Chapter 3: Wrapper Classes

Wrapper Classes

Some situations in Java don't work with primitive data types. We've just seen one of them. Java's collections are set up so that they can hold any type of object, but they can't hold any of the primitive data types. The types stored in a collection *must* be subclasses of Object, Java's ultimate parent class. Primitive types don't meet that requirement.

But we still want to be able to have collections that hold primitive types, like ints. Java meets that need by providing *wrapper classes* for us to use in such situations. Each primitive type has a wrapper class: The wrapper class for a char type is *Character*; for int it's *Integer*, for float it's *Float*, and so on. An object of each wrapper type holds one value of the associated primitive type—an Integer object holds one int value, for example.

Wrapper objects' values are set in their constructors, and each wrapper class defines a *typeValue()* method: *charValue()* for the Character class and *floatValue()* for the Float class, to give examples. The following code snippet shows examples of creating and retrieving wrapper values, which we call *boxing* and *unboxing* in Java.

```
Integer i = new Integer(5);

Character c = new Character('A');

Boolean b = new Boolean(true);

 . . .

int j = i.intValue();

char d = c.charValue();

boolean e = b.boolValue();

 . . .
```

When we actually store and retrieve entries with our map, then, we'll use Integer objects as keys rather than int values. You'll see more of how we do that in the code we're going to write today.

Loading the Player File

Now, let's look at the changes we need to make in our program so our application will work using a map. Nothing changes in our menus or window format. The first code we need to modify is in the OpenMenuItemListener inner class, where we open a file and load its data. We'll start by replacing the line that created our list:

```
list = new ArrayList<Player>();
```

with one that creates a map:

```
map = new TreeMap<Integer, Player>();
```

Our next change will be to replace the line that adds a Player to the list:

```
list.add(new Player(name, nbr, position, avgPoints, avgRebounds, avgAssists));
```

with one that adds a Player to the map, using her number as a key:

```
map.put(new Integer(nbr),
        new Player(name, nbr, position, avgPoints, avgRebounds, avgAssists));
```

Third, we don't need to sort our players anymore, so we can delete this line:

```
Collections.sort(list);
```

Our fourth change will replace the code that displays the first player in the Player View tab. These are the lines we'll replace:

```
lit = list.listIterator();
isForward = true;
if (lit.hasNext())
{
    Player p = lit.next();
    getPlayer(p);
}
```

We can also delete the declaration of the Boolean variable isForward, since we won't be using it.

Let's replace the above lines with the following line, which will call a new helper method, *findPlayer()*, to ask our user for a player number to display. When the user enters a number, this method will then display that player's information.

```
findPlayer();
```

The findPlayer() method, which we'll be using again, should be added as the last method in our program:

```
private void findPlayer()
{
    boolean isGoodNumber = false;
    Integer playerNum = new Integer(0);
    while (!isGoodNumber)
    {
        try
        {
            playerNum = new Integer(
                        Integer.parseInt(JOptionPane.showInputDialog(frame,
                        "Enter a player number:",
                        "Player Entry",
                        JOptionPane.QUESTION_MESSAGE)));
            isGoodNumber = true;
        }
        catch(NumberFormatException nfe)
        {
            JOptionPane.showMessageDialog(frame,
                "That wasn't a player number!",
                "Player Number Error",
                JOptionPane.ERROR_MESSAGE);
        }

        if(isGoodNumber)
        {
            Player p = map.get(playerNum);
            if (p == null)
            {
                JOptionPane.showMessageDialog(frame,
                    "Player number " + playerNum.intValue() + "does not exist!",
                    "Player Number Error",
                    JOptionPane.ERROR_MESSAGE);
                isGoodNumber = false;
            }
            else
                getPlayer(p);
        }
    }
}
```

Let me explain how this method works. It uses two local variables, *isGoodNumber* and *playerNum*. The main logic of the method is a loop that repeats until we find a player to display, as indicated by isGoodNumber.

Within the loop, we get a player number for our variable playerNum in the try block. In that block, we ask the user for a player number in a dialog box, then use Integer's parseInt() method to convert the user's input into an int value, which we store in playerNum. If users input something other than a number, our catch block makes sure the program doesn't blow up. The catch block tells users they entered a bad number, and then the loop repeats.

The *if* statement first makes sure we have a valid number; then it tries to get that player from the map using its get() method with the number as the key. If no map entry exists for that number, get() returns a null reference, we display an appropriate error dialog, and we set isGoodNumber back to false to force another try. If the map returns a Player object for the number, we call our getPlayer() helper method to display the player's information.

Once we have completed the Player View, we only have one more task to fix in OpenMenuItemListener. We need to build the Team View. We did that before with a for-each loop that processed the list, but for-each loops don't work with maps. Java maps do, however, have a method named *values()*, which gives us a collection that for-each loops do work with. Near the end of OpenMenuItemListener in our original program, there's this line:

```
for (Player p : list)
```

We'll replace the reference to *list* with a call to our map's values() method, and the for-each loop will work. The updated line looks like this:

```
for (Player p : map.values())
```

We've now finished our biggest task—loading the input file. The rest of our tasks will be simpler, as you'll see in the next chapter.