

Chapter 3: The Team Class

The Team Class

I've also updated the Team class to supply a formatted string for an output file. I made a few other changes while I was at it to simplify the class. Before I describe the changes, let me give you the new class so you have it to look at when I go over them. Here it is:

```
import java.util.ArrayList;

public class Team
{
    private ArrayList<Player> roster; // declare array for roster

    public Team()
    {
        roster = new ArrayList<Player>(); // create array for roster
    }

    public void addPlayer(Player player)
    {
        roster.add(player); // add player to roster
    }

    public String toString()
    {
        String teamRoster = "Team Roster\n\n"; // output String

        for (Player p : roster) // for each player in roster
        {
            teamRoster = teamRoster + p.toString() + "\n"; // add name to roster
        }

        return teamRoster; // return roster
    }

    public String toFile()
    {
        String fileRoster = ""; // output String

        for (Player p : roster) // for each player in roster
        {
            fileRoster = fileRoster + p.toFile() + "\n"; // add name to roster
        }

        return fileRoster; // return roster
    }
}
```

This class looks quite a bit different than it did before. I made some changes to get rid of some difficulties associated with using arrays. For example, when you're using an array, you have to tell Java how big it should be before you can use it. If you guess wrong and need more space, it's a pain because you can't expand an array. If you made the array too big, you wasted space.

Another problem with using an array is that we had to keep track of the number of players ourselves to know when we reached the end of our array. And for the `toString()` method to work properly, the array size had to match the number of players exactly.

Well, there's a way to get Java to do all the hard work for us with arrays. There is a class in the Java class library that acts like an array, with some additional benefits. It lets us store a bunch of similar items like an array does, it keeps track of how many items are stored, and it even grows automatically if we need more space than we said in the first place.

You might have already guessed from the import statement at the start of the chapter that the class I'm referring to is the *ArrayList* class. I added that statement so I could create an `ArrayList` object in my program.

`ArrayList` is one of a group of classes in Java that are called *collections*. These classes give us different ways to organize groups of things. Some examples of Java collections are lists, sets, and maps. We'll see more of these in future lessons. But I don't want to get too far off track right now, so we'll save those for later. Right now, let's just look at how to use the `ArrayList` and how it makes our class simpler.

The first thing in the `Team` class now is a declaration of the `ArrayList` variable *roster*. Part of the `ArrayList` declaration is the type of items we want to put in it, `Player` items. The item type is put in *angle brackets* (`<>`) to keep it separate from the variable name. In short, this line tells Java that we want to declare an `ArrayList` object named *roster* that will hold `Player` items.

Next, be sure to notice that I did not have to declare a variable to keep track of the team size. The `ArrayList` will do it for us. Any time we want to know how many items are in our list, all we need to do is call the `ArrayList`'s *size()* method. Since our `ArrayList` is named *roster*, we can call its *size()* method like this:

```
roster.size()
```

The constructor now only needs one line to allocate the new `ArrayList` object and set our variable *roster* to refer to it. I deleted the statement that initialized the team size, since the `ArrayList` takes care of it for us.

The `addPlayer()` method doesn't need to worry about trying to put too many players into the array anymore. The `ArrayList` will expand if it needs to. So all this method does is call the `ArrayList`'s *add()* method to add a player.

One nice thing about the for-each loop we used last time is that it works exactly the same for an `ArrayList` as it does for an array, so we don't need to change our `toString()` method at all!

The last change I made to this class is to add a new method named *toFile()*. Its purpose is to convert the team to a string in our file format, just like the *toString()* method built a string in display format. The new method looks pretty similar to *toString()*. The only differences are that it starts with an empty string instead of starting with a display heading. Then we add a new line to the string for each player on the team by calling the Player's *toFile()* method instead of its *toString()* method.

That takes care of the Team class. The last thing left to do is to change the TeamDriver class to write an output file.

The TeamDriver Class

There are fewer changes to this class than the last one, so there's not much left to this chapter. (I think I hear cheering out there!) As before, here's the class, then we'll go over the changes:

```
import java.io.*;

public class TeamDriver
{
    public static void main(String[] args)
    {
        Team t = new Team();

        PrintStream oFile;

        t.addPlayer(new Player("Jane Doe", 'C', 19.19, 15.15, 4.4));
        t.addPlayer(new Player("Jill Spratt", 'F', 11.11, 7.7, 3.3));
        t.addPlayer(new Player("Emily Hall", 'G', 13.13, 4.4, 6.6));
        t.addPlayer(new Player("Daisy Duck", 'G', 8.8, 2.2, 1.1));
        t.addPlayer(new Player("Minnie Mouse", 'C', 5.5, 1.1, 2.2));
        t.addPlayer(new Player("Petunia Pig", 'G', 9.9, 3.3, 6.6));
        t.addPlayer(new Player("Mother Goose", 'F', 8.0, 4.0, 2.0));

        System.out.println(t.toString());

        try
        {
            oFile = new PrintStream("TeamFile.txt");

            oFile.print(t.toFile());

            oFile.close();
        }
        catch(IOException e)
        {
            System.out.println("*** I/O Error ***\n" + e);
        }
    }
}
```

The first change is the first line, where we need to import Java's *java.io* package to be able to do file input and output. This package contains a number of classes that are used for different types of input and output.

The next change is the eighth line, where we declare our output file, a PrintStream I named *oFile*.

The data also changed slightly, in that none of the players have three names now, like "Rip Van Winkle" did. To make our input file work later, each player now has two names, as we will be looking for two name strings for each player when we read the file.

The lines that create and display the players didn't change at all.

The rest of the program creates and writes the output file. For now, don't worry too much about the *try* and *catch* keywords other than to know why they are there. Here's a brief explanation: Almost any command dealing with files can cause errors because input and output are error-prone processes. For example, what if you try to read a file that's not there? Or write a file that already exists? Or specify the wrong directory? And on and on. So Java forces us to deal with these errors in try and catch blocks.

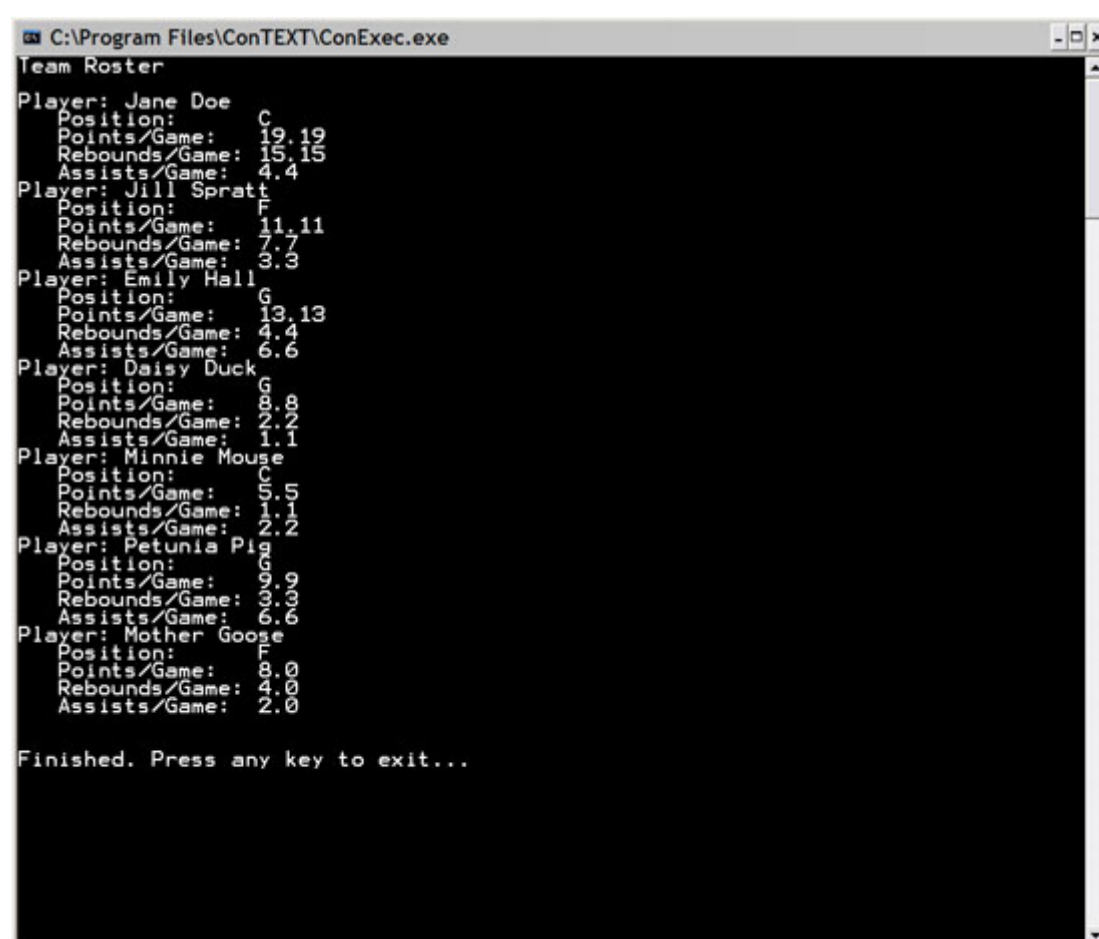
The statements that can cause output errors go in the try block. The catch blocks—there can be several—tell Java what types of errors (*exceptions*) to watch for and what to do if one is found. In our case, the catch block tells Java to look for input and output errors, which Java creates as *IOException* objects. If an *IOException* occurs (or gets *thrown*, in Java terminology), our catch block will catch it and display the ***** I/O Error ***** message I set up. That's all I'll say about these right now, I promise. For now, just put these in your program the same way I did.

The actual file statements are easier than the whole try-and-catch setup. The first command in the block creates the file using the statement format we discussed at the start of the lesson. In this case, I named my file *TeamFile.txt* and put it in the same directory as the program.

The second line uses the *print()* method from the *PrintStream* class, but it works exactly like the method we have been using for output to the screen already, so that should make sense. This command writes the output string from the *Team* object to the file with all the team's data in it.

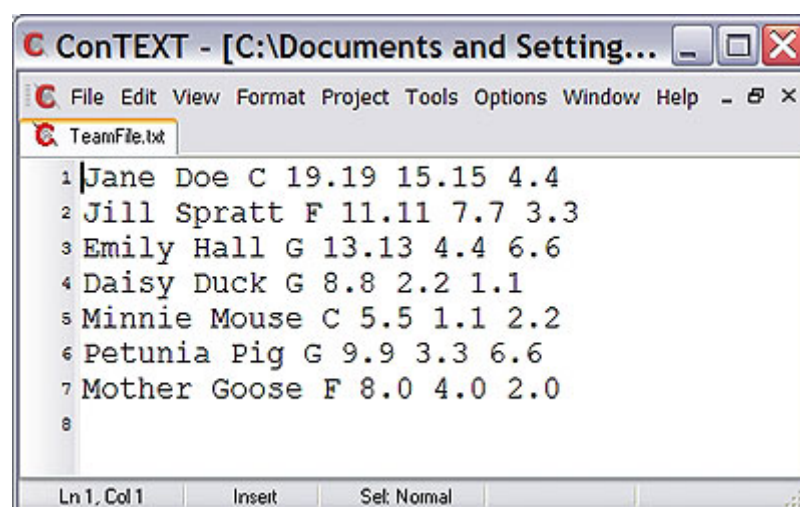
The third line is very important when dealing with files. It *closes* the file, which means that it makes sure the end of the output file is written properly. Output files need to be closed properly or we'll get errors when we try to read them. So be sure to run the *close()* method every time you finish an output file.

Now we're finally done writing our file! When we compile these classes and run the driver program, we get a window that looks like this:



```
C:\Program Files\ConTEXT\ConExec.exe
Team Roster
Player: Jane Doe
  Position: C
  Points/Game: 19.19
  Rebounds/Game: 15.15
  Assists/Game: 4.4
Player: Jill Spratt
  Position: F
  Points/Game: 11.11
  Rebounds/Game: 7.7
  Assists/Game: 3.3
Player: Emily Hall
  Position: G
  Points/Game: 13.13
  Rebounds/Game: 4.4
  Assists/Game: 6.6
Player: Daisy Duck
  Position: G
  Points/Game: 8.8
  Rebounds/Game: 2.2
  Assists/Game: 1.1
Player: Minnie Mouse
  Position: C
  Points/Game: 5.5
  Rebounds/Game: 1.1
  Assists/Game: 2.2
Player: Petunia Pig
  Position: G
  Points/Game: 9.9
  Rebounds/Game: 3.3
  Assists/Game: 6.6
Player: Mother Goose
  Position: F
  Points/Game: 8.0
  Rebounds/Game: 4.0
  Assists/Game: 2.0
Finished. Press any key to exit...
```

We also get an output file named "TeamFile.txt". When I open it in my text editor, this is what I see:



The screenshot shows a text editor window titled "ConTEXT - [C:\Documents and Setting...". The menu bar includes File, Edit, View, Format, Project, Tools, Options, Window, and Help. The file name in the title bar is "TeamFile.txt". The text content is as follows:

```
1 Jane Doe C 19.19 15.15 4.4
2 Jill Spratt F 11.11 7.7 3.3
3 Emily Hall G 13.13 4.4 6.6
4 Daisy Duck G 8.8 2.2 1.1
5 Minnie Mouse C 5.5 1.1 2.2
6 Petunia Pig G 9.9 3.3 6.6
7 Mother Goose F 8.0 4.0 2.0
8
```

The status bar at the bottom shows "Ln 1, Col 1", "Insert", and "Set: Normal".

Team roster output file

I hope that's what you got, too. If not, please let me know in the Discussion Area and we'll figure out what went wrong.