Chapter 2: Moving Forward Through the List

Moving Forward Through the List

When we left our application in Lesson 10, we had already created the Next menu item in the View menu and a placeholder listener for it, NextMenuItemListener, that doesn't do anything yet. The code that makes the Next menu item work goes into the actionPerformed() method of its listener. We want this option to move us forward one position in our list and display the information about the next player in the screen's text fields.

If we think about moving in the list, though, there are three situations that can prevent our program from displaying the next player. Can you think of them? Don't worry if you can't come up with all three. I couldn't, either, until I ran the program a few times and realized there was one situation I'd missed! (I crashed my program, too, but at least I found it before you did. I hate it when that happens!)

The first situation I thought of that would prevent us from seeing the next player was if we'd already reached the end of the list. Moving through the list, we've displayed our players until we've seen them all and there are no more to see. So that's one possibility we'll have to allow for.

The second one was this: What if users click the Next option before they've opened a file and loaded a list? That would cause an error because we don't actually create a list until our user opens a file. We'll need to allow for that, too.

The third one was this: What if we have a list, but no players in it? (That's the one I forgot!) If a user opens an empty file, our program will create a list to hold the players, but there won't be any players in it. That's the third condition we'll have to deal with.

One additional word of warning: If we try to get a list element when there isn't one, it's like going past the end of the list. Java throws an exception that causes the program to crash unless we catch it. By using the iterator's hasNext() method, though, we can avoid the situation.

If we check for those three conditions and don't find them, we'll know we have a list with an element for us to get, so we'll get it and display it. Here's the code to do that:

```
private class NextMenuItemListener implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        if (list == null || list.size() == 0)
            return;

      if (!isForward)
      {
         lit.next();
         isForward = true;
      }

      if (lit.hasNext())
      {
         Player p = lit.next();
         getPlayer(p);
      }
      else
      {
         JOptionPane.showMessageDialog(frame,
            "There are no more players.\nYou have reached the end of the list.",
            "End of List",
            JOptionPane.WARNING_MESSAGE);
      }
   }
}
```

The first *if* statement takes care of our first two error conditions. If we don't have a list yet, or if we have an empty list, we have nothing to display, so we just return from the listener.

The next *if* statement checks one other condition—one that's not an error, but one we need to clear up a little. Remember when we first looked at iterators and you learned that if we reverse directions with the iterator, we'll get the same list entry we got last time? In order to prevent us from seeing the same player twice when we change directions, this *if* statement checks the direction our iterator last moved. If it was backward and we're now moving forward, we do one extra read to get past the list element we just processed, and we also reset our direction to forward. To manage that, we need a Boolean variable to keep track of our direction, so we'll add one more declaration to our instance variables:

```
private boolean isForward;
```

And we'll add one more statement at the end of our OpenMenuItemListener's actionPerformed() method:

```
isForward = true;
```

Finally, the last *if* statement in the NextMenuItemListener's code checks to see if we have another element in our list. If we do, we can get it and display it using getPlayer(). If not, we're trying to move forward past the end of the list, so we'll pop up a warning dialog to tell our user.

If you run the program now and open the player file, you should be able to move forward through the list until you hit the end of it. If not, let me know and we'll figure out why.

Moving Backward Through the List

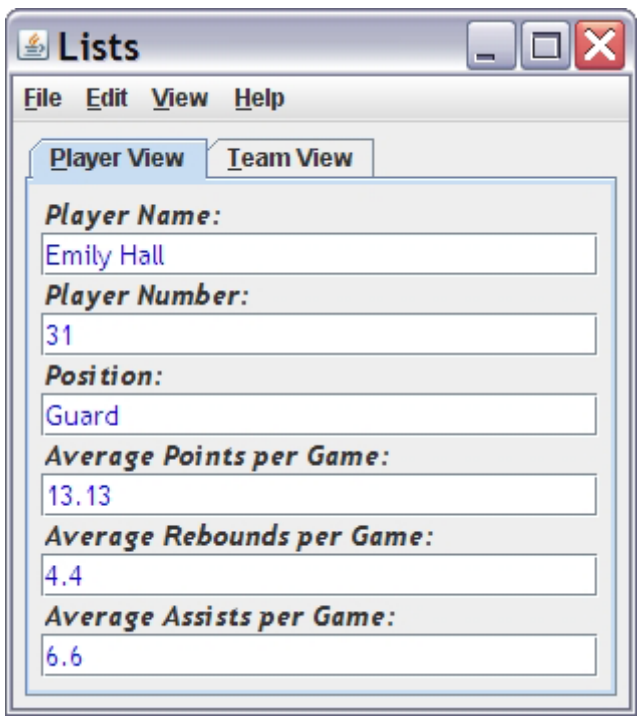
To allow us to move backward through the list, we need to change the PrevMenuItemListener in ways similar to what we just did to the NextMenuItemListener. The code is almost identical, but we need to reverse some conditions. I'd like you to try changing that method on your own, and let me know if you have questions. If you need to look at my code for reassurance, you can find it at the link below, but try to make the changes yourself first.

PrevMenuItemListener code

Changing Views

Now that we can move through our list, let's talk about our views. The cleanest way to have multiple views in a window is a technique we've all seen at one time or another as users of an application: tabbed panes. Almost as long as GUIs have been in use, there has been the need to display more data than will fit in a window or to display multiple groups of data that don't necessarily fit together. Tabbed panes have filled that need.

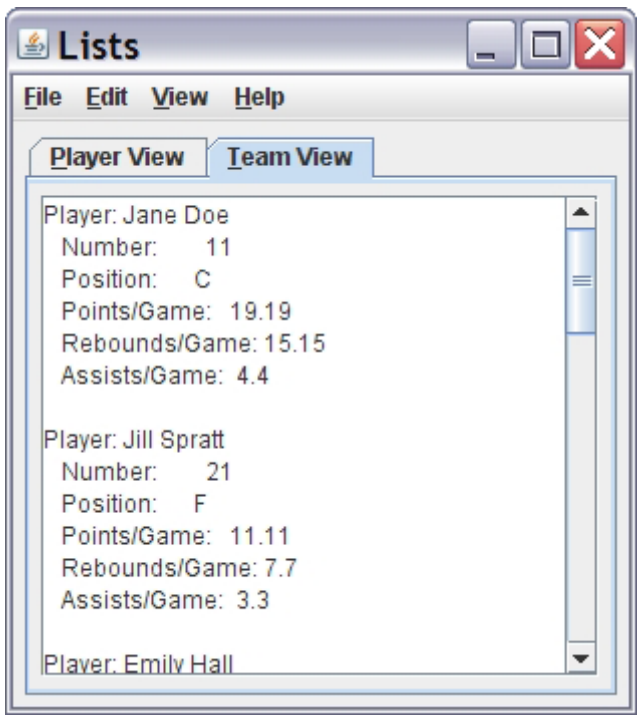
Here's a picture of our window with two tabs at the top, just below the menus, that will allow us to switch between the Player View and the Team View.



Lists window with tabs showing Player View

Java gives us the option of putting the tabs on any edge we want (top, bottom, left, or right), but the default position is at the top of the pane.

If we click the Team View tab or use the ALT + T key combination, the view will switch to the other tab, and this is what we'll see:



Lists window with tabs showing Team View

To make the tabbed panes, we'll have to make several changes to our makeContent() method. First, we'll add a tabbed pane object using the JTabbedPane class. Then we'll add two tabs to it, one that contains a panel with our Player View, and one that contains a panel with our Team View. That means we'll need to build each view in a panel instead of directly in the content pane. That doesn't sound too bad, does it? Let's get started.