Chapter 4: The Save Order Option

The Save Order Option

Once our user clicks the Save Order menu option, we want to capture all the information for the order in a text file so we can use it to make what was ordered. We'll need to open a file, write the text to it, and close it when we're done. We did that in Lesson 4, if you remember. (Don't worry if you don't remember all the details. Part of the purpose of this lesson is reinforcement! We'll go over everything again briefly in this lesson.)

Before we do that, though, let's decide what text we want to write to the file. Let's assume our user has ordered a pizza with a deep-dish crust and toppings of sausage, peppers, onions, and extra cheese. (Sounds good to me!) Oh, and let's throw in two orders of bread sticks. For that order, here's how the text description that gets written to the file will look:

```
Pizza Order

===========

Crust:

     Deep-Dish

Toppings:

     Sausage

     Extra Cheese

     Peppers

     Onions

Sides:

     2 Bread Sticks

Deliver To:

     A. Name

     123 Some Street

     A Town, ST  12345

*** END OF ORDER ***
```

This provides an easy-to-read order, with a section in the order for each part of the form on the screen. The Toppings and Sides sections can grow or shrink as necessary. For a plain cheese pizza or an order with no sides, either or both of those sections can even be empty.

There are two ways we can write the order to a file. We can either write each line directly to the file as we decide what to write, or we can build one output string holding the entire order and then write that string to the file all at once. The trade-offs are complexity versus efficiency.

Writing each line of the order to the file as soon as we decide we need it is the simpler route. But the more file reads and writes we do, the less efficient our program is. Given the size of our program, it doesn't really matter, and in some systems, there are sometimes valid reasons for making a choice that is less efficient. But in large, complex systems, efficiency can be very important. And writing to a disk can take tens of thousands, or even hundreds of thousands, of times longer than storing information in memory. So we're going to take the more efficient path and build one output string with the whole order, then write that string all at once as our last action.

The first thing we'll do is take any existing code out of the actionPerformed() method of the SaveListener class, just like we did with the NewListener class. Then we're ready to add our output code. That gives us this class as a starting point:

```
private class SaveListener implements ActionListener
{
public void actionPerformed(ActionEvent e)
    {
    }
}
```

Next, we'll add code to create a String object to hold the text for our order. We will initialize it with the first three lines of the order output, since they're always the same. Our first line of code, then, is this:

```
String order = "Pizza Order\n" +
"===========\n" +
               "Crust:\n";
```

After that, we'll decide which crust is selected and put a line into the order to describe it, like this:

```
if (regularCrustButton.isSelected())
   order += "     Regular\n";
else if (thinCrustButton.isSelected())
   order += "     Thin\n";
else if (deepCrustButton.isSelected())
   order += "     Deep-Dish\n";
else if (handCrustButton.isSelected())
   order += "     Hand-Tossed\n";
else
   JOptionPane.showMessageDialog(frame,
   "You must select a crust type!",
   "Crust Type Error",
   JOptionPane.ERROR_MESSAGE);
```

This code lets us check the value of each radio button to see if it's selected using the isSelected() method of the JRadioButton class. That method returns *true* if the button is selected and *false* if not. Notice that as soon as we find a button that's selected, the rest of the *ifs* are bypassed by the *else* part of the *if*. We only keep checking as long as we haven't found a checked button. That logic works with radio buttons in a group because only one of them can be selected at a time, and we started out with one of them selected as a default option.

If none of the buttons is checked, we know we have an error situation, and we'll pop up a message telling our user about the problem so he or she can correct it.

We'll write the logic for our check boxes in the Toppings section a little bit differently. Unlike with our radio buttons, any, none, or all of the check boxes can be selected. This means we need to go through and check each of them to find out what toppings go in the order. We'll do that like this:

```
order += "Toppings:\n";

if (pepperoniBox.isSelected())

   order += "     Pepperoni\n";

if (sausageBox.isSelected())

   order += "     Sausage\n";

if (cheeseBox.isSelected())

   order += "     Extra Cheese\n";

if (pepperBox.isSelected())

   order += "     Peppers\n";

if (onionBox.isSelected())

   order += "     Onions\n";

if (mushroomBox.isSelected())

   order += "     Mushrooms\n";

if (oliveBox.isSelected())

   order += "     Olives\n";

if (anchovyBox.isSelected())

   order += "     Anchovies\n";
```

The JCheckBox class also has a method named *isSelected()*, and we use it here to find out which buttons are checked so we can write the corresponding toppings to the order file.

The next thing we need to check is the side order numbers. This involves three steps. First, we need to make sure that if users entered anything, they entered integer numbers. It wouldn't work to send out @#$ orders of bread sticks, would it? Second, if the entries are not numbers, we need to let users know they made a mistake and give them a chance to correct it. Third, if there is a valid numeric entry, we need to add appropriate information to our output string. The following code does all of this:

```
int bs = 0;

int bw = 0;

try

{

   if (!breadSticksText.getText().isEmpty())

      bs = Integer.parseInt(breadSticksText.getText());

   if (!buffaloWingsText.getText().isEmpty())

      bw = Integer.parseInt(buffaloWingsText.getText());

}

catch (NumberFormatException nfe)

{

   JOptionPane.showMessageDialog(frame,

      "Side order entries must be numeric,\n" +

       "and must be whole numbers",

      "Side Order Error",

      JOptionPane.ERROR_MESSAGE);

}

if (bs > 0 || bw > 0)

{

   order += "Sides:\n";

   if (bs > 0)

      order += "     " + bs + " Bread Sticks\n";

   if (bw > 0)

      order += "     " + bw + " Buffalo Wings\n";

}
```

This code creates two int variables to hold the bread stick and buffalo wing order numbers, and it initializes them to zero. Then it tries to extract the numbers from the text fields, but only if they are not empty. It does this by using the getText() method to get the text from the text field, the isEmpty() method to see if the String is empty, and the logical *not* operator (!) to test for a field that is *not* empty. If there is something in either of the Strings, the parseInt() method from the Integer class tries to convert it to an int. If there's anything there that keeps the conversion from working, the NumberFormatException will cause an error to pop up so the user can correct it.

Once we successfully convert the numbers, we add appropriate text to our output string if either of them is greater than zero.

The last item we have to add to our output is the delivery address. The only error our user could make there that we could catch is to leave a field empty. We don't want to get into editing names and addresses for validity in this example. So we'll present an error pop-up if any of the address fields are empty; otherwise we'll add them to our output string. Here's how we'll do that:

```
if (nameText.getText().isEmpty() ||
   addressText.getText().isEmpty() ||
   cityText.getText().isEmpty())
JOptionPane.showMessageDialog(frame,
   "Address fields may not be empty.",
   "Address Error",
   JOptionPane.ERROR_MESSAGE);
else
{
   order += "Deliver To:\n";
   order += "     " + nameText.getText() + "\n";
   order += "     " + addressText.getText() + "\n";
   order += "     " + cityText.getText() + "\n";
}
   order += "\n***END OF ORDER ***\n";
```

That completes the creation of the output string. If we get this far in the logic, we're ready to write the output to a file. We have done that before, so I won't bore you with the details. Here's the code:

```
try
{
   PrintStream oFile = new PrintStream("PizzaOrder.txt");
   oFile.print(order);
   oFile.close();
}
catch(IOException ioe)
{
   System.out.println("\n*** I/O Error ***\n" + ioe);
}
```

And believe it or not, that's it! Give your program a try and let me know in the Discussion Area if you have any problems with it. If you want to compare your code to mine, you can look at it here.

Complete GUI Pizza application