Chapter 2: Maps

Maps

As you learned in Lesson 10, a map is a data structure that uses two data components for each entry. One data component in a map entry is its *key*, the value we use to look up the entry. The other data component is its *value*, the information we get back when we look it up. In Lesson 10, I used the analogy of a phone book, where you look up a phone number by finding the name of the person or business you want to call. The key is the name, and the value is the phone number.

Many business applications use this type of search when looking up records. That's why someone usually asks you for your account number when you call your bank, for example, or your phone number when you call your phone company, or your student ID when you're talking to your school office. Those pieces of information are the keys to look up your records.

We're going to set up a similar structure for our basketball team application. Suppose that instead of starting at the beginning of the team and looking through the entries one by one, we want to go directly to a particular entry. The best way to do that is by using a map.

There are two types of maps in Java, and the first thing we need to do is decide which one to use for our application. In some ways, these two types function the same, and the methods that store and extract data by key look the same to us from a programming point of view. The differences between them have to do with their internal storage mechanism. I won't go into much detail about the internals—just take it from me that their internal structures make each of them better at some things and not so good at others.

The first map type we'll talk about is the *HashMap* class. It stores and retrieves data elements by key far more efficiently than any other collection structure. But it has one drawback: Its internal storage scheme makes it impossible to access the data in any predictable order. So operations like Next and Previous have no meaning in a HashMap.

The *TreeMap*, on the other hand, allows us to store and retrieve data by key much more efficiently than with any kind of list, but still not as efficiently as with a HashMap. It does have one significant advantage, though: It allows us to find an entry's next and previous entries based on the sequence of its key values.

Let me give you an example. Let's say that we want to store and retrieve the players in our player file by number. To do that, we need to read the file like we did in our earlier versions of the program, but this time we want to store the data in a map and use each player's number as her key.

First, assume that we store all the players in a HashMap using their numbers as keys. Let's also assume that the first player we look up is player number 22. If we display her information, we see that we've looked up Mother Goose. If we try to look up the next player in a HashMap, though, we could end up getting *any other player*. We would probably *not* get the player with the next number, or the player with the next name alphabetically, or the player we added to the map after Mother Goose. We could not predict who would show up next. That's because a HashMap does not allow iterators, nor does it have any kind of Next or Previous operations.

Now, in contrast, assume that instead of storing players in a HashMap, we store them in a TreeMap. We can look up player number 22, just not quite as efficiently as with a HashMap. But in this case, we *can* predict which player we'll get in a Next or Previous operation. If we ask for the next player, we get the next higher number, which in our file is player 31, Emily Hall (my personal favorite). If we ask for the previous player, we get player 21, Jill Spratt. Next and Previous operations get the next or previous player by key (player number). And if there is no next or previous entry, we get back a NULL player, which tells us that we have reached the end of the players.

Since we have been providing Next and Previous operations in our applications so far, using a TreeMap to store players makes the most sense so that we can continue to use those operations.

There is one more decision to make before we set up our map. What should we use for the key? We could use name, or we could use player number. (Technically, we could use any data item that is unique to the player, but these are the two ways we normally think of to identify players on a team.) Since we have already been talking about using player numbers, let's stick with that.

Now that we've got that covered, let's talk about the changes we'll need to make to our program.

Declaring a Map

The first change is right at the top of our class. Instead of using an ArrayList to store our Player objects, we'll set up a TreeMap. And since TreeMaps don't use iterators, we won't need the ListIterator declaration, either. We'll replace both of those declarations with one TreeMap declaration that looks like this:

```
private TreeMap<Integer, Player> map;
```

This declaration has a slightly different format than the ArrayList's did. Instead of one data type in the angle brackets after the class name, there are two. That's because whenever we declare a map in Java, whether it's a TreeMap or a HashMap, we have to tell Java the key's data type *and* the data type of the value that the key *maps* to. In Java terminology, each key maps to the value that we get when we retrieve data using that key. So in our example, player number 22 (key) maps to Mother Goose (value), while 31 (key) maps to Emily Hall (value).

Using a type of Player for the data we look up makes sense; when we enter a key we want to get a Player object back. But why did I use a type of Integer for the key? The player number in the Player class is an int value; why not just use that? The answer is that all Java collections require the data they store to be objects; that applies to keys as well as values. So I substituted int's *wrapper class*, Integer, in its place.

This is the first wrapper class we've used in this course, so in the next chapter I'll take a moment to explain wrapper classes.