Exercise 4.37 Add a further field, of your choosing, to the **Track** class, and provide accessor and mutator methods to query and manipulate it. Find a way to use this information in your version of the project; for instance, include it in a track's details string, or allow it to be set via a method in the **MusicOrganizer** class.

Exercise 4.38 If you play two tracks without stopping the first one, both will play simultaneously. This is not very useful. Change your program so that a playing track is automatically stopped when another track is started.

4.12 The Iterator type

Iteration is a vital tool in almost every programming project, so it should come as no surprise to discover that programming languages typically provide a wide range of features that support it, each with their own particular characteristics suited for different situations.

Concept

An **iterator** is an object that provides functionality to iterate over all elements of a collection.

We will now discuss a third variation for how to iterate over a collection that is somewhat in the middle between the while loop and the for-each loop. It uses a while loop to perform the iteration, and an *iterator object* instead of an integer index variable to keep track of the position in the list. We have to be very careful with naming at this point, because **Iterator** (note the uppercase *I*) is a Java *class*, but we will also encounter a *method* called **iterator** (lowercase *i*), so be sure to pay close attention to these differences when reading this section and when writing your own code.

Examining every item in a collection is so common that we have already seen a special control structure—the for-each loop—that is custom made for this purpose. In addition, Java's various collection library classes provide a custom-made common type to support iteration, and **ArrayList** is typical in this respect.

The **iterator** method of **ArrayList** returns an **Iterator** object. **Iterator** is also defined in the **java.util** package, so we must add a second import statement to the class file to use it:

```
import java.util.ArrayList;
import java.util.Iterator;
```

An **Iterator** provides just four methods, and two of these are used to iterate over a collection: **hasNext** and **next**. Neither takes a parameter, but both have non-**void** return types, so they are used in expressions. The way we usually use an **Iterator** can be described in pseudo-code as follows:

```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next element
    do something with that element
}
```

In this code fragment, we first use the **iterator** method of the **ArrayList** class to obtain an **Iterator** object. Note that **Iterator** is also a generic type, so we parameterize it with the type of the elements in the collection we are iterating over. Then we use that **Iterator** to repeatedly check whether there are any more elements, **it.hasNext**(), and to get the next element, **it.next**(). One important point to note is that it is the **Iterator** object that we ask to return the next item, and not the collection object. Indeed, we tend not to refer directly to the collection at all in the body of the loop; all interaction with the collection is done via the **Iterator**.

Using an **Iterator**, we can write a method to list the tracks, as shown in Code 4.8. In effect, the **Iterator** starts at the beginning of the collection and progressively works its way through, one object at a time, each time we call its **next** method.

Code 4.8
Using an **Iterator** to list the tracks

```
/**
 * List all the tracks.
 */
public void listAllTracks()
{
    Iterator<Track> it = tracks.iterator();
    while(it.hasNext()) {
        Track t = it.next();
        System.out.println(t.getDetails());
    }
}
```

Take some time to compare this version to the one using a for-each loop in Code 4.7, and the two versions of **listAllFiles** shown in Code 4.3 and Code 4.5. A particular point to note about the latest version is that we use a while loop, but we do not need to take care of the **index** variable. This is because the **Iterator** keeps track of how far it has progressed through the collection, so that it knows both whether there are any more items left (**hasNext**) and which one to return (**next**) if there is another.

One of the keys to understanding how an **Iterator** works is that the call to **next** causes the **Iterator** to return the next item in the collection *and then move past that item*. Therefore, successive calls to **next** on an **Iterator** will always return distinct items; you cannot go back to the previous item once **next** has been called. Eventually, the **Iterator** reaches the end of the collection and then returns *false* from a call to **hasNext**. Once **hasNext** has returned *false*, it would be an error to try to call **next** on that particular **Iterator** object—in effect, the **Iterator** object has been "used up" and has no further use.

On the face of it, **Iterator** seems to offer no obvious advantages over the previous ways we have seen to iterate over a collection, but the following two sections provide reasons why it is important to know how to use it.

4.12.1 Index access versus iterators

We have seen that we have at least three different ways in which we can iterate over an **ArrayList**. We can use a for-each loop (as seen in Section 4.9.1), the **get** method with an integer index variable (Section 4.10.2), or an **Iterator** object (this section).

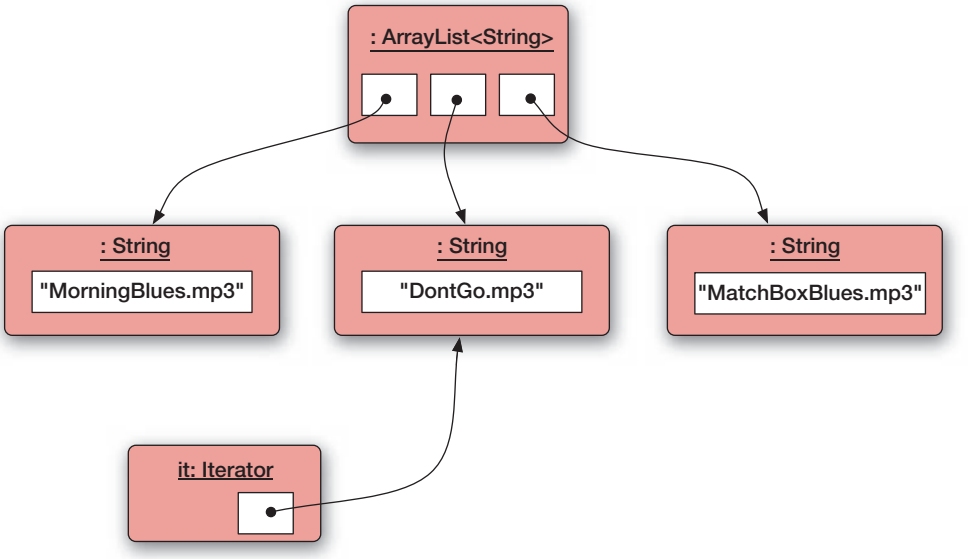

Figure 4.5
An iterator, after one iteration, pointing to the next item to be processed

From what we know so far, all approaches seem about equal in quality. The first one was maybe slightly easier to understand, but the least flexible.

The first approach, using the for-each loop, is the standard technique used if all elements of a collection are to be processed (i.e., definite iteration), because it is the most concise for that case. The two latter versions have the benefit that iteration can more easily be stopped in the middle of processing (indefinite iteration), so they are preferable when processing only a part of the collection.

For an **`ArrayList`**, the two latter methods (using the while loops) are in fact equally good. This is not always the case, though. Java provides many more collection classes besides the **`ArrayList`**. We shall see several of them in the following chapters. For some collections, it is either impossible or very inefficient to access individual elements by providing an index. Thus, our first while loop version is a solution particular to the **`ArrayList`** collection and may not work for other types of collections.

The most recent solution, using an **`Iterator`**, is available for all collections in the Java class library and thus is an important *code pattern* that we shall use again in later projects.

4.12.2 Removing elements

Another important consideration when choosing which looping structure to use comes in when we have to remove elements from the collection while iterating. An example might be that we want to remove all tracks from our collection that are by an artist we are no longer interested in.

We can quite easily write this in pseudo-code:

```
for each track in the collection {
    if track.getArtist() is the out-of-favor artist:
        collection.remove(track)
}
```

It turns out that this perfectly reasonable operation is not possible to achieve with a for-each loop. If we try to modify the collection using one of the collection's **remove** methods while in the middle of an iteration, the system will report an error (called a **ConcurrentModificationException**). This happens because changing the collection in the middle of an iteration has the potential to thoroughly confuse the situation. What if the removed element was the one we were currently working on? If it has now been removed, how should we find the next element? There are no generally good answers to these potential problems, so using the collection's **remove** method is just not allowed during an iteration with the for-each loop.

The proper solution to removing while iterating is to use an **Iterator**. Its third method (in addition to **hasNext** and **next**) is **remove**. It takes no parameter and has a **void** return type. Calling **remove** will remove the item that was returned by the most recent call to **next**. Here is some sample code:

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it.remove();
    }
}
```

Once again, note that we do not use the **tracks** collection variable in the body of the loop. While both **ArrayList** and **Iterator** have **remove** methods, we must use the **Iterator**'s **remove** method, not the **ArrayList**'s.

Using the **Iterator**'s **remove** is less flexible–we cannot remove arbitrary elements. We can remove only the last element we retrieved from the **Iterator**'s **next** method. On the other hand, using the **Iterator**'s **remove** is allowed during an iteration. Because the **Iterator** itself is informed of the removal (and does it for us), it can keep the iteration properly in sync with the collection.

Such removal is not possible with the for-each loop, because we do not have an **Iterator** there to work with. In this case, we need to use the while loop with an **Iterator**.

Technically, we can also remove elements by using the collection's **get** method with an index for the iteration. This is not recommended, however, because the element indices can change when we add or delete elements, and it is quite easy to get the iteration with indices wrong when we modify the collection during iteration. Using an **Iterator** protects us from such errors.

Exercise 4.39 Implement a method in your music organizer that lets you specify a string as a parameter, and then removes all tracks whose titles contain that string.