Chapter 2: Types of Collections

Types of Collections

Before we get into the particulars of the classes, let's think about the different ways we organize our data. The reason there is more than one Java collection class is that there are several ways we organize data, and one class can't support them all. We often use these different organizations in our daily lives without stopping to recognize how they're different.

The simplest and perhaps most often used type of collection is a basic *list*. We use lists constantly: grocery lists, task lists, lists of pros and cons about decisions, and on and on. Lists are a part of our lives, like it or not. They're also often part of computer systems, where they take on some more specific characteristics.

In a computer system, we define a list as an *ordered* collection. When we use the term *ordered* here, we mean that each item has a position in the list, with one item in front of it (unless it's the first item) and one item behind it (unless it's the last item). In Java, we can retrieve a list's items from first to last, from last to first, or from a particular position in the list.

Sorted lists are another common form of lists. We often want to see list items in a certain order. For example, we might want to sort a task list by due date. Keeping a list sorted involves a certain amount of overhead, so we should only use them when necessary.

Queues and *stacks* are both more specialized forms of lists that we use often without thinking about it. If you've ever stood in line at an amusement park, a bank, or a grocery store, you have been in a queue. A queue has one added restriction that makes it different from a list: All new items are added at one end of the list and removed from the other end. It's a *first-in, first-out* list. We use that concept so often in computer systems that it has its own acronym: FIFO. Computer systems, especially shared or networked systems, use queues constantly. Every time we send an e-mail, it moves through a number of queues on the way to its destination.

Stacks also differ from basic lists in one way: Instead of FIFO, stacks process items in LIFO order. As you might have guessed, that means *last-in, first-out*. To visualize a stack, think of a stack of plates. We normally add to the stack by putting plates on top, then take plates off the top when we need them. Computer systems use stacks to manage their resources, especially memory. You may not have realized it, but every time you call a method in a Java program, the computer uses a stack to manage your program's resources.

The last one I'll mention in detail is the *map*. A map also has some special properties that differentiate it from other collections.

And it's another structure that we use frequently—whenever we look up a telephone number, in fact. A map is a structure that contains two values for each entry: its *key* and its *value*. In a phone book (or in your cell phone), the key is the name of the person you want to call. The value is the phone number you get when you look up the name. Again, computer systems use maps often to store and retrieve information quickly when you have a key. It's much faster to find an entry when you can go straight to it rather than have to look through an entire list.

Java has a few more collections that are available if you ever need them, but they're not used as often, so I'll keep their descriptions very short.

- A *set* is a collection with no duplicate values. If you remember the sets you learned about in high school algebra, you'll realize that property applies to them, too.

- A *priority queue* is like a queue, but whenever you retrieve the next item from the queue, you're guaranteed to get the highest priority item in the queue. Obviously, for this to work, each item must have a priority value to establish its position in the queue.

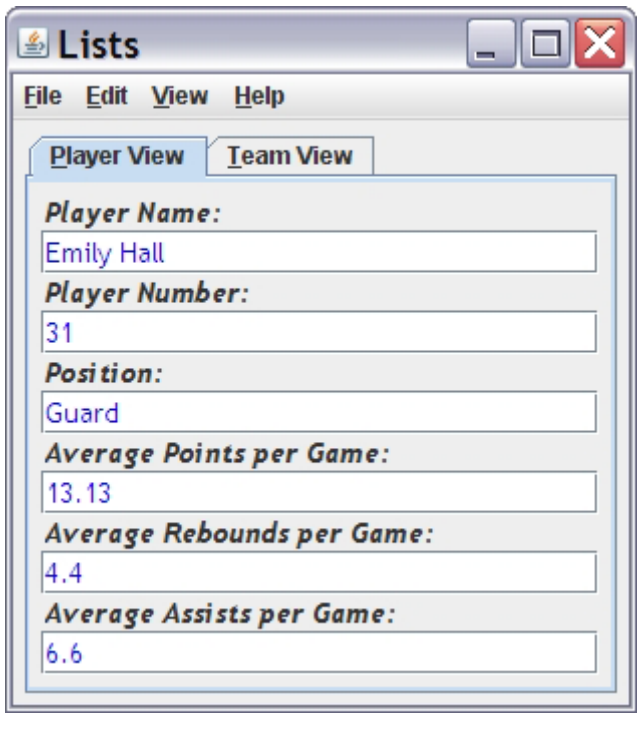
- Last are *trees* and *graphs*, often used in games and transportation systems. A *tree* is a list where instead of one predecessor and one successor, each item has one predecessor (its parent) and can have multiple successors (its children). A *graph* is a collection where there are no limits to predecessor or successor connections. Think of our airport system, where you can get from one end of the country to the other in many different ways. That's a graph.

Sets, priority queues, trees, and graphs are important in a number of technical applications, but we don't use them very often in early programming. So we won't talk about them anymore in this course. We'll concentrate on the collections that see more use in general-purpose programming.

Our approach to these collections will be to first create a simple application that uses a list and then see how using different collections changes the application. Our application will reuse some of the concepts we learned in working on our team roster projects back in Lessons 2 through 4, and we'll add some new capabilities to our repertoire as well.

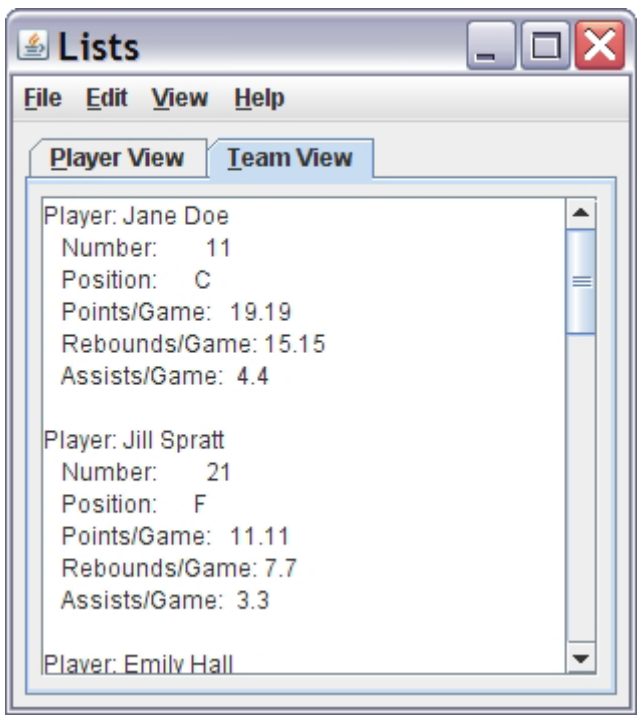
The Lists Application

The first version of our application will present two views of our team roster, with each view using certain features of our list. The first view will display all the information about a single player, and we'll be able to move forward and backward through the list of players. That screen will look like this:



Team list Player View

The second view will show a scrollable window with a text version of all players' information, something like this:



Team List Team View

The File menu will have Open and Exit options. The Open menu item will allow us to choose a player file to open, and the Exit menu item will close the window.

The View menu will have Next and Previous options. The Next and Previous menu items will move us forward and backward in the list, updating the text fields in the Player View.

Between the two views, we will explore several of the capabilities of Java's list classes. Let's get started!