Chapter 4: for Loops

`for` Loops

Now that we've played around with arrays a little bit, let's look at another improvement we can make to our classes. Specifically, we're going to focus on the while loop we used to build the output string for our team roster. It's in the Team class, right at the end, in the toString() method:

```
public String toString()

    {

        String teamRoster = "Team Roster\n\n"; // output String

        int i = 0;                             / loop counter


    while (i < teamSize)       // while more players

    {

      teamRoster = teamRoster + roster[i].toString() + "\n"; // add to roster

      i++;                 // increment loop counter

    }

        return teamRoster;                        // return roster

    }
```

To make the loop work, I had to do three things. First, I had to set up a counter to keep track of how many players I had already added to the output string. I used a local variable named *i* for that. Second, each time through the loop, I had to check to see if *i* had reached the size of the team yet so I'd know when to stop. And third, I had to increment my counter *i* each time the loop ran. Wouldn't it be nice if there were a convenient loop form that let me do all that in one place instead of in three different lines of code? (Don't you just hate it when teachers ask rhetorical questions?)

Of course, since I asked that question, you already know there must be a more convenient form of a loop for me to use. It's called a *for* loop. It lets us do all three of those things in one statement. We use it most often when we want a counter to control a loop.

Since it's called a for loop, you might expect it to have the word *for* in it, and I won't disappoint you. Here is Java's for loop format:

```
for (<initialization>; <condition>; <increment>)

{

    // code for loop body goes here

}
```

The first line of this loop gives us all three of the items we used to control the while loop. The *initialization* is where we'll set the starting value for our counter. This part of the loop control is executed only once, at the beginning of the loop process.

The *condition* part of the line is where we put the condition that will control how many times the loop executes. Whatever condition we put there is tested at the beginning of each cycle through the loop to see if it's time to stop the loop yet. The loop will continue as long as the condition is true.

The last part of the line, the *increment* part, is also executed each time through the loop, but it's the last thing done.

There's one important format detail to notice in the for loop. We separate the different parts of the control with semicolons (;), not commas like other lists in Java.

To show you how it works, let's set up a for loop that's equivalent to the while loop we used in our method:

```
for (int i = 0; i < teamSize; i++)

{

    teamRoster = teamRoster + roster[i].toString() + "\n"; // add to roster

}
```

This loop does *exactly* what the while loop did. It sets up the counter *i*, compares it to the variable teamSize at the start of each iteration to see when to stop, and increments *i* at the end of each iteration. To put it in plain English, this loop will execute once for every value of *i* between zero and teamSize.

Here is the complete method with the loop modified to give you a better picture of how it looks:

```
public String toString()

{

    String teamRoster = "Team Roster\n\n"; // output String



    for (int i = 0; i < teamSize; i++)     // for all players

    {

        teamRoster = teamRoster + roster[i].toString() + "\n"; // add name to roster

    }



    return teamRoster;                    // return roster

}
```

By now you might be asking, "Why add another type of loop if the first one was good enough?" That's a good question. The answer is that programmers are just as lazy as anyone else, and this format makes their job a little easier by putting all the details of managing a loop in one place, especially a loop controlled by a counter.

I hope that made sense, because now I'm going to throw yet one more loop variation at you. (Then I'll quit, I promise!)

This last format is even simpler when working with a Java *collection*. Java has several forms of collections, and we'll see more of them in future lessons. For now, we'll define a collection as a group of things put together into one object. In Java, arrays are one form of collection, and that makes this last form of loop workable for us. With this loop format, we don't even have to keep a count or have a condition to test. Java will take care of all that for us.

The last loop format is often called the *for-each* loop since it runs once for each item in the collection, no matter how many there are. Java makes sure that each item in the collection is processed and that none of them are processed twice. Here's the format of the for-each loop:

```
for (<element> : <collection>)

{

    // code for loop body goes here

}
```

To describe this loop in English, we could say, "For each *element* in *collection*, execute the loop body." Let's set up our toString method one more time with this new loop type. Ideally we could set it up like as below. But if you run it that way, you'll get an error, so don't run it yet:

```
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

}
```

The reason this would fail if you ran it is because `for Player p : roster` means "for each player in the roster array". But earlier in the Team class I said there would be "up to 20" players in a team by creating an array of 20 (initially empty) elements.

```
roster = new Player[20]; // create array
```

Then in TeamDriver, I only added 7 members to the team. To use the for...each loop, you would have to change that 20 to match the actual number of elements in the array like this:

```
roster = new Player[7]; // create array
```

If you make that change, compile, and run, then the code should run without an error.

As with most things in programming, there really isn't a "right way", "wrong way", "good way", "bad way" to do things. There are just different ways. In this last example, I had three choices for looping through the team players. The last one, of for...each, would just require that I change the code so that the initial array size match the actual number of elements in the array. In situations where that's not practical, just use a looping method that is practical for the occasion.