Chapter 2: Files

Files

Data files are a simple concept, but because people are constantly finding new ways to use and improve on them, they're still a very big topic. *Files* are simply data that is organized in a recognizable, reusable way and stored on some external media. We use files for an endless variety of purposes: for storing pictures from digital cameras; for sound files, like MP3 and WAV formats; for video, like on YouTube; and for databases that store incredible amounts of data for businesses and other organizations. But all those files use formats that are very complex.

We're going for a much simpler scheme—one that's used on practically every computer in existence. The process we'll use simply writes text to a file in a format we can read again. Even considering text in a file requires some planning, though, since finding the following text in a file probably wouldn't be very useful to most of us:

```
#gbarl{border-top:1px solid #c9d7f1;font-size:0;position:absolute;width:110%}
```

This is one line of CSS code that would be stored in a file and used by a web page for styling. The colons and semicolons within the text act as delimiters, separating different parts of the text so the computer can break it apart and process the text accordingly. When designing your own data text files, you also need to use delimiters to organize the text in a way that's easy for a computer to break apart into smaller chunks.

There are a number of formats used for text files that contain data. Before we write a file, we need to make sure that everyone who might want to use it agrees on the format. And if we want to read an existing file, we'll need to find out its format before we can successfully read it in a program. Let me explain what I mean by *format* a little further.

A text file that contains data is usually divided into *records*, with one record on each line of the file. A record is a group of related data fields containing information about one of the entities in the file. For example, if we used data on the players from the team project we did in the last lesson, each record would hold information about one of the players.

We also need to have a *record format* that matches the file we want to use. This helps us know what information is in each record and where it is in the record. Without a record format, you wouldn't know what information the following line contains:

```
John Doe C 19.19 15.15 4.4
```

It's not easy to tell that this is the player information for one of the players from the exercise in the last lesson. But if we knew the record format, we would know that John Doe is the player's name, he's a center, and he averages 19 points per game. So it's important to have a record format that tells us what information is in each record of the file.

We'll also need to know what *separator character* the file uses between fields in a record, if any. Let me show you some of the more popular ways to store the same information with different separator characters:

```
John Doe C 19.19 15.15 4.4

John,Doe,C,19.19,15.15,4.4

John         Doe          C19.1915.15 4.4
```

The first line uses a blank character as a separator. The second uses a comma. The third does not have a separator at all but expects each data item to start in a certain location in the record. The first name, for example, can be up to 12 characters long, followed by 12 more for the last name. The player's position is in the next column (column 25). The average points per game for the player is next, in column 26, and takes five columns, including a decimal point. After that is average rebounds, and last is average assists.

You can see that there are a number of text file formats. So before we write or read them, we need to make sure we have the format correct. For this lesson, since none of our fields will contain blanks, we'll use the first format above, with a space between fields in a record. We could have just as easily used one of the others, since they all work equally well. But I had to choose one of them!

Declaring Files in Java

As you might suspect, if we're going to use a file in a Java program, we have to tell Java about it. To do that, we have to declare the files in our program, just like any other data object we want to use.

We'll start by going over how to declare an output text file. Java provides a convenient class for us to use when declaring an output text file. It's called the *PrintStream* class. This class does text output in a way that's very similar to displaying text on your screen. There are other ways to write output files, but for learning purposes, this class is the easiest way to start.

Before we can use the output *stream*, we have to declare it like this:

```
PrintStream outputFile = new PrintStream("FileName.txt");
```

This declaration creates a variable, *outputFile*, that we'll use in our program to refer to the file. The string *FileName.txt* represents the name of the physical file on the disk. So if we wanted to create a file named *NewFile.txt* on our disk, we would use this declaration:

```
PrintStream outputFile = new PrintStream("NewFile.txt");
```

I need to mention where Java creates and looks for files. If you use an unqualified name like I just did, Java will look for (or create) the file in the same disk directory the program file is in. If you want to create your file in another directory or on another disk, you need to put the full path into the name, like this:

```
PrintStream outputFile = new PrintStream("D:\\NewFile.txt");
```

This will create a file named *NewFile.txt* in the root directory of the D: drive. Notice that I put two backslashes (\\) where there would normally be one in the file path. That's because in Java, you have to use two backslashes to represent one inside a character string. If Java finds a single backslash in a string, it uses the next character to try to create an *escape character*, a character with a special meaning. For example, Java treats "\n" as a line return and "\t" as a tab character.

Okay, let's get off this rabbit trail and back to the topic of creating a file. We've declared the file so that Java knows how to use it. Now we need to actually write data to it in the format we said we would. That means we need to create a character string from a player that represents one line in the file. I've modified my *Player* class to do that, so let me show it to you, then I'll explain the changes I made. Here is my new Player class:

Print this code

```
public class Player

{

    private String name;

    private char position;

    private double avgPoints;

    private double avgRebounds;

    private double avgAssists;







    public Player(String pName, char pPos, double pPoints, double pRebounds, double pAssists)

    {

      name = pName;

      position = pPos;

      avgPoints = pPoints;

      avgRebounds = pRebounds;

      avgAssists = pAssists;

   }



 public String toString()

 {

    return "Player: " + name +

           "\n   Position:      " + position +

           "\n   Points/Game:   " + avgPoints +

           "\n   Rebounds/Game: " + avgRebounds +

           "\n   Assists/Game:  " + avgAssists;

 }



 public String toFile()

    {

       return name + " " + position + " " + avgPoints + " " + avgRebounds + " " +

           avgAssists;

    }

}
```

I added a new method I named *toFile()* to the class. The purpose of this method is to format the player's data for output to a file in the format we laid out earlier: consecutive fields with spaces between them.

That gives us a string to use for each line of our output file. To see how we write the strings to the file, we'll look at our other classes, Team and TeamDriver.