterminate state on the Smartphone check box, add this line: ***cbSmartphone.setAllowIndeterminate(true);***
* Add the Smartphone event handler, as:

cbSmartphone.**setOnAction**(

**new EventHandler<ActionEvent>**(){ **public void handle**(***ActionEvent*** ae){ **if**(cbSmartphone.**isIndeterminate**()) response.**setText**("*Smartphone indeterminate*.");

**else** **if**(cbSmartphone.**isSelected**()) response.**setText**("Smartphone selected.");

**else** response.**setText**("Smartphone cleared."); showAll(); } });

* ListView: ListView is a generic class that is declared like this: ***class ListView<T>*** Here, T specifies the type of entries stored in the list view. Often, these are entries of type String, but other types are also allowed.
* Here is the ListView constructor that we will use: ***ListView(ObservableList<T> list)***
* The list of items to be displayed is specified by list. It is an object of type ObservableList. As explained earlier, ObservableList supports a list of objects. By default, a ListView allows single-selection mode. You can allow multiple selections by changing the selection mode.
* Probably the easiest way to create an ObservableList for use in a ListView is to use the factory method ***observableArrayList( )***, which is a static method defined by the FXCollections class (which is packaged in ***javafx.collections***). The version we will use is shown here:

***static <E> ObservableList<E> observableArrayList(E ... elements)***

* In this case, E specifies the ***type of elements***, which are passed via elements.
* Although ListView provides a default size, sometimes you will want to set the preferred height and/or width to best match your needs. One way to do this is to call the ***setPrefHeight()*** and ***setPrefWidth()*** methods, shown here: ***final void setPrefHeight(double height)***

***final void setPrefWidth(double width)***

* To set both dimensions at same time use, ***setPrefSize()***: ***void setPrefSize(double width, double height)***
* There are two ways to use a ListView. First, ignore ***events generated*** by the list and simply obtain the selection in the list when ***program needs it***.

Second, ***monitor the list for changes*** by registering a change listener. It respond each time the user changes a selection in the list. Second approach used here.

* A ***change listener*** is supported by the ChangeListener interface, which is packaged in javafx.beans.value. The ***ChangeListener interface*** defines only one method, called ***changed()***. It is shown here: ***void changed(ObservableValue<? extends T> changed, T oldVal, T newVal)***
* In this case, changed is the instance of ***ObservableValue<T>*** which encapsulates an object that can be ***watched for changes***. The oldVal and newVal parameters pass the previous value and the new value, respectively. Thus, in this case, newVal holds a *reference to the list item* that has just been selected.
* To listen for change events, you must first obtain the selection model used by the ListView. This is done by calling ***getSelectionModel()*** on the list. It is shown here: ***final MultipleSelectionModel<T> getSelectionModel()***
* It returns a reference to the model. MulitpleSelectionModel is a class that defines the model used for multiple selections, and it ***inherits*** SelectionModel. However, *multiple selections* are allowed in a ListView only if *multiple-selection mode* is turned on.
* Using the model returned by ***getSelectionModel()***, a reference to the selected item property is obtained that defines what takes place when an element in the list is selected. By calling ***selectedItemProperty()***: ***final ReadOnlyObjectProperty<T> selectedItemProperty()***
* You will add the change listener to this property by using the ***addListener()*** method on the returned property. The ***addListener()*** method is shown here: ***void addListener(ChangeListener<? super T> listener)*** In this case, T specifies the type of the property.
* Example 15: Following creates a list view that displays a list of computer types, allowing the user to select one. When one is chosen, the selection is displayed.

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*; import javafx.scene.control.\*;***

***import javafx.geometry.\*; import javafx.beans.value.\*; import javafx.collections.\*;***

**public** **class** ListViewDemo **extends** **Application** { **Label** response;

**public static void main(String[] args)** { **launch**(args); }

**public void start**(**Stage** myStage) { myStage.**setTitle**("ListView Demo"); **FlowPane** rootNode = **new** ***FlowPane***(10, 10);

rootNode.**setAlignment**(Pos.CENTER); **Scene** myScene = **new** ***Scene***(rootNode, 200, 120);

myStage.***setScene***(myScene); response = **new** ***Label***("Select Computer Type");

**ObservableList<String>** computerTypes = FXCollections.***observableArrayList***("Smartphone", "Tablet", "Notebook", "Desktop" );

**ListView<String>** lvComputers = **new** ***ListView<String>***(computerTypes);

lvComputers.***setPrefSize***(100, 70); **MultipleSelectionModel<String>** lvSelModel = lvComputers.***getSelectionModel***();

*// Handle change events: Use a change listener to respond to a change of selection within a list view.*

lvSelModel.***selectedItemProperty***().***addListener***( **new ChangeListener<String>**() {

**public void changed**(**ObservableValue<? extends String>** changed, **String** oldVal, **String** newVal) {response.***setText***("Computer selected is " + newVal); } });

rootNode.***getChildren***().***addAll***(lvComputers, response); *// Add the label and list view to the scene graph.*

myStage.***show***(); } */\* start() ends\*/* }

* ObservableList is created by: ***ObservableList<String> computerTypes = FXCollections.observableArrayList("Smartphone", "Tablet", "Notebook", "Desktop" );***
* It uses the ***observableArrayList()*** to create a list of strings. Then, the ObservableList is used to initialize a ListView, as:

***ListView<String> lvComputers = new ListView<String>(computerTypes);***

* Notice how the selection model is obtained for lvComputers: ***MultipleSelectionModel<String> lvSelModel = lvComputers.getSelectionModel();***
* ListView uses MultipleSelectionModel, even when only a single selection is allowed. The ***selectedItemProperty()*** is then called on the model and a change listener is registered, as shown here:

lvSelModel.***selectedItemProperty***().***addListener***( **new ChangeListener<String>**() {

**public void** **changed**(**ObservableValue<? extends String>** **changed**, **String** oldVal, **String** newVal) {

response.***setText***("Computer selected is " + newVal); } });

* The ***same basic mechanism*** used to listen for and handle change events can be applied to any control that generates change events.
* Enable multiple selections in a ListView: To allow *more than one item* to be selected in ListView, you must explicitly request it. To do so, you must set the selection mode by calling ***setSelectionMode( )*** on the ListView model. It is shown here: ***final void setSelectionMode(SelectionMode mode)***
* In this case, mode must be either SelectionMode.MULTIPLE or SelectionMode.SINGLE. To enable multiple selections, use **SelectionMode.MULTIPLE**.
* One way to get a list of the selected items is to call **getSelectedItems()** on the selection model. It is: **ObservableList<T> getSelectedItems()**
* It returns an ObservableList of the items. You could then cycle through the returned list using a for-each for.
* TextField: JavaFX includes several text-based controls. TextField allows one line of text to be entered. It is useful for obtaining names, ID strings, addresses etc.
* Like all JavaFX text controls, TextField inherits TextInputControl. TextField defines two constructors. The first is the default constructor, which creates an ***empty text field*** that has the ***default size***. The second lets you specify the initial contents of the field. Here, we will use the default constructor.
* To specify size, call ***setPrefColumnCount()***: ***final void setPrefColumnCount(int columns)*** columns is used to determine size.
* You can set the text in a text field by calling ***setText()***. You can obtain the current text by calling ***getText()***. In addition to these fundamental operations, TextField supports several other capabilities, such as cut, paste, and append. You can also select a portion of the text under program control.
* To set a prompting message inside text field when user attempts to use a blank field, call **setPromptText()**: **final void setPromptText(String str)**
* In this case, str is the string displayed in the text field when no text has been entered. It is displayed using low-intensity (such as a gray tone).
* When the user presses enter while inside a TextField, an action ***event is generated***.
* Example 16: Following creates a text field that requests a name. When the user presses enter while the text field has input focus, or presses the Get Name button, the string is obtained and displayed. Notice that a *prompting message* is also included.

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*;***

***import javafx.scene.control.\*; import javafx.event.\*; import javafx.geometry.\*;***

|  |  |
| --- | --- |
| **public class** TextFieldDemo **extends Application** { **TextField** tf;  **Label** response;  **public static void main**(**String[]** args) { **launch**(args); }  **public void start**(**Stage** myStage) {  myStage.**setTitle**("Demonstrate a TextField");  **FlowPane** rootNode = **new** ***FlowPane***(10, 10);  rootNode.***setAlignment***(Pos.CENTER);  **Scene** myScene = **new** ***Scene***(rootNode, 230, 140);  myStage.***setScene***(myScene);  response = **new** ***Label***("Enter Name: ");  **Button** btnGetText = **new** ***Button***("Get Name");  tf = ***new*** ***TextField***(); *// Create a text field.*  tf.***setPromptText***("Enter a name."); *// Set the prompt.*  tf.***setPrefColumnCount***(15); *// Set preferred column count.* | */\* Use a* ***LE*** *to handle action events for the text field..\*/*  tf.***setOnAction***( (ae) ***->*** response.***setText***("*Enter\_prs. Name:* " + tf.***getText***()) );  /\* Use a LE to get text from the text field when the button is pressed. \*/  btnGetText.***setOnAction***((ae) **->** response.***setText***("*Button\_psh. Name:* " + tf.***getText***()) );  **Separator** separator = **new** ***Separator***(); *// Use a separator to better organize the layout.*  separator.***setPrefWidth***(180);  *// Add controls to the scene graph.*  rootNode.***getChildren***().***addAll***(tf, btnGetText, separator, response);  myStage.***show***(); *// Show the stage and its scene.*  }}   * In the program, notice that LEs are used as event handlers. Each *handler* consists of a *single method* call. This makes them perfect candidates for *LEs*. |

* Other text controls supported by JavaFX: Other text controls include TextArea, which supports multiline text, and PasswordField, which can be used to input passwords. You might also find HTMLEditor helpful.

**11.17 Effects and Transforms**

Effects: Effects are supported by the abstract Effect class and its concrete subclasses, which are packaged in javafx.scene.effect. Using these effects, you can customize the way a node in a scene graph looks. Several built-in effects are:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***Bloom*** | ***BoxBlur*** | ***DropShadow*** | ***Glow*** | ***InnerShadow*** | ***Lighting*** | ***Reflection*** |
| Increases the brightness of the brighter parts of a node | Blurs a node. | Displays a shadow that appears behind the node. | Produces a glowing effect | Displays a shadow inside a node | Creates the shadow effects of a light source | Displays a reflection |

* To set an ***effect on a node***, call ***setEffect()***, which is defined by Node: ***final void setEffect(Effect effect)***
* effect is the effect that will be applied. To specify no effect, pass null. Thus, to add an effect to a node, first create an instance of that effect and then pass it to ***setEffect()***. Once this has been done, the effect will be used whenever the node is rendered (as long as the effect is supported by the environment).
* BoxBlur ***blurs the*** node on which it is used. It is called BoxBlur because it uses a blurring technique based on adjusting pixels within a rectangular region. The amount of blurring is under your control. To use a blur effect, you must first create a BoxBlur instance. ***BoxBlur*** supplies ***two constructors***. The constructor that we will use:

***BoxBlur(double width, double height, int iterations)***

Here, width and height specify the *size of box* into which a pixel will be blurred. These values must be between 0 and 255, inclusive. The *number of times* that the *blur effect* is applied is specified by iterations, which must be between 0 and 3, inclusive. A *default constructor* available, sets width & height to 5.0 and iterations to 1.

* After a BoxBlur instance has been created, the width and height of the box can be changed by using ***setWidth()*** and ***setHeight()***, shown here:

***final void setWidth(double width)*** and ***final void setHeight(double height)***

* The number of iterations can be changed by calling ***setIterations()***: ***final void setIterations(int iterations)***
* Reflection simulates a ***reflection of the node*** on which it is called. It is particularly useful on text. You can set the opacity of both the top and the bottom of the reflection. You can also set the space between the image and its reflection, and the ***amount reflected***. These can set by the Reflection constructor:

***Reflection(double offset, double fraction, double topOpacity, double bottomOpacity)***

offset sets the distance between the ***bottom of the image*** and its ***reflection***. The ***amount of the reflection*** is specified by fraction and must be between 0 and 1.0. The top and bottom opacity is specified by topOpacity and bottomOpacity, both must be between 0 and 1.0. A ***default constructor*** is also supplied, which sets the offset to 0, the amount to 0.75, the top opacity to 0.5, and the bottom opacity to 0.

* The offset, amount shown, and opacities can be controlled by corresponding methods. The opacities are set using ***setTopOpacity()*** and ***setBottomOpacity()*** : ***final void setTopOpacity(double opacity)*** and  ***final void setBottomOpacity(double opacity)***
* The offset is changed by calling ***setTopOffset()***: ***final void setTopOffset(double offset)***
* The amount of the reflection displayed can be set by calling ***setFraction()***: ***final void setFraction(double amount)***
* Transforms: Transforms are supported by the abstract Transform class, which is ***packaged*** in javafx.scene.transform. Four of its subclasses are Rotate, Scale, Shear, and Translate. It is possible to perform ***more than one transform on a node***. Eg: you could rotate and scale it. Transforms are supported by the Node class.
* One way to add a transform to a node is to add it to the *list of transforms* maintained by the node. This list is obtained by calling ***getTransforms()***, which is defined by Node. It is: ***final ObservableList<Transform> getTransforms()*** It returns a reference to the list of transforms.
* To add a transform, simply add it to this list by calling ***add()***. You can clear the list by calling ***clear()***. Use ***remove()*** to remove a specific element.
* In some cases, you can specify a transform directly by setting one of Node’s properties. For example,
* You can set the rotation angle of a node, with the pivot point being at the center of the node, by calling ***setRotate()***, passing in the desired angle.
* You can set a scale by using ***setScaleX()*** and ***setScaleY()***, and you can translate a node by using ***setTranslateX()*** and ***setTranslateY()***.
* Rotate rotates a node through a specified angle around a specified point. These values can be set when a Rotate instance is created. Rotate constructor:

***Rotate(double angle, double x, double y)***

* angle specifies the ***number of degrees to rotate***. The ***center of rotation***, called the pivot point, is specified by x and y. Default constructor is available.
* use the ***setAngle()***, ***setPivotX()***, and ***setPivotY()*** methods to set the rotation values after a ***Rotate*** object has been created, shown here:

|  |  |  |
| --- | --- | --- |
| ***final void setAngle(double angle)*** | ***final void setPivotX(double x)*** | ***final void setPivotY(double y)*** |

* angle specifies the number of degrees to rotate and the center of rotation is specified by x and y.
* Scale scales a node as specified by a scale factor. Thus, it changes a node’s size. Scale defines several constructors. Here is the one:

***Scale(double widthFactor, double heightFactor)***

* widthFactor specifies the scaling factor applied to the *node’s width*, and heightFactor specifies the scaling factor applied to the *node’s height*.
* These factors can be changed after a Scale instance has been created by using ***setX()*** and ***setY()***, shown here:

***final void setX(double widthFactor)*** and ***final void setY(double heightFactor)***

* Example 17: The following program demonstrates the use of effects and transforms. It does so by creating three buttons and a label. The buttons are called Rotate, Scale, and Blur. Each time one of these buttons is pressed, the corresponding effect or transform is applied to the button.

***import javafx.application.\*; import javafx.scene.\*; import javafx.stage.\*; import javafx.scene.layout.\*; import javafx.scene.control.\*; import javafx.event.\*;***

***import javafx.geometry.\*; import javafx.scene.transform.\*; import javafx.scene.effect.\*; import javafx.scene.paint.\*;***

**public class** EffectsAndTransformsDemo **extends Application** { **double** angle = 0.0; **double** scaleFactor = 0.4; **double** blurVal = 1.0;

|  |  |
| --- | --- |
| *// Create initial effects and transforms.*  **Reflection** reflection = new Reflection();  **BoxBlur** blur = **new** BoxBlur(1.0, 1.0, 1);  **Rotate** rotate = **new** Rotate();  **Scale** scale = **new** Scale(scaleFactor, scaleFactor);  *// Create push buttons.*  **Button** btnRotate = **new** Button("Rotate");  **Button** btnBlur = **new** Button("Blur off");  **Button** btnScale = **new** Button("Scale");  **Label** reflect = **new** Label("Reflection Adds Visual ");  **public static void main(String[] args)** { launch(args); } | **public void start**(**Stage** myStage) { myStage.**setTitle**("Effects and Transforms Demo");  **FlowPane** rootNode = ***new*** FlowPane(20, 20); rootNode.***setAlignment***(Pos.CENTER);  **Scene** myScene = ***new*** Scene(rootNode, 300, 120); myStage.***setScene***(myScene);  btnRotate.***getTransforms***().***add***(rotate); *// Add rotation to the transform list for the Rotate button.*  btnScale.***getTransforms***().***add***(scale); *// Add scaling to the transform list for the Scale button.*  *// Set the reflection effect on the reflection label.*  reflection.***setTopOpacity***(0.7); reflection.***setBottomOpacity***(0.3); reflect.***setEffect***(reflection);  *// Handle the action events for the Rotate button.*  btnRotate.**setOnAction**(**new EventHandler<ActionEvent>**() { **public void handle**(***ActionEvent*** ae) {  */\* rotated 30 degrees per ckick \*/* angle += 15.0; rotate.***setAngle***(angle);  rotate.***setPivotX***(btnRotate.***getWidth***()/2); rotate.***setPivotY***(btnRotate.***getHeight***()/2); } }); |
| *// Handle the action events for the Scale button.*  btnScale.**setOnAction**(**new EventHandler<ActionEvent>**() { **public void handle**(***ActionEvent*** ae) { scaleFactor += 0.1; */\* scale is changed. Per click \*/*  **if**(scaleFactor > 2.0) scaleFactor = 0.4; scale.**setX**(scaleFactor); scale.**setY**(scaleFactor); } });  btnBlur.**setOnAction**(**new EventHandler<ActionEvent>**() { **public void handle**(***ActionEvent*** ae) { *// Handle the action events for the Blur button.*  /\* blur changed per click \*/ **if**(blurVal == 10.0) { blurVal = 1.0; btnBlur.***setEffect***(null); btnBlur.***setText***("Blur off"); }  **else** { blurVal++; btnBlur.***setEffect***(blur); btnBlur.***setText***("Blur on"); }  blur.***setWidth***(blurVal); blur.***setHeight***(blurVal); } });  rootNode.**getChildren**().addAll(btnRotate, btnScale, btnBlur, reflect); *// Add the label and buttons to the scene graph.*  myStage.**show**(); */\*Show the stage and its scene.\*/* } } | |

* Text node: Text is a class packaged in javafx.scene.text. It creates a node that consists of text. Because it is a node, the text can be easily manipulated as a unit and various effects and transforms can be applied.

What Next?

|  |  |
| --- | --- |
| Learn about Java, its libraries, and its subsystems, but you now have a solid foundation upon which you can build your knowledge and expertise. Here are a few of the topics that you will want to learn more about: | |
| 1. JavaFX and Swing 2. Event handling. 3. Java’s networking classes. 4. Native methods. | 1. Java’s utility classes, especially its Collections Framework, which simplifies a number of common programming tasks. 2. The Concurrent API, which offers detailed control over high-performance multithreaded applications. 3. Java Beans, which supports the creation of software components in Java. 4. Servlets for writing high-powered web applications. Servlets are to the server what applets are to the browser. |

* Continue to advance in your knowledge of Java. A good way to start is by examining Java’s core packages, such as java.lang, java.util, and java.net. Write sample programs that demonstrate their various classes and interfaces. In general, the best way to become a great Java programmer is to write lots of code. *[Luhu\_Saher: Sept-14-2020]*