

13 using swing

Work on Your Swing



Swing is easy. Unless you actually *care* where things end up on the screen. Swing code *looks* easy, but then you compile it, run it, look at it and think, "hey, *that's* not supposed to go *there*." The thing that makes it *easy* to *code* is the thing that makes it *hard* to *control*—the **Layout Manager**. Layout Manager objects control the size and location of the widgets in a Java GUI. They do a ton of work on your behalf, but you won't always like the results. You want two buttons to be the same size, but they aren't. You want the text field to be three inches long, but it's nine. Or one. And *under* the label instead of *next* to it. But with a little work, you can get layout managers to submit to your will. In this chapter, we'll work on our Swing and in addition to layout managers, we'll learn more about widgets. We'll make them, display them (where we choose), and use them in a program. It's not looking too good for Suzy.

Swing components

Component is the more correct term for what we've been calling a *widget*. The *things* you put in a GUI. *The things a user sees and interacts with*. Text fields, buttons, scrollable lists, radio buttons, etc. are all components. In fact, they all extend `javax.swing.JComponent`.

Components can be nested

In Swing, virtually *all* components are capable of holding other components. In other words, *you can stick just about anything into anything else*. But most of the time, you'll add *user interactive* components such as buttons and lists into *background* components such as frames and panels. Although it's *possible* to put, say, a panel inside a button, that's pretty weird, and won't win you any usability awards.

With the exception of `JFrame`, though, the distinction between *interactive* components and *background* components is artificial. A `JPanel`, for example, is usually used as a background for grouping other components, but even a `JPanel` can be interactive. Just as with other components, you can register for the `JPanel`'s events including mouse clicks and keystrokes.

A widget is technically a Swing Component. Almost every thing you can stick in a GUI extends from `javax.swing.JComponent`.

Four steps to making a GUI (review)

- Make a window (a `JFrame`)
`JFrame frame = new JFrame();`
- Make a component (button, text field, etc.)
`JButton button = new JButton("click me");`
- Add the component to the frame
`frame.getContentPane().add(BorderLayout.EAST, button);`
- Display it (give it a size and make it visible)
`frame.setSize(300,300);`
`frame.setVisible(true);`

Put interactive components:



Into background components:



Layout Managers

A layout manager is a Java object associated with a particular component, almost always a *background* component. The layout manager controls the components contained *within* the component the layout manager is associated with. In other words, if a frame holds a panel, and the panel holds a button, the panel's layout manager controls the size and placement of the button, while the frame's layout manager controls the size and placement of the panel. The button, on the other hand, doesn't need a layout manager because the button isn't holding other components.

If a panel holds five things, even if those five things each have their own layout managers, the size and location of the five things in the panel are all controlled by the panel's layout manager. If those five things, in turn, hold *other* things, then those *other* things are placed according to the layout manager of the thing holding them.

When we say *hold* we really mean *add* as in, a panel *holds* a button because the button was *added* to the panel using something like:

```
myPanel.add(button);
```

Layout managers come in several flavors, and each background component can have its own layout manager. Layout managers have their own policies to follow when building a layout. For example, one layout manager might insist that all components in a panel must be the same size, arranged in a grid, while another layout manager might let each component choose its own size, but stack them vertically. Here's an example of nested layouts:

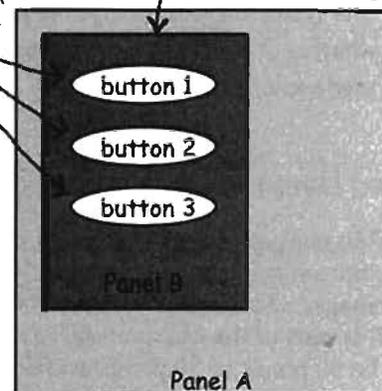
```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```

As a layout manager, I'm in charge of the size and placement of your components. In this GUI, I'm the one who decided how big these buttons should be, and where they are relative to each other and the frame.



Panel B's layout manager controls the size and placement of the three buttons

Panel A's layout manager controls the size and placement of Panel B.



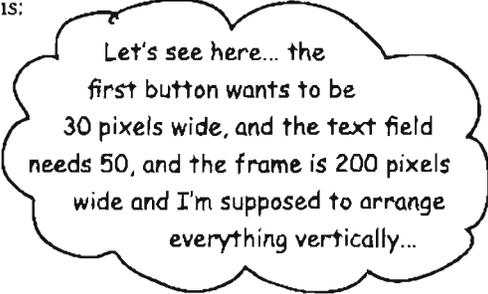
Panel A's layout manager has *NOTHING* to say about the three buttons. The hierarchy of control is only one level—Panel A's layout manager controls only the things added directly to Panel A, and does not control anything nested within those added components.

How does the layout manager decide?

Different layout managers have different policies for arranging components (like, arrange in a grid, make them all the same size, stack them vertically, etc.) but the components being layed out do get at least *some* small say in the matter. In general, the process of laying out a background component looks something like this:

A layout scenario:

- ① Make a panel and add three buttons to it.
- ② The panel's layout manager asks each button how big that button prefers to be.
- ③ The panel's layout manager uses its layout policies to decide whether it should respect all, part, or none of the buttons' preferences.
- ④ Add the panel to a frame.
- ⑤ The frame's layout manager asks the panel how big the panel prefers to be.
- ⑥ The frame's layout manager uses its layout policies to decide whether it should respect all, part, or none of the panel's preferences.



Let's see here... the first button wants to be 30 pixels wide, and the text field needs 50, and the frame is 200 pixels wide and I'm supposed to arrange everything vertically...



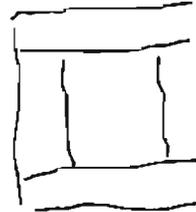
Different layout managers have different policies

Some layout managers respect the size the component wants to be. If the button wants to be 30 pixels by 50 pixels, that's what the layout manager allocates for that button. Other layout managers respect only part of the component's preferred size. If the button wants to be 30 pixels by 50 pixels, it'll be 30 pixels by however wide the button's background *panel* is. Still other layout managers respect the preference of only the *largest* of the components being layed out, and the rest of the components in that panel are all made that same size. In some cases, the work of the layout manager can get very complex, but most of the time you can figure out what the layout manager will probably do, once you get to know that layout manager's policies.

The Big Three layout managers: border, flow, and box.

BorderLayout

A BorderLayout manager divides a background component into five regions. You can add only one component per region to a background controlled by a BorderLayout manager. Components laid out by this manager usually don't get to have their preferred size. BorderLayout is the default layout manager for a frame!



one component
per region

FlowLayout

A FlowLayout manager acts kind of like a word processor, except with components instead of words. Each component is the size it wants to be, and they're laid out left to right in the order that they're added, with "word-wrap" turned on. So when a component won't fit horizontally, it drops to the next "line" in the layout. FlowLayout is the default layout manager for a panel!



components added left
to right, wrapping to a
new line when needed

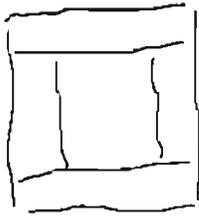
BoxLayout

A BoxLayout manager is like FlowLayout in that each component gets to have its own size, and the components are placed in the order in which they're added. But, unlike FlowLayout, a BoxLayout manager can stack the components vertically (or horizontally, but usually we're just concerned with vertically). It's like a FlowLayout but instead of having automatic 'component wrapping', you can insert a sort of 'component return key' and force the components to start a new line.



components added top
to bottom, one per 'line'

border layout



BorderLayout cares about five regions: east, west, north, south, and center

Let's add a button to the east region:

```
import javax.swing.*;
import java.awt.*; ← BorderLayout is in java.awt package

public class Button1 {

    public static void main (String[] args) {
        Button1 gui = new Button1();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JButton button = new JButton("click me");
        frame.getContentPane().add(BorderLayout.EAST, button); ← specify the region
        frame.setSize(200,200);
        frame.setVisible(true);
    }
}
```

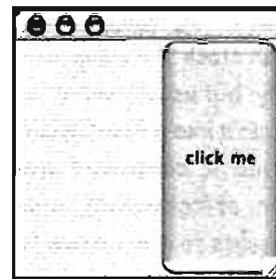


Brain Barbell

How did the BorderLayout manager come up with this size for the button?

What are the factors the layout manager has to consider?

Why isn't it wider or taller?



Watch what happens when we give the button more characters...

```
public void go() {  
    JFrame frame = new JFrame();  
    JButton button = new JButton("click like you mean it");  
    frame.getContentPane().add(BorderLayout.EAST, button);  
    frame.setSize(200,200);  
    frame.setVisible(true);  
}
```

We changed only the text on the button

First, I ask the button for its preferred size.



I have a lot of words now, so I'd prefer to be 60 pixels wide and 25 pixels tall.



Since it's in the east region of a border layout, I'll respect its preferred width. But I don't care how tall it wants to be: it's gonna be as tall as the frame, because that's my policy.



The button gets its preferred width, but not height

Next time I'm goin' with flow layout. Then I get EVERYTHING I want.



border layout

Let's try a button in the NORTH region

```
public void go() {  
    JFrame frame = new JFrame();  
    JButton button = new JButton("There is no spoon...");  
    frame.getContentPane().add(BorderLayout.NORTH, button);  
    frame.setSize(200,200);  
    frame.setVisible(true);  
}
```



← The button is as tall as it wants to be, but as wide as the frame.

Now let's make the button ask to be taller

How do we do that? The button is already as wide as it can ever be—as wide as the frame. But we can try to make it taller by giving it a bigger font.

```
public void go() {  
    JFrame frame = new JFrame();  
    JButton button = new JButton("Click This!");  
    Font bigFont = new Font("serif", Font.BOLD, 28);  
    button.setFont(bigFont);  
    frame.getContentPane().add(BorderLayout.NORTH, button);  
    frame.setSize(200,200);  
    frame.setVisible(true);  
}
```



← A bigger font will force the frame to allocate more space for the button's height

← The width stays the same, but now the button is taller. The north region stretched to accommodate the button's new preferred height.

I think I'm getting it... if I'm in **east** or **west**, I get my preferred width but the height is up to the layout manager. And if I'm in **north** or **south**, it's just the opposite—I get my preferred height, but not width.



But what happens
in the center region?

The center region gets whatever's left!

(except in one special case we'll look at later)

```
public void go() {
    JFrame frame = new JFrame();

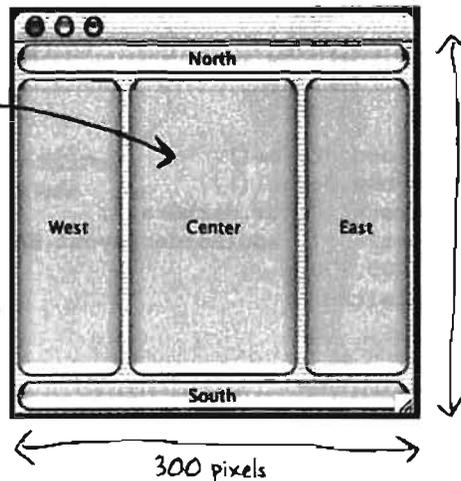
    JButton east = new JButton("East");
    JButton west = new JButton("West");
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton center = new JButton("Center");

    frame.getContentPane().add(BorderLayout.EAST, east);
    frame.getContentPane().add(BorderLayout.WEST, west);
    frame.getContentPane().add(BorderLayout.NORTH, north);
    frame.getContentPane().add(BorderLayout.SOUTH, south);
    frame.getContentPane().add(BorderLayout.CENTER, center);

    frame.setSize(300, 300);
    frame.setVisible(true);
}
```

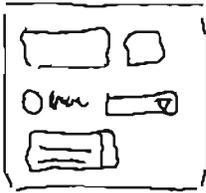
Components in the center get whatever space is left over, based on the frame dimensions (300 x 300 in this code).

Components in the east and west get their preferred width.
Components in the north and south get their preferred height



When you put something in the north or south, it goes all the way across the frame, so the things in the east and west won't be as tall as they would be if the north and south regions were empty.

flow layout



FlowLayout cares about the flow of the components:

left to right, top to bottom, in the order they were added.

Let's add a panel to the east region:

A JPanel's layout manager is FlowLayout, by default. When we add a panel to a frame, the size and placement of the panel is still under the BorderLayout manager's control. But anything *inside* the panel (in other words, components added to the panel by calling `panel.add(aComponent)`) are under the panel's FlowLayout manager's control. We'll start by putting an empty panel in the frame's east region, and on the next pages we'll add things to the panel.

The panel doesn't have anything in it, so it doesn't ask for much width in the east region.

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

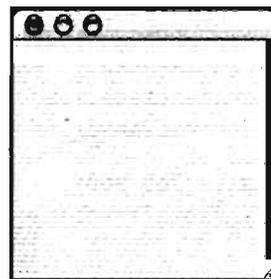
```
public class Panell {
```

```
    public static void main (String[] args) {
        Panell gui = new Panell();
        gui.go();
    }
```

```
    public void go() {
```

```
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();
        panel.setBackground(Color.darkGray);
        frame.getContentPane().add(BorderLayout.EAST, panel);
        frame.setSize(200,200);
        frame.setVisible(true);
```

```
    }
}
```



Make the panel gray so we can see where it is on the frame.

Let's add a button to the panel

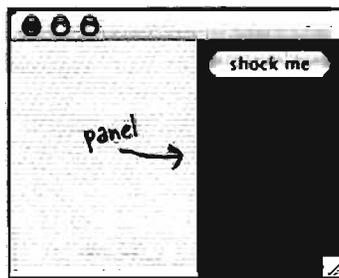
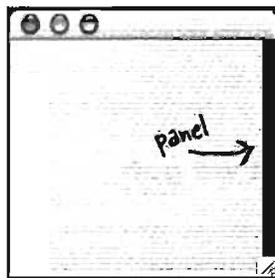
```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    JButton button = new JButton("shock me");

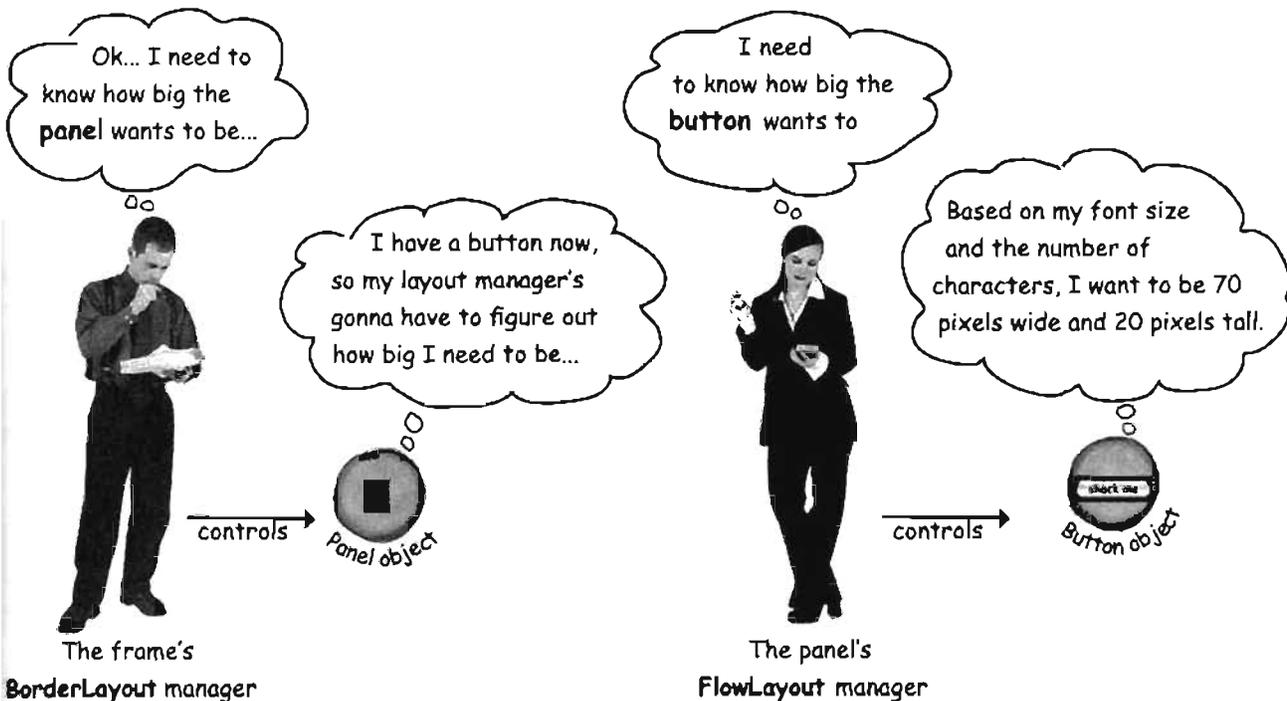
    panel.add(button);
    frame.getContentPane().add(BorderLayout.EAST, panel);

    frame.setSize(250,200);
    frame.setVisible(true);
}
```

Add the button to the panel and add the panel to the frame. The panel's layout manager (flow) controls the button, and the frame's layout manager (border) controls the panel.



The panel expanded!
And the button got its preferred size in both dimensions, because the panel uses flow layout, and the button is part of the panel (not the frame).



flow layout

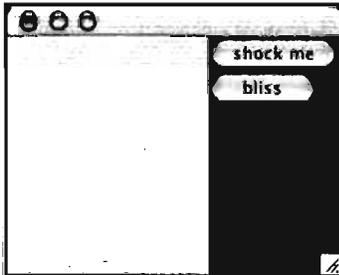
What happens if we add TWO buttons to the panel?

```
public void go() {  
    JFrame frame = new JFrame();  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
  
    JButton button = new JButton("shock me");  
    JButton buttonTwo = new JButton("bliss");  
  
    panel.add(button);  
    panel.add(buttonTwo);  
  
    frame.getContentPane().add(BorderLayout.EAST, panel);  
    frame.setSize(250,200);  
    frame.setVisible(true);  
}
```

← make TWO buttons

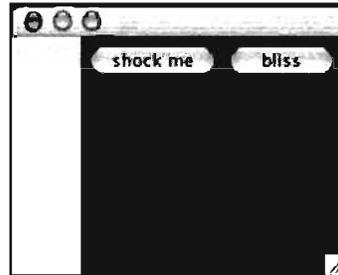
← add BOTH to the panel

what we wanted:



We want the buttons stacked on top of each other

what we got:



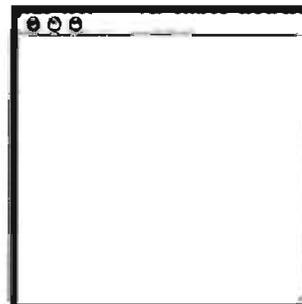
The panel expanded to fit both buttons side by side.

notice that the 'bliss' button is smaller than the 'shock me' button... that's how flow layout works. The button gets just what it needs (and no more).

Sharpen your pencil

If the code above were modified to the code below, what would the GUI look like?

```
JButton button = new JButton("shock me");  
JButton buttonTwo = new JButton("bliss");  
JButton buttonThree = new JButton("huh?");  
panel.add(button);  
panel.add(buttonTwo);  
panel.add(buttonThree);
```



Draw what you think the GUI would look like if you ran the code to the left.

(Then try it!)



BoxLayout to the rescue!

It keeps components stacked, even if there's room to put them side by side.

Unlike FlowLayout, BorderLayout can force a 'new line' to make the components wrap to the next line, even if there's room for them to fit horizontally.

But now you'll have to change the panel's layout manager from the default FlowLayout to BorderLayout.

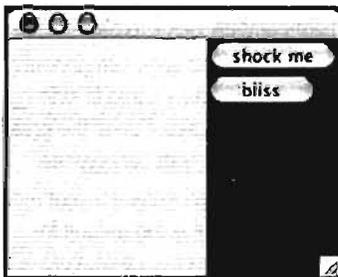
```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    panel.setLayout(new BorderLayout(panel, BorderLayout.Y_AXIS));

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");
    panel.add(button);
    panel.add(buttonTwo);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250, 200);
    frame.setVisible(true);
}
```

Change the layout manager to be a new instance of BorderLayout

The BorderLayout constructor needs to know the component its laying out (i.e., the panel) and which axis to use (we use Y_AXIS for a vertical stack).



Notice how the panel is narrower again, because it doesn't need to fit both buttons horizontally. So the panel told the frame it needed enough room for only the largest button, 'shock me'.

layout managers

there are no Dumb Questions

Q: How come you can't add directly to a frame the way you can to a panel?

A: A JFrame is special because it's where the rubber meets the road in making something appear on the screen. While all your Swing components are pure Java, a JFrame has to connect to the underlying OS in order to access the display. Think of the content pane as a 100% pure Java layer that sits on top of the JFrame. Or think of it as though JFrame is the window frame and the content pane is the... glass. You know, the window *pane*. And you can even *swap* the content pane with your own JPanel, to make your JPanel the frame's content pane, using,

```
myFrame.setContentPane(myPanel);
```

Q: Can I change the layout manager of the frame? What if I want the frame to use flow instead of border?

A: The easiest way to do this is to make a panel, build the GUI the way you want in the panel, and then make that panel the frame's content pane using the code in the previous answer (rather than using the default content pane).

Q: What if I want a different preferred size? Is there a `setSize()` method for components?

A: Yes, there is a `setSize()`, but the layout managers will ignore it. There's a distinction between the *preferred size* of the component and the size *you* want it to be. The preferred size is based on the size the component actually *needs* (the component makes that decision for itself). The layout manager calls the component's `getPreferredSize()` method, and *that* method doesn't *care* if you've previously called `setSize()` on the component.

Q: Can't I just put things where I want them? Can I turn the layout managers off?

A: Yep. On a component by component basis, you can call `setLayout(null)` and then it's up to you to hard-code the exact screen locations and dimensions. In the long run, though, it's almost always easier to use layout managers.



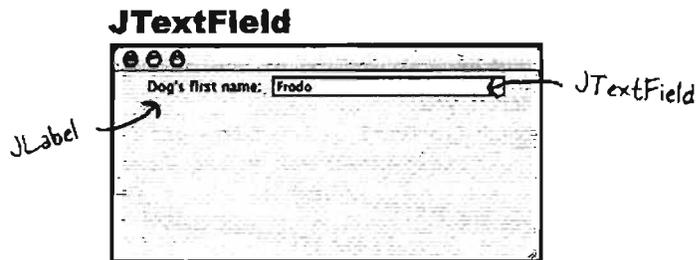
BULLET POINTS

- Layout managers control the size and location of components nested within other components.
- When you add a component to another component (sometimes referred to as a *background* component, but that's not a technical distinction), the added component is controlled by the layout manager of the *background* component.
- A layout manager asks components for their preferred size, before making a decision about the layout. Depending on the layout manager's policies, it might respect all, some, or none of the component's wishes.
- The BorderLayout manager lets you add a component to one of five regions. You must specify the region when you add the component, using the following syntax:

```
add(BorderLayout.EAST, panel);
```
- With BorderLayout, components in the north and south get their preferred height, but not width. Components in the east and west get their preferred width, but not height. The component in the center gets whatever is left over (unless you use `pack()`).
- The `pack()` method is like shrink-wrap for the components; it uses the full preferred size of the center component, then determines the size of the frame using the center as a starting point, building the rest based on what's in the other regions.
- FlowLayout places components left to right, top to bottom, in the order they were added, wrapping to a new line of components only when the components won't fit horizontally.
- FlowLayout gives components their preferred size in both dimensions.
- BorderLayout lets you align components stacked vertically, even if they could fit side-by-side. Like FlowLayout, BorderLayout uses the preferred size of the component in both dimensions.
- BorderLayout is the default layout manager for a frame; FlowLayout is the default for a panel.
- If you want a panel to use something other than flow, you have to call `setLayout()` on the panel.

Playing with Swing components

You've learned the basics of layout managers, so now let's try out a few of the most common components: a text field, scrolling text area, checkbox, and list. We won't show you the whole darn API for each of these, just a few highlights to get you started.



Constructors

```
JTextField field = new JTextField(20);
```

```
JTextField field = new JTextField("Your name");
```

20 means 20 columns, not 20 pixels. This defines the preferred width of the text field.

How to use it

- Get text out of it

```
System.out.println(field.getText());
```

- Put text in it

```
field.setText("whatever");
```

```
field.setText("");
```

This clears the field

- Get an ActionEvent when the user presses return or enter

You can also register for key events if you really want to hear about it every time the user presses a key.

```
field.addActionListener(myActionListener);
```

- Select/Highlight the text in the field

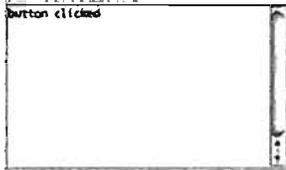
```
field.selectAll();
```

- Put the cursor back in the field (so the user can just start typing)

```
field.requestFocus();
```

text area

JTextArea



Unlike JTextField, JTextArea can have more than one line of text. It takes a little configuration to make one, because it doesn't come out of the box with scroll bars or line wrapping. To make a JTextArea scroll, you have to stick it in a JScrollPane. A JScrollPane is an object that really loves to scroll, and will take care of the text area's scrolling needs.

Constructor

```
JTextArea text = new JTextArea(10,20);
```

10 means 10 lines (sets the preferred height)
20 means 20 columns (sets the preferred width)

How to use it

- Make it have a vertical scrollbar only

```
JScrollPane scroller = new JScrollPane(text);  
text.setLineWrap(true);
```

Turn on line wrapping

```
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);  
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

Tell the scroll pane to use only a vertical scrollbar

```
panel.add(scroller);
```

Important!! You give the text area to the scroll pane (through the scroll pane constructor), then add the scroll pane to the panel. You don't add the text area directly to the panel!

- Replace the text that's in it

```
text.setText("Not all who are lost are wandering");
```

- Append to the text that's in it

```
text.append("button clicked");
```

- Select/Highlight the text in the field

```
text.selectAll();
```

- Put the cursor back in the field (so the user can just start typing)

```
text.requestFocus();
```

JTextArea example

```

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

public class TextAreal implements ActionListener {

    JTextArea text;

    public static void main (String[] args) {
        TextAreal gui = new TextAreal();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();
        JButton button = new JButton("Just Click It");
        button.addActionListener(this);
        text = new JTextArea(10,20);
        text.setLineWrap(true);

        JScrollPane scroller = new JScrollPane(text);
        scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

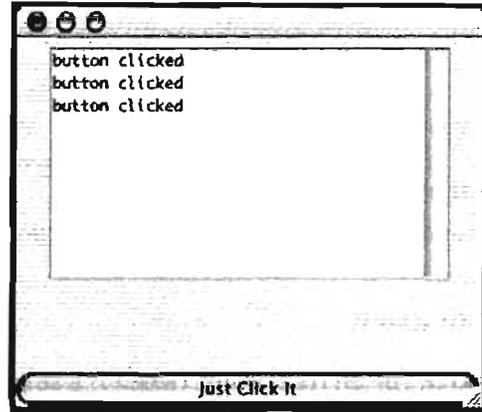
        panel.add(scroller);

        frame.getContentPane().add(BorderLayout.CENTER, panel);
        frame.getContentPane().add(BorderLayout.SOUTH, button);

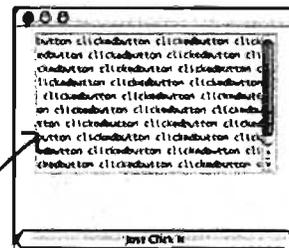
        frame.setSize(350,300);
        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent ev) {
        text.append("button clicked \n ");
    }
}

```



↑
 Insert a new line so the words go on a separate line each time the button is clicked. Otherwise, they'll run together.



check box

JCheckBox



Constructor

```
JCheckBox check = new JCheckBox("Goes to 11");
```

How to use it

- 1 Listen for an item event (when it's selected or deselected)

```
check.addItemListener(this);
```

- 2 Handle the event (and find out whether or not it's selected)

```
public void itemStateChanged(ItemEvent ev) {  
    String onOrOff = "off";  
    if (check.isSelected()) onOrOff = "on";  
    System.out.println("Check box is " + onOrOff);  
}
```

- 3 Select or deselect it in code

```
check.setSelected(true);  
check.setSelected(false);
```

there are no Dumb Questions

Q: Aren't the layout managers just more trouble than they're worth? If I have to go to all this trouble, I might as well just hard-code the size and coordinates for where everything should go.

A: Getting the exact layout you want from a layout manager can be a challenge. But think about what the layout manager is really doing for you. Even the seemingly simple task of figuring out where things should go on the screen can be complex. For example, the layout manager takes care of keeping your components from overlapping one another. In other words, it knows how to manage the spacing between components (and between the edge of the frame). Sure you can do that yourself, but what happens if you want components to be very tightly packed? You might get them placed just right, by hand, but that's only good for your JVM!

Why? Because the components can be slightly different from platform to platform, especially if they use the underlying platform's native 'look and feel'. Subtle things like the bevel of the buttons can be different in such a way that components that line up neatly on one platform suddenly squish together on another.

And we're still not at the really Big Thing that layout managers do. Think about what happens when the user resizes the window! Or your GUI is dynamic, where components come and go. If you had to keep track of re-laying out all the components every time there's a change in the size or contents of a background component...yikes!

JList



JList constructor takes an array of any object type. They don't have to be Strings, but a String representation will appear in the list

Constructor

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
    "epsilon", "zeta", "eta", "theta "};
```

```
list = new JList(listEntries);
```

How to use It

- Make it have a vertical scrollbar

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

This is just like with JTextArea -- you make a JScrollPane (and give it the list), then add the scroll pane (NOT the list) to the panel.

- Set the number of lines to show before scrolling

```
list.setVisibleRowCount(4);
```

- Restrict the user to selecting only ONE thing at a time

```
list.setSelectionMode(ListSelectionMode.SINGLE_SELECTION);
```

- Register for list selection events

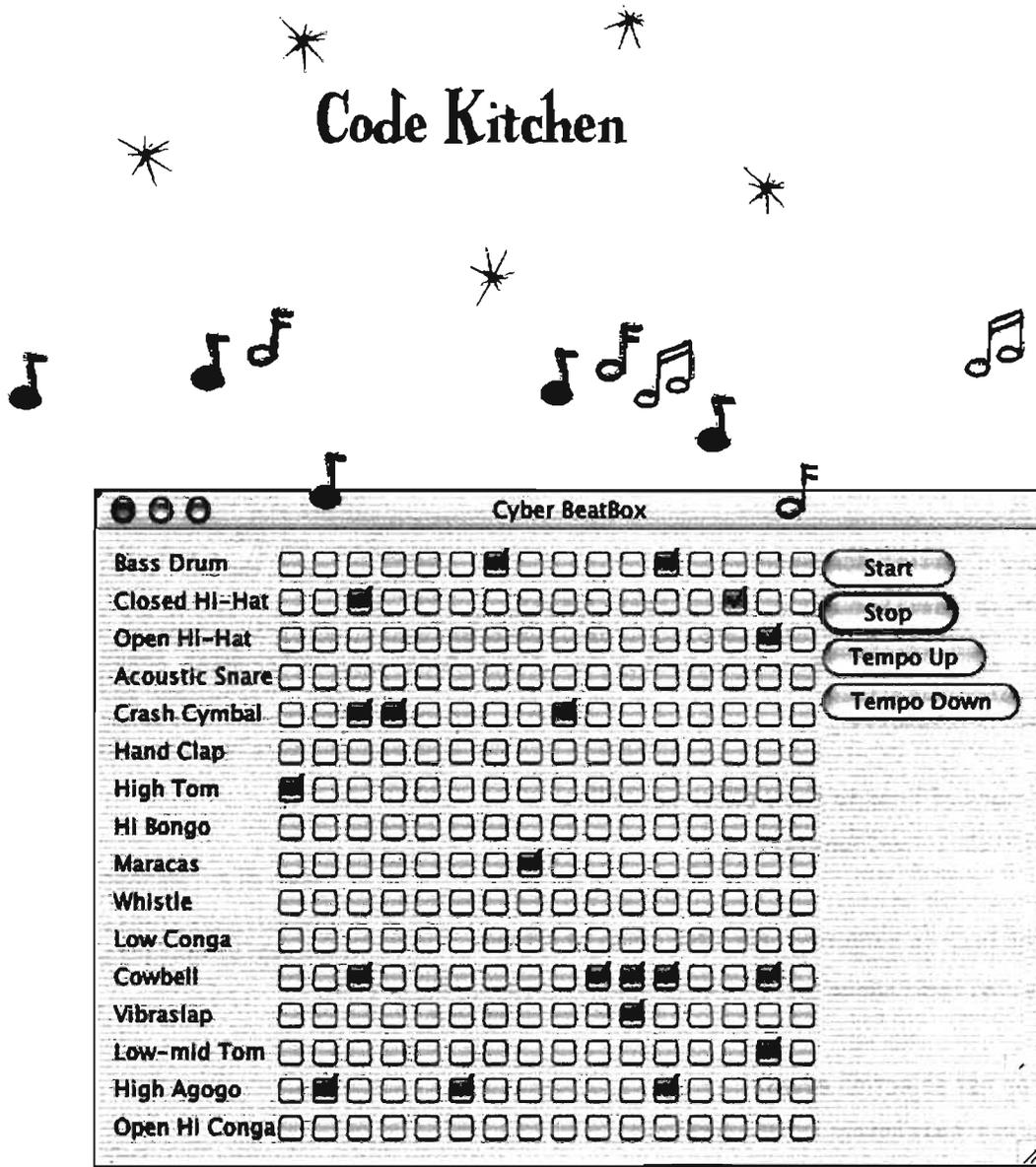
```
list.addListSelectionListener(this);
```

- Handle events (find out which thing in the list was selected)

```
public void valueChanged(ListSelectionEvent lse) {
    if (!lse.getValueIsAdjusting()) {
        String selection = (String) list.getSelectedValue();
        System.out.println(selection);
    }
}
```

You'll get the event TWICE if you don't put in this if test.

getSelectedValue() actually returns an Object. A list isn't limited to only String objects.



This part's optional. We're making the full BeatBox, GUI and all. In the Saving Objects chapter, we'll learn how to save and restore drum patterns. Finally, in the networking chapter (Make a Connection), we'll turn the BeatBox into a working chat client.

Making the BeatBox

This is the full code listing for this version of the BeatBox, with buttons for starting, stopping, and changing the tempo. The code listing is complete, and fully-annotated, but here's the overview:

- Build a GUI that has 256 checkboxes (JCheckBox) that start out unchecked, 16 labels (JLabel) for the instrument names, and four buttons.
- Register an ActionListener for each of the four buttons. We don't need listeners for the individual checkboxes, because we aren't trying to change the pattern sound dynamically (i.e. as soon as the user checks a box). Instead, we wait until the user hits the 'start' button, and then walk through all 256 checkboxes to get their state and make a MIDI track.
- Set-up the MIDI system (you've done this before) including getting a Sequencer, making a Sequence, and creating a track. We are using a sequencer method that's new to Java 5.0, `setLoopCount()`. This method allows you to specify how many times you want a sequence to loop. We're also using the sequence's tempo factor to adjust the tempo up or down, and maintain the new tempo from one iteration of the loop to the next.
- When the user hits 'start', the real action begins. The event-handling method for the 'start' button calls the `buildTrackAndStart()` method. In that method, we walk through all 256 checkboxes (one row at a time, a single instrument across all 16 beats) to get their state, then use the information to build a MIDI track (using the handy `makeEvent()` method we used in the previous chapter). Once the track is built, we start the sequencer, which keeps playing (because we're looping it) until the user hits 'stop'.

BeatBox code

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
```

```
public class BeatBox {
```

```
    JPanel mainPanel;
    ArrayList<JCheckBox> checkBoxList;
    Sequencer sequencer;
    Sequence sequence;
    Track track;
    JFrame theFrame;
```

↖ We store the checkboxes in an ArrayList

```
    String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat",
        "Open Hi-Hat", "Acoustic Snare", "Crash Cymbal", "Hand Clap",
        "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga",
        "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo",
        "Open Hi Conga"};
    int[] instruments = {35, 42, 46, 38, 49, 39, 50, 60, 70, 72, 64, 56, 58, 47, 67, 63};
```

↖ These are the names of the instruments, as a String array, for building the GUI labels (on each row)

```
    public static void main (String[] args) {
        new BeatBox2().buildGUI();
    }
```

↖ These represent the actual drum 'keys'. The drum channel is like a piano, except each 'key' on the piano is a different drum. So the number '35' is the key for the Bass drum, 42 is Closed Hi-Hat, etc.

```
    public void buildGUI() {
        theFrame = new JFrame("Cyber BeatBox");
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        BorderLayout layout = new BorderLayout();
        JPanel background = new JPanel(layout);
        background.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
```

```
        checkBoxList = new ArrayList<JCheckBox>();
        Box buttonBox = new Box(BoxLayout.Y_AXIS);

        JButton start = new JButton("Start");
        start.addActionListener(new MyStartListener());
        buttonBox.add(start);
```

↖ An 'empty border' gives us a margin between the edges of the panel and where the components are placed. Purely aesthetic.

```
        JButton stop = new JButton("Stop");
        stop.addActionListener(new MyStopListener());
        buttonBox.add(stop);
```

```
        JButton upTempo = new JButton("Tempo Up");
        upTempo.addActionListener(new MyUpTempoListener());
        buttonBox.add(upTempo);
```

Nothing special here, just lots of GUI code. You've seen most of it before.

```
        JButton downTempo = new JButton("Tempo Down");
```

```

downTempo.addActionListener(new MyDownTempoListener());
buttonBox.add(downTempo);

Box nameBox = new Box(BoxLayout.Y_AXIS);
for (int i = 0; i < 16; i++) {
    nameBox.add(new Label(instrumentNames[i]));
}

background.add(BorderLayout.EAST, buttonBox);
background.add(BorderLayout.WEST, nameBox);

theFrame.getContentPane().add(background);

GridLayout grid = new GridLayout(16,16);
grid.setVgap(1);
grid.setHgap(2);
mainPanel = new JPanel(grid);
background.add(BorderLayout.CENTER, mainPanel);

for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    checkBoxList.add(c);
    mainPanel.add(c);
} // end loop

setUpMidi();

theFrame.setBounds(50,50,300,300);
theFrame.pack();
theFrame.setVisible(true);
} // close method

public void setUpMidi() {
    try {
        sequencer = MidiSystem.getSequencer();
        sequencer.open();
        sequence = new Sequence(Sequence.PPQ,4);
        track = sequence.createTrack();
        sequencer.setTempoInBPM(120);

    } catch(Exception e) {e.printStackTrace();}
} // close method

```

Still more GUI set-up code.
Nothing remarkable.

Make the checkboxes, set them to 'false' (so they aren't checked) and add them to the ArrayList AND to the GUI panel.

The usual MIDI set-up stuff for getting the Sequencer, the Sequence, and the Track. Again, nothing special.

BeatBox code

This is where it all happens! Where we turn checkbox state into MIDI events, and add them to the Track.

```
public void buildTrackAndStart() {  
    int[] trackList = null;  
  
    sequence.deleteTrack(track);  
    track = sequence.createTrack();
```

We'll make a 16-element array to hold the values for one instrument, across all 16 beats. If the instrument is supposed to play on that beat, the value at that element will be the key. If that instrument is NOT supposed to play on that beat, put in a zero.

} get rid of the old track, make a fresh one.

```
    for (int i = 0; i < 16; i++) {  
        trackList = new int[16];
```

← do this for each of the 16 ROWS (i.e. Bass, Congo, etc.)

```
        int key = instruments[i];
```

← Set the 'key' that represents which instrument this is (Bass, Hi-Hat, etc. The instruments array holds the actual MIDI numbers for each instrument.)

```
        for (int j = 0; j < 16; j++) {
```

← Do this for each of the BEATS for this row

```
            JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));  
            if ( jc.isSelected() ) {  
                trackList[j] = key;  
            } else {  
                trackList[j] = 0;  
            }  
        } // close inner loop
```

} Is the checkbox at this beat selected? If yes, put the key value in this slot in the array (the slot that represents this beat). Otherwise, the instrument is NOT supposed to play at this beat, so set it to zero.

```
        makeTracks(trackList);  
        track.add(makeEvent(176,1,127,0,16));  
    } // close outer
```

← For this instrument, and for all 16 beats, make events and add them to the track.

```
    track.add(makeEvent(192,9,1,0,15));  
    try {
```

← We always want to make sure that there IS an event at beat 16 (it goes 0 to 15). Otherwise, the BeatBox might not go the full 16 beats before it starts over.

```
        sequencer.setSequence(sequence);  
        sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);  
        sequencer.start();  
        sequencer.setTempoInBPM(120);  
    } catch (Exception e) {e.printStackTrace();}  
} // close buildTrackAndStart method
```

← Let's you specify the number of loop iterations, or in this case, continuous looping.

NOW PLAY THE THING!!

```
public class MyStartListener implements ActionListener {  
    public void actionPerformed(ActionEvent a) {  
        buildTrackAndStart();  
    }  
} // close inner class
```

} First of the inner classes, listeners for the buttons. Nothing special here.

```

public class MyStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sequencer.stop();
    }
} // close inner class

public class MyUpTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    }
} // close inner class

public class MyDownTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
} // close inner class

```

The other inner class listeners for the buttons

The Tempo Factor scales the sequencer's tempo by the factor provided. The default is 1.0, so we're adjusting +/- 3% per click.

```

public void makeTracks(int[] list) {
    for (int i = 0; i < 16; i++) {
        int key = list[i];

        if (key != 0) {
            track.add(makeEvent(144,9,key, 100, i));
            track.add(makeEvent(128,9,key, 100, i+1));
        }
    }
}

```

This makes events for one instrument at a time, for all 16 beats. So it might get an int[] for the Bass drum, and each index in the array will hold either the key of that instrument, or a zero. If it's a zero, the instrument isn't supposed to play at that beat. Otherwise, make an event and add it to the track.

Make the NOTE ON and NOT OFF events, and add them to the Track.

```

public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);
    } catch (Exception e) {e.printStackTrace(); }
    return event;
}

} // close class

```

This is the utility method from last chapter's CodeKitchen. Nothing new.

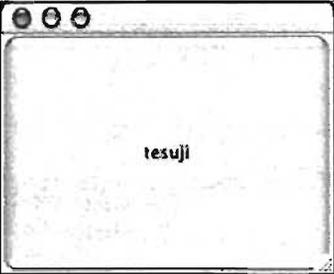
exercise: Which Layout?



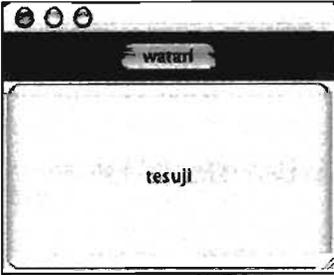
Which code goes with which layout?

Five of the six screens below were made from one of the code fragments on the opposite page. Match each of the five code fragments with the layout that fragment would produce.

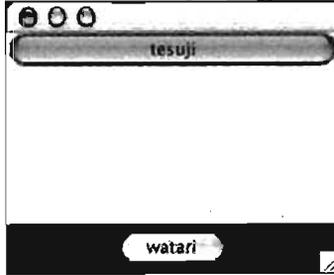
1



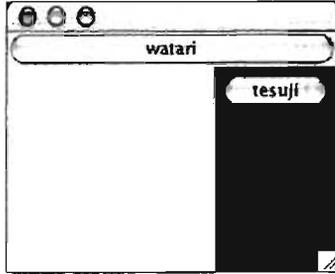
2



3



4



5



6



?

Code Fragments

D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

A

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

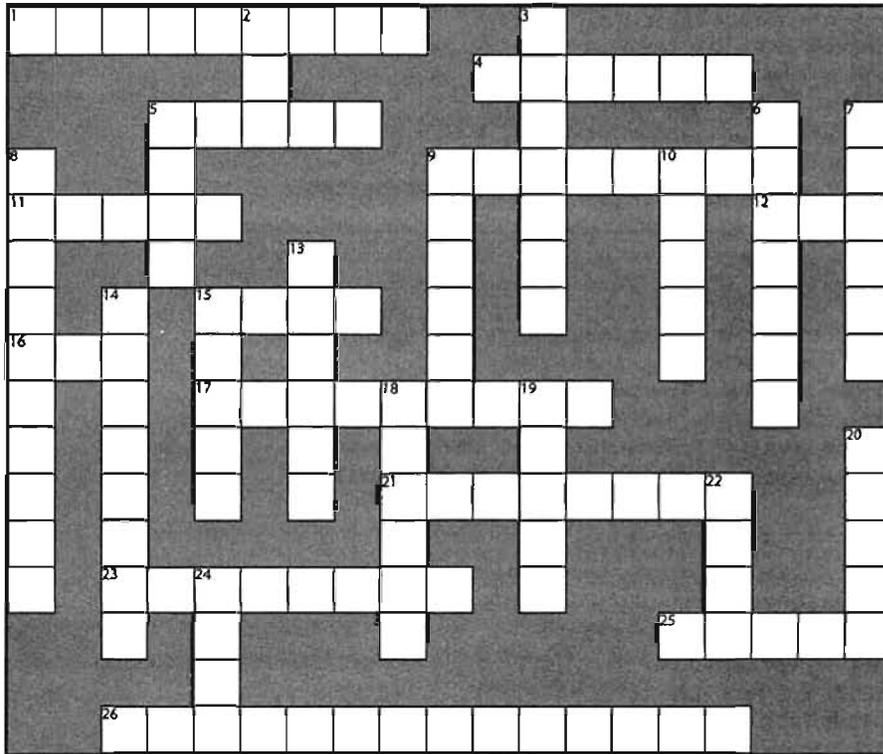
E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

puzzle: crossword



GUI-CROSS 7.0



You can do it.

Across

- 1. Artist's sandbox
- 4. Border's catchall
- 5. Java look
- 9. Generic waiter
- 11. A happening
- 12. Apply a widget
- 15. JPanel's default
- 16. Polymorphic test

- 17. Shake it baby
- 21. Lots to say
- 23. Choose many
- 25. Burton's pal
- 26. Home of
actionPerformed

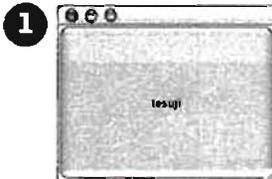
Down

- 2. Swing's dad
- 3. Frame's purview
- 5. Help's home
- 6. More fun than text
- 7. Component slang
- 8. Romulin command
- 9. Arrange
- 10. Border's top

- 13. Manager's rules
- 14. Source's behavior
- 15. Border by default
- 18. User's behavior
- 19. Inner's squeeze
- 20. Backstage widget
- 22. Mac look
- 24. Border's right

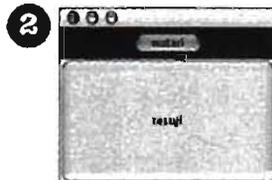


Exercise Solutions



C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER, button);
```



D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH, panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER, button);
```



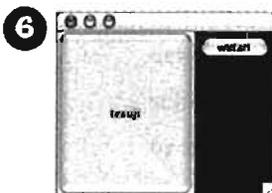
E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH, panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH, button);
```



A

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH, buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```



B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER, button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

Puzzle Answers

GUI-CROSS 7.0

