CHAPTER 15

Creating Graphical User Interfaces

Chapter Learning Objectives

- To make graphical user interfaces out of components such as windows, text fields, buttons, and scrolling lists.
- To use callbacks to handle user interface events.
- To use trees for conceptualizing user interface structure.

15.1 WHERE DO GRAPHICAL USER INTERFACES COME FROM?

The first computers were incredibly painful and tedious to work with. You "programmed" them by literally rewiring them. There wasn't a screen, so you couldn't have a "graphical" user interface at all. Our common experience of using a keyboard to interact with the computer didn't come until much later.

In the late 1960's, the dominant mode of interaction with the computer was through *punched cards*. Using a keyboard on a card punch machine, you prepared your instructions for the computer on pieces of cardboard that were then ordered in the right sequence and loaded into the computer. Heaven help the person who dropped her stack of cards and had to re-order hundreds of cards of a large program! The output from the computer back to the programmer was typically large piles of paper printouts, though was some starting work with computer music and computer graphics (on specialized, expensive monitors).

In the 1960's and 1970's, computer scientists began envisioning a new role for the computer in its interaction with humans. Douglas Engelbart had the idea that a computer might *augment* human intelligence, by reminding us of things, helping us to visualize ideas, and helping us collaborate, even with audio and video conferencing (that he demonstrated for the first time in 1968 and had features that even tools like NetMeeting don't have today!). Those were really radical ideas. Alan Kay went a step further and suggested the idea of *personal computers* that you might interact with to learn through creating and exploring multimedia–a goal for the computer that he called a *Dynabook*. That was an amazing vision given where human-computer interaction was then!

It was hard to imagine interacting with a computer "creatively" in multimedia when all interaction took place through cardboard cards. In the early 1970's, Kay and his team (e.g., Adele Goldberg, Dan Ingalls, Ted Kaehler, and others) at the Xerox PARC (Palo Alto Research Center) set out to invent the user interface for the Dynabook. Engelbart's group had already invented the *mouse*. Others had come up with the idea of windows, but Kay's group made the analogy of a window being like a piece of paper that could overlap on your *desktop*. They invented *menus* that would pop-up then go away. In sum, they invented the *WIMP interface* (overlapping Windows, Icons, Menus, and mouse Pointer), also called the *desktop interface* (because the interface was meant to resemble a physical working desk). That's where the *graphical user interface* (also known as a *GUI* and pronounced "gooey") as we know it today was born.

15.2 CREATING A BASIC GRAPHICAL USER INTERFACE

In JES, we have the advantage of programming in Jython (as opposed to Python) and its special relationship with Java. Java has a very nice user interface set of objects and methods called *Swing*. We can use all of Swing from within JES without doing much special at all.

We're going to build our first GUI from the Command Area. It won't do much, but it will help us see the basic components and how to use them. The problem we're going to work toward solving is having a new way to browse files. Rather than always using a `pickAFile` file dialog, we'll have a window which will let us explore the contents of files–viewing pictures in JPEG files and listening sounds in WAV files.

The first that we need to do is to make Swing available to us in JES. Like any other set of capabilities that go beyond the base language, we need to use an `import` statement.

```
>>> import javax.swing as swing
```

The name `javax.swing` is the formal name that Java uses for identifying the Swing user interface toolkit. That's long to type each time we use it in dot notation, so we ask Jython to let us use an *alias* for it: `swing`.

Now we can create our first window. A window in Swing is called a `JFrame`. We create an instance of `JFrame` just as we would create any instance of a class in Jython.

```
>>> win = swing.JFrame("File Contents Viewer",size=(200,200))
```

Let's talk about what's in this function call.

- The first part is the title of the window: `"File Contents Viewer"`.
- The next part specifies the default size of the window. We'll be able to make it bigger later, but 200 pixels wide by 200 pixels high is how we'll first see the window.

  We're using a different kind of function call here that we haven't used up until now. Rather than simply associating input values with input variables

by *position* (e.g., first value goes to first input variable), it's possible to do it by name explicitly. Here, we're setting the input variable `size` seemingly by assignment in the function call. User interface tools tend to have many options, so making the input variable settings explicit makes the code a bit more readable.

We can now make our window become *visible* by explicitly setting `visible` to true (Figure 15.1.

```
>>> win.visible = 1
```

FIGURE 15.1: Opening our first JFrame (window)

While it's nice to get a window up on the screen, it is not particularly useful since it doesn't *do* anything yet. There's nothing in the window! It will resize if you drag a corner or a side, and it will close (but don't close it yet). Let's next put something in our window.

Let's add a *text field* to our window. This will allow us to display text. The class we use is `JTextField`.

```
>>> field=swing.JTextField(preferredSize=(200,20))
```

In this example, we're creating a `JTextField` with a default size of 200 pixels across and 20 pixels top-to-bottom. We're naming it `field`. Let's put some text into this `field`. A `JTextField` instance has an instance variable named `text`. When we set that instance variable, we're setting the contents of the field.

```
>>> field.text = "Welcome to Swing!"
```

Great–we now have an *invisible* text field. Not the most useful thing in the world. To make it visible, we need to put it in a window–we can't have a text field floating in space. But in order to do that, we need to say something about how user interfaces are constructed in memory.

User interfaces are constructed as a *tree*. There are components inside of components. Our `win` window (instance of `JFrame`) actually has a lot of components

to it, such as the title bar, the close box, and so on. The big (currently empty) part where we would expect to see pieces like text fields and buttons is called the *content pane*. In order to get the text `field` into our window, we need to add it to the content pane–we call that *composing* the text field within the window. When we do that, Jython responds with all the details of what's inside the `field`–which is a *big* long list.

```
>>> win.contentPane.add(field)
javax.swing.JTextField[, 0, 0, 0x0, invalid,
layout=javax.swing.plaf.basic.BasicTextUI$UpdateHandler,
alignmentX=null, alignmentY=null,
border=javax.swing.plaf.BorderUIResource$CompoundBorderUIResource@518924,
flags=296, maximumSize=, minimumSize=,
preferredSize=java.awt.Dimension[width=200, height=20],
caretColor=javax.swing.plaf.ColorUIResource[r=0, g=0, b=0],
disabledTextColor=javax.swing.plaf.ColorUIResource[r=153, g=153,
b=153], editable=true,
margin=javax.swing.plaf.InsetsUIResource[top=0, left=0, bottom=0,
right=0], selectedTextColor=javax.swing.plaf.ColorUIResource[r=0,
g=0, b=0], selectionColor=javax.swing.plaf.ColorUIResource[r=204,
g=204, b=255], columns=0, columnWidth=0, command=,
horizontalAlignment=LEADING]
```

Our window is still empty. To see our text field, we have to remind the window to be visible (much as we have to tell a `Picture` to `repaint`). Then we can see our window and our text field (Figure 15.2).

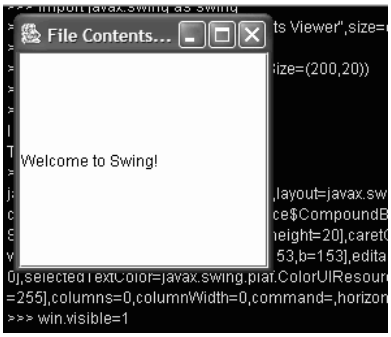


FIGURE 15.2: Our JFrame window with a JTextField text area composed

Now let's add a *button* to our user interface. A button is a graphical object that users can click on to generate events. The class that we use for buttons in Swing is called `JButton`.

```
>>> button=swing.JButton("View Contents",preferredSize=(200,20))
```

In this example, we're creating an instance of `JButton` with a label of "View Contents" and naming it `button`. Again, we're giving it a default size of 200 pixels

across and 20 pixels top-to-bottom. To make it appear in the window, we add it to the content pane (with a *long* response) then remind the window to be visible (Figure 15.3).

```
>>> win.contentPane.add(button)
javax.swing.JButton[, 0, 0, 0x0, invalid,
layout=javax.swing.OverlayLayout, alignmentX=0.0, alignmentY=0.5,
border=javax.swing.plaf.BorderUIResource$CompoundBorderUIResource@9165fa,
flags=296, maximumSize=, minimumSize=,
preferredSize=java.awt.Dimension[width=200, height=20],
defaultIcon=, disabledIcon=, disabledSelectedIcon=,
margin=javax.swing.plaf.InsetsUIResource[top=2, left=14, bottom=2,
right=14], paintBorder=true, paintFocus=true, pressedIcon=,
rolloverEnabled=false, rolloverIcon=, rolloverSelectedIcon=,
selectedIcon=, text=View Contents,defaultCapable=true]
>>> win.visible=1
```

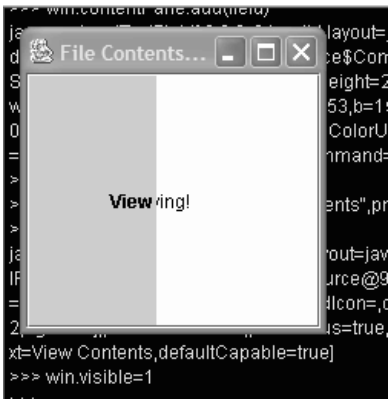


FIGURE 15.3: Our JFrame window with text and button composed

Our user interface at this point is wholly unsatisfactory. Neither our button nor our text area are legible. Clicking on the button makes nothing at all happen. Fixing up these problems is the point of the next section, but let's set the stage a bit here.

Think about the internal structure of our user interface. We have a window (`JFrame`) containing a content pane which itself contains a text area (`JTextField`) and a button (`JButton`). What we have is a *tree* (Figure 15.9). What our tree doesn't show us is how things should be laid out within the components. We will use a *layout manager* to structure how this tree is rendered.

There is a default layout manager, when one isn't specified. The layout isn't very good as a default, but we can try it. We tell the layout manager to take a shot at rendering the user interface tree by telling the window to `pack`.

```
>>> win.pack()
>>> win.visible=1
```

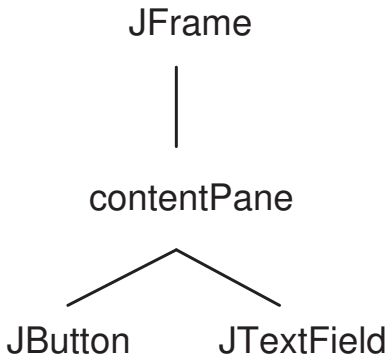


FIGURE 15.4: Representing the internal structure of our user interface as a tree

The initial result of this is a teeny window, but when you expand it, you can see that both pieces are there. They overlap one another, but they're both there (Figure 15.5).

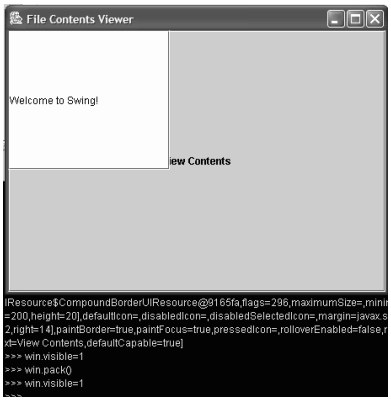


FIGURE 15.5: Our first window, after packing and growing

15.3 CALLBACKS AND LAYOUT MANAGERS

Let's start over again and create a program in the Program Area. User interfaces are much easier to manage with object-oriented programming. You might recall reading about Alan Kay in the earlier chapter when we were discussing object-oriented programming. Modern object-oriented programming and graphical user interfaces were developed at the same time by the same people, so they fit together. Objects make GUIs eaiser.

In order to make our buttons do anything, we have to use *callbacks*. We have to tell them what code to execute (to "call back" to) when a *user interface event* of

interest (like clicking on a button, or tabbing out a field, or double-clicking on an item in a list) occurs. It's easiest to use methods for callbacks because it avoids the issue of *scope*. At the time that the button is clicked, which functions and variables will be available and in the current context? That's a hard question to answer. But if the user interface is associated with an instance of a class, we can expect that all the methods and instance variables associated with that instance will be available.

Here's a recipe that actually builds our file contents viewer, with a slightly extended user interface structure than what we described earlier (Figure 15.6). The idea is to have a `JTextField` that holds a filename in the current media path. When you click the `JButton` "View Contents," the named file is shown (if it's a picture) or played (if it's a sound). The second `JButton` "Set Folder" lets you change the default media folder.

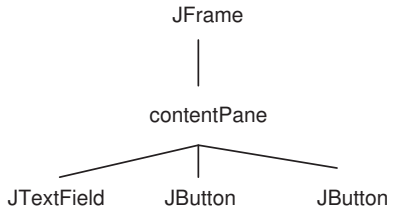
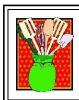


FIGURE 15.6: Representing the internal structure of the File Contents Viewer GUI



Recipe 128: A File Contents Viewer, with text field and buttons

```
import javax.swing as swing
import java

class FileContentsViewer(swing.JFrame):
  def __init__(self):
    swing.JFrame.__init__(self, title="File Contents Viewer", size=(200,200))
    self.contentPane.layout=java.awt.FlowLayout()

    self.field=swing.JTextField(size=(200,60))
    self.field.text="barbara.jpg"
    self.contentPane.add(self.field)

    fileView = swing.JButton("View Contents", size=(65,30),
actionPerformed=self.checkContents)
    self.contentPane.add(fileView)

    setFolder = swing.JButton("Set Folder", size=(65,30),
actionPerformed=self.setFolder)
    self.contentPane.add(setFolder)
```

```
    self.visible = 1

  def checkContents(self,event):
    if self.field.text.endswith(".jpg"):
      pic = makePicture(getMediaPath(self.field.text))
      show(pic)
    if self.field.text.endswith(".wav"):
      snd=makeSound(getMediaPath(self.field.text))
      play(snd)

  def setFolder(self,event):
    setMediaPath()
```

To use this recipe we create an instance of the class, like `fcv = FileContentsViewer()`. We make it visible in `__init__`, so it shows up right away (Figure 15.7).

FIGURE 15.7: File Contents Viewer, with text field and buttons

How it works: This is a fairly complex example, so let's walk through it piece-by-piece.

- We start out importing `swing` and also the general `java` libraries.

  We create the class `FileContentsViewer` as a subclass of `JFrame`. That means that creating an instance of `FileContentsViewer` is actually creating a window. But it makes a little trickier for creating the basic window–how do we pass on to `JFrame` all those settings like title of the window and default size?

  At the start of the `__init__` method, we explicitly call `JFrame`'s `__init__` method. That's where we set the `title` and the `size`. We call it by saying `self.JFrame.__init__` which is saying, "I want to call one of my parent class's `__init__` method, but call it through me, `self`, so that all references to `self` in the method are references to me." That makes sure that *this* window is the one whose `title` and `size` gets set.

- Next comes a piece of code that we haven't seen yet: `self.contentPane.layout = java.awt.FlowLayout()`. `java.awt.FlowLayout` is a class that is one of those layout managers that we mentioned earlier. To make an instance of `FlowLayout` be the layout manager for our window, we tell our `contentPane` to make its `layout` be an instance of `FlowLayout`. `FlowLayout` is not a particularly sophisticated layout manager–it simply puts one thing right after the other. (That is at least better than the default layout manager which allowed things to *overlap* one another.)

  The name is a little complex to understand. The name `java.awt.FlowLayout` means, "In the `java` module, find the `awt` module (A Windowing Toolkit–the original UI tools in Java before Swing was introduced), and use the `FlowLayout` class in there." It is possible to have modules (or *libraries* or *packages*, which are other names for similar ideas) within other modules.

- The lines of code that create the text field are similar to the ones that we saw previously. We create the text file with a default size of 200 pixels wide and 60 pixels high. We put some text into it as a default value. Then we add the text field to the window's `contentPane`. Notice that the `JTextField` is stored in the object's instance variable `field`. That allows methods of `FileContentsViewer` to access the field through `self.field`, and the text in the field as `self.field.text`.

- Our first `JButton` is the one to view the content of whatever filename is typed into the field. Notice that we do *not* store the button in an instance variable! We don't need to. None of the methods in `FileContentsViewer` will need to access the button after it's created. We use the normal variable `fileView` to name the button while we're setting it up, then let the variable disappear when the method ends and its context (scope) ends.

  The button is named "View Contents," and has a size of 65 pixels across and 30 pixels high. (I tried others and decided that I liked that size best–do feel free to try different ones and see what you like.) The important part for us is the `actionPerformed`.

  The `actionPerformed` identifies what method will be used for a *callback*. The expression `actionPerformed=self.checkContents` means "When this button is clicked, call the method on me `checkContents`." The clicking on the button (mouse down and mouse up, with the mouse pointer over the button) is referred to as a *user interface event* or just an *event*. Setting up the `actionPerformed` callback sets up the linkage between the user interface event and the particular method (that appears a bit later in the program.)

- The second `JButton` that we create is to change the media folder, if you want. If you click on it, it called `self.setFolder`.

- Finally, in the `__init__` method, we make the window visible.

- Next comes the method `checkContents` which is used as the callback for the "View Contents" button. *The inputs to a callback method* ***must*** *be* `self,event`. The input `self` has to be there, because this is a method.

The `event` is an event object that gets passed to all callback methods. We really don't have much control over how the callback method gets called. A callback method, by its nature, is one that we don't call from the Command Area–it's only called by Swing itself in response to a user interface event.

The `event` object has details about what kind of user interface event occurred. Was the mouse pressed down (called a *mouse down event*), or released after being pressed down (called a *mouse up event*)? In this example, we know that the method is being called as a callback on a button, but you could imagine the same method being used as a callback for several different buttons or other event-generating GUI elements. Then, the method might need to look at the `event` object to figure out which event it was.

- The `checkContents` method is like much of the code that we saw in the earlier chapters. We check if the filename in the `field.text` ends in ".jpg" or ".wav," then we make the picture or sound and `show` or `play` it.
- The method `setFolder` is the callback for the "Set Folder" method. It takes the same inputs as `checkContents`, because that's how callbacks are set up. But then we just execute `setMediaPath`.

Debugging Tip: Add more fields for debugging
The fact that callbacks are only called from within the user interface makes them difficult to debug. You can't `rint` from within them. If there's a bug in them, no errors are displayed anywhere. If there are any problems with the callback, clicking the button just does nothing. So, you have to be more innovative in your debugging. What I've done sometimes is to insert an extra text field for debugging and "print" things to there as a way of checking values in the running program.

15.4 USING SCROLLING LISTS

As a user interface, the `FileContentsViewer` is a pretty lousy one. We have to remember the file names–if we mistype it, it simply won't work. It only works with the current media folder. What if you want to look at files in several different folders? It would be nice to be able to open several file viewers at once.

We are going to build one more version of the `FileContentsViewer` trying to make it a little more usable. We'll open up this `FileContentsViewer` on a given directory, like this: `fcv = FileContentsViewer("/Users/guzdial/mediasources")` That way, you can open several at once on different directories. Instead of the text field, we'll have a scrolling list of all the files in the input directory so that you can point-and-click on the filename instead of typing it in (Figure 15.8).

Recipe 129: File Contents Viewer with scrolling list

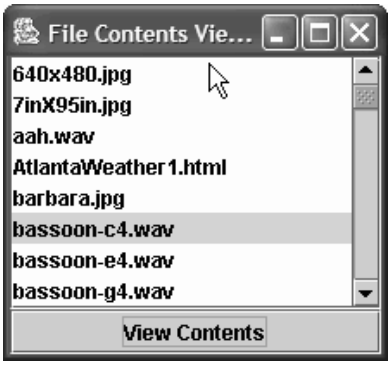


FIGURE 15.8: File Contents Viewer, with a scrolling list

```
import javax.swing as swing
import os
import java

class FileContentsViewer(swing.JFrame):
  def __init__(self,directory):
    swing.JFrame.__init__(self, title="File Contents Viewer", size=(210,250))
    self.contentPane.layout = java.awt.BorderLayout()

    self.currentDirectory = directory
    self.files=swing.JList(os.listdir(self.currentDirectory))
    pane = swing.JScrollPane(self.files)
    self.contentPane.add(pane,java.awt.BorderLayout.CENTER)

    fileView = swing.JButton("View Contents", size=(65,30),
actionPerformed=self.fileView)
    self.contentPane.add(fileView, java.awt.BorderLayout.SOUTH)
    self.pack()
    self.visible = 1

  def fileView(self,event):
   selected=self.files.getSelectedIndices()
   selectedFile = self.files.getModel( ).getElementAt( selected[0])
   selectedFile = self.currentDirectory+ "//"+ selectedFile
   if selectedFile.endswith(".jpg"):
      pic = makePicture(selectedFile)
      show(pic)
   if selectedFile.endswith(".wav"):
      snd = makeSound(selectedFile)
      play(snd)
```

How it works: Most of this `FileContentsViewer` is very similar to the one in the previous section. Most of the difference is in the scrolling pane and list that we use to show the files. The way that we compose this structure is that the *scrolling pane* (`JScrollPane`) is added to the windows `contentPane` and the list (`JList`) is composed inside of the scrolling pane (Figure 15.8). You might want a list that does *not* scroll, so Swing gives you that option.

- We use a different layout manager in this example. The layout manager `java.awt.BorderLayout` is smarter than the previous layout managers we've seen and allows us to tell it where GUI elements should go in the window.

- This version of `FileContentsViewer` takes in a `directory` as input. This directory is saved inside the object in an instance variable `self.directory`. We create the `JList` as `self.files`, using the list of file names from `os.listdir` on the `directory`. We create the scrolling pane `JScrollPane` as `pane` (we don't expect to ever need to access the scrolling pane after it's created), and take `self.files` (the list) as what the scrolling pane scrolls.

- We add the scrolling pane to the window in the center (`java.awt.BorderLayout.CENTER`) of the window. `BorderLayout`'s allow us specify where we want something added.

- We create the "View Contents" button much as we did before, but now we add it in at the bottom of the window (`java.awt.BorderLayout.SOUTH`).

- When the button is pushed, the method `fileView` is used for callback. Now we have to figure out which button is clicked on.

  - We ask the list `self.files` to tell us which things are selected–`getSelectedIndices()`. That gives a list of index numbers (zero-based). We don't really want to deal with many files selected at once, so we'll only use the first one.
  - We get the first index `selected[0]`, and get the element at that index, using the original list of file names that we put in the list. That original list of file names can be accessed through `getModel()` on the list.
  - The filename is no longer in `getMediaPath` as it was in the earlier `FileContentsViewer`. Fortunately, since we saved the input directory, we can create the whole path to the file using `self.currentDictionary + "//" + selectedFile`. Now we can make the media object and view it.

PROGRAMMING SUMMARY

Here are some of the pieces that we talked about using to program GUIs in JES.

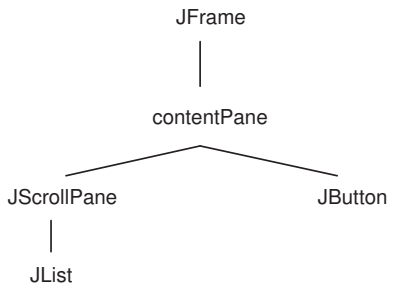


FIGURE 15.9: Representing the `FileContentsViewer` with a scrolling list

JFrame	The class that creates windows in Java Swing. `JFrame` instances have `contentPane`'s that contain their pieces and a `visible` flag. They know how to `pack` themselves using a layout manager.
JTextField	The class that creates text fields (places to show text) in Java Swing.
JButton	The class that creates buttons (things to click to generate action) in Java Swing.
JScrollPane	The class that handles scrolling for things like lists.
JList	The class that handles lists.

PROBLEMS

15.1. In our first `FileContentsViewer`, what do you think would happen if we did *not* provide a default value for the text? We used "barbara.jpg" in the recipe. Try making that the *empty string*, `''` (a string with no characters in it.

15.2. Create a file contents viewer that works by useful name rather than filename. Create a kind of `FileContentsViewer` where you construct a hash table associating a useful name ("Red Caterpillar on Green Leaf") with the actual path and file name. List the names in the scrolling list. When a user chooses a name, open up the corresponding file (to `show` or `play`).

15.3. Build a textual rock-scissors-paper player with a GUI. Create an RSP window with a button and a text area in it. When the user clicks the button, the computer randomly selects "rock," "scissors," or "paper" and displays that word in the text area.

15.4. Build an audio version of the rock-scissors-paper player. Record yourself saying "rock," "scissors," and "paper." Create a user interface with a single button in it. When the user clicks on the button, it randomly chooses one of the three sounds then `play`s it.

TO DIG DEEPER

This is just the briefest overview of how to build user interfaces, and we haven't talked at all about how to do it *well*! It's fairly easy to put up a user interface. It's much harder to put up a user interface that is usable and useful. A good book on

designing good user interfaces is *Human-Computer Interaction* [9]. A good book on the details of building user interfaces (and how toolkits like Swing work) is *Developing User Interfaces* [30].