when printing the full listing) or we can be selective and filter the list (as we did when we printed only a subset of the collection). The body of the loop can be as complicated as we like.

With its essential simplicity necessarily comes some limitations. For instance, one restriction is that we cannot change what is stored in the collection while iterating over it, either by adding new items to it or removing items from it. That doesn't mean, however, that we cannot change the states of objects already within the collection.

We have also seen that the for-each loop does not provide us with an index value for the items in the collection. If we want one, then we have to declare and maintain our own local variable. The reason for this has to do with abstraction, again. When dealing with collections and iterating over them, it is worth bearing two things in mind:

- A for-each loop provides a general control structure for iterating over different types of collection.
- There are some types of collections that do not naturally associate integer indices with the items they store. We will meet some of these in Chapter 6.

So the for-each loop abstracts the task of processing a complete collection, element by element, and is able to handle different types of collection. We do not need to know the details of how it manages that.

One of the questions we have not asked is whether a for-each loop can be used if we want to stop partway through processing the collection. For instance, suppose that instead of playing every track by our chosen artist, we just wanted to find the first one and play it, without going any further. While in principle it *is* possible to do this using a for-each loop, our practice and advice is not to use a for-each loop for tasks that might not need to process the whole collection. In other words, we recommend using a for-each loop only if you *definitely* want to process *the whole collection*. Again, stated in another way, once the loop starts, you know for sure how many times the body will be executed—this will be equal to the size of the collection. This style is often called *definite iteration*. For tasks where you might want to stop partway through, there are more appropriate loops to use—for instance, the *while loop*, which we will introduce next. In these cases, the number of times the loop's body will be executed is less certain; it typically depends on what happens during the iteration. This style is often called *indefinite iteration*, and we explore it next.

4.10 Indefinite iteration

Using a for-each loop has given us our first experience with the principle of carrying out some actions repeatedly. The statements inside the loop body are repeated for each item in the associated collection, and the iteration stops when we reach the end of the collection. A for-each loop provides *definite iteration*; given the state of a particular collection, the

Copyright © 2016. Pearson Education, Limited. All rights reserved.

loop body will be executed the number of times that exactly matches the size of that collection. But there are many situations where we want to repeat some actions, but we cannot predict in advance exactly how many times that might be. A for-each loop does not help us in those cases.

Imagine, for instance, that you have lost your keys and you need to find them before you can leave the house. Your search will model an indefinite iteration, because there will be many different places to look, and you cannot predict in advance how many places you will have to search before you find the keys; after all, if you could predict that, you would go straight to where they are! So you will do something like mentally composing a list of possible places they could be, and then visit each place in turn until you find them. Once found, you want to stop looking rather than complete the list (which would be pointless).

What we have here is an example of *indefinite iteration*: the (search) action will be repeated an unpredictable number of times, until the task is complete. Scenarios similar to key searching are common in programming situations. While we will not always be searching for something, situations in which we want to keep doing something until the repetition is no longer necessary are frequently encountered. Indeed, they are so common that most programming languages provide at least one—and commonly more than one—loop construct to express them. Because what we are trying to do with these loop constructs is typically more complex than just iterating over a complete collection from beginning to end, they require a little more effort to understand. But this effort will be well rewarded from the greater variety of things we can achieve with them. Our focus here will be on Java's *while loop*, which is similar to loops found in other programming languages.

4.10.1 The while loop

A *while loop* consists of a header and a body; the body is intended to be executed repeatedly. Here is the structure of a while loop where *boolean condition* and *loop body* are pseudocode, but all the rest is the Java syntax:

```
while(boolean condition) {
    loop body
}
```

The loop is introduced with the keyword **while**, which is followed by a boolean condition. The condition is ultimately what controls how many times a particular loop will iterate. The condition is evaluated when program control first reaches the loop, and it is re-evaluated each time the loop body has been executed. This is what gives a while loop its indefinite character—the re-evaluation process. If the condition evaluates to *true*, then the body is executed; and once it evaluates to *false*, the iteration is finished. The loop's body is then skipped over and execution continues with whatever follows immediately after the loop. Note that the condition could actually evaluate to *false* on the very first

Copyright © 2016. Pearson Education, Limited. All rights reserved.

time it is tested. If this happens, the body won't be executed at all. This is an important feature of the while loop: the body might be executed zero times, rather than always at least once.

Before we look at a proper Java example, let us look at a pseudo-code version of the key hunt described earlier, to try to develop a feel for how a while loop works. Here is one way to express the search:

```
while(the keys are missing) {
    look in the next place
}
```

When we arrive at the loop for the first time, the condition is evaluated: the keys are missing. That means we enter the body of the loop and look in the next place on our mental list. Having done that, we return to the condition and re-evaluate it. If we have found the keys, the loop is finished and we can skip over the loop and leave the house. If the keys are still missing, then we go back into the loop body and look in the next place. This repetitive process continues until the keys are no longer missing.[1]

Note that we could equally well express the loop's condition the other way around, as follows:

```
while(not (the keys have been found)) {
    look in the next place
}
```

The distinction is subtle—one expressed as a status to be changed, and the other as a goal that has not yet been achieved. Take some time to read the two versions carefully to be sure you understand how each works. Both are equally valid and reflect choices of expression we will have to make when writing real loops. In both cases, what we write inside the loop when the keys are finally found will mean the loop conditions "flip" from *true* to *false* the next time they are evaluated.

Exercise 4.29 Suppose we express the first version of the key search in pseudo-code as follows:

```
boolean missing = true;
while(missing) {
    if(the keys are in the next place) {
        missing = false;
    }
}
```

[1] At this stage, we will ignore the possibility that the keys are not found, but taking this kind of possibility into account will actually become very important when we look at real Java examples.

Copyright © 2016. Pearson Education, Limited. All rights reserved.

Try to express the second version by completing the following outline:

```
boolean found = false;
while(...) {
    if(the keys are in the next place) {
        ...
    }
}
```

4.10.2 Iterating with an index variable

For our first while loop in correct Java code, we shall write a version of the **`listAllFiles`** method shown in Code 4.3. This does not really illustrate the indefinite character of while loops, but it does provide a useful comparison with the equivalent, familiar for-each example. The while-loop version is shown in Code 4.5. A key feature is the way that an integer variable (**`index`**) is used both to access the list's elements and to control the length of the iteration.

Code 4.5
Using a while loop to list the tracks

```
/**
 * Show a list of all the files in the collection.
 */
public void listAllFiles()
{
    int index = 0;
    while(index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
        index++;
    }
}
```

It is immediately obvious that the while-loop version requires more effort on our part to program it. Consider:

- We have to declare a variable for the list index, and we have to initialize it ourselves to 0 for the first list element. The variable must be declared outside the loop.
- We have to work out how to express the loop's condition in order to ensure that the loop stops at the right time.
- The list elements are not automatically fetched out of the collection and assigned to a variable for us. Instead, we have to do this ourselves, using the **`get`** method of the **`ArrayList`**. The variable **`filename`** will be local to the body of the loop.
- We have to remember to increment the counter variable (**`index`**) ourselves, in order to ensure that the loop condition will eventually become *false* when we have reached the end of the list.

The final statement in the body of the while loop illustrates a special operator for incrementing a numerical variable by 1:

```
index++;
```

Copyright © 2016. Pearson Education, Limited. All rights reserved.

This is equivalent to

```
index = index + 1;
```

So far, the for-each loop is clearly nicer for our purpose. It was less trouble to write, and it is safer. The reason it is safer is that it is always guaranteed to come to an end. In our while-loop version, it is possible to make a mistake that results in an *infinite loop*. If we were to forget to increment the index variable (the last line in the loop body), the loop condition would never become *false*, and the loop would iterate indefinitely. This is a typical programming error that catches even experienced programmers from time to time. The program will then run forever. If the loop in such a situation does not contain an output statement, the program will appear to "hang": it seems to do nothing, and does not respond to any mouse clicks or key presses. In reality, the program does a lot. It executes the loop over and over, but we cannot see any effect of this, and the program seems to have died. In BlueJ, this can often be detected by the fact that the red-and-white-striped "running" indicator remains on while the program appears to be doing nothing.

So what are the benefits of a while loop over a for-each loop? They are twofold: first, the while loop does not need to be related to a collection (we can loop on any condition that we can write as a boolean expression); second, even if we are using the loop to process the collection, we do not need to process every element—instead, we could stop earlier if we wanted to by including another component in the loop's condition that expresses why we would want to stop. Of course, strictly speaking, the loop's condition actually expresses whether we want to continue, and it is the negation of this that causes the loop to stop.

A benefit of having an explicit index variable is that we can use its value both inside and outside the loop, which was not available to us in the for-each examples. So we can include the index in the listing if we wish. That will make it easier to choose a track by its position in the list. For instance:

```
int index = 0;
while(index < files.size()) {
    String filename = files.get(index);
    // Prefix the file name with the track's index.
    System.out.println(index + ": " + filename);
    index++;
}
```

Having a local index variable can be particularly important when searching a list, because it can provide a record of where the item was located, which is still available once the loop has finished. We shall see this in the next section.

4.10.3 Searching a collection

Searching is one of the most important forms of iteration you will encounter. It is vital, therefore, to have a good grasp of its essential elements. The sort of loop structures that result occur again and again in practical programming situations.

The key characteristic of a search is that it involves *indefinite iteration*; this is necessarily so, because if we knew exactly where to look, we would not need a search at all! Instead, we have to initiate a search, and it will take an unknown number of iterations before we

Copyright © 2016. Pearson Education, Limited. All rights reserved.

succeed. This implies that a for-each loop is inappropriate for use when searching, because it will complete its full set of iterations.[2]

In real search situations, we have to take into account the fact that the search might fail: we might run out of places to look. That means that we typically have two finishing possibilities to consider when writing a searching loop:

- The search succeeds after an indefinite number of iterations.
- The search fails after exhausting all possibilities.

Both of these must be taken into account when writing the loop's condition. As the loop's condition should evaluate to *true* if we want to iterate one more time, each of the finishing criteria should, on their own, make the condition evaluate to *false* to stop the loop.

The fact that we end up searching the whole list when the search fails does not turn a failed search into an example of definite iteration. The key characteristic of definite iteration is that you can determine the number of iterations *when the loop starts*. This will not be the case with a search.

If we are using an index variable to work our way through successive elements of a collection, then a failed search is easy to identify: the index variable will have been incremented beyond the final item in the list. That is exactly the situation that is covered in the **listAllFiles** method in Code 4.5, where the condition is:

```
while(index < files.size())
```

The condition expresses that we want to continue as long as the index is within the valid index range of the collection; as soon as it has been incremented out of range, then we want the loop to stop. This condition works even if the list is completely empty. In this case, **index** will have been initialized to zero, and the call to the **size** method will return zero too. Because zero is not less than zero, the loop's body will not be executed at all, which is what we want.

We also need to add a second part to the condition that indicates whether we have found the search item yet and stops the search when we have. We saw in Section 4.10.1 and Exercise 4.29 that we can often express this positively or negatively, via appropriately set boolean variables:

- A variable called **searching** (or **missing**, say) initially set to *true* could keep the search going until it is set to *false* inside the loop when the item is found.
- A variable called **found**, initially set to *false* and used in the condition as **!found**, could keep the search going until set to *true* when the item is found.

Here are the two corresponding code fragments that express the full condition in both cases:

```
int index = 0;
boolean searching = true;
while(index < files.size() && searching)
```

[2] While there are ways to subvert this characteristic of a for-each loop, and they are used quite commonly, we consider them to be bad style and do not use them in our examples.

Copyright © 2016. Pearson Education, Limited. All rights reserved.

or

```
int index = 0;
boolean found = false;
while(index < files.size() && !found)
```

Take some time to make sure you understand these two fragments, which accomplish exactly the same loop control, but expressed in slightly different ways. Remember that the condition *as a whole* must evaluate to *true* if we want to continue looking, and *false* if we want to stop looking, for any reason. We discussed the "and" operator **&&** in Chapter 3, which only evaluates to *true* if *both* of its operands are *true*.

The full version of a method to search for the first file name matching a given search string can be seen in Code 4.6 (*music-organizer-v4*). The method returns the index of the item as its result. Note that we need to find a way to indicate to the method's caller if the search has failed. In this case, we choose to return a value that cannot possibly represent a valid location in the collection—a negative value. This is a commonly used technique in search situations: the return of an *out-of-bounds* value to indicate failure.

Code 4.6
Finding the first matching item in a list

```
/**
 * Find the index of the first file matching the given
 * search string.
 * @param searchString The string to match.
 * @return The index of the first occurrence, or -1 if
 *         no match is found.
 */
public int findFirst(String searchString)
{
    int index = 0;
    // Record that we will be searching until a match is found.
    boolean searching = true;

    while(searching && index < files.size()) {
        String filename = files.get(index);
        if(filename.contains(searchString)) {
            // A match. We can stop searching.
            searching = false;
        }
        else {
            // Move on.
            index++;
        }
    }
    if(searching) {
        // We didn't find it.
        return -1;
    }
    else {
        // Return where it was found.
        return index;
    }
}
```

Copyright © 2016. Pearson Education, Limited. All rights reserved.

It might be tempting to try to have just one condition in the loop, even though there are two distinct reasons for ending the search. A way to do this would be to artificially arrange for the value of index to be too large if we find what we are looking for. This is a practice we discourage, because it makes the termination criteria of the loop misleading, and clarity is always preferred.

4.10.4 Some non-collection examples

Loops are not only used with collections. There are many situations were we want to repeat a block of statements that does not involve a collection at all. Here is an example that prints out all even numbers from 0 up to 30:

```
int index = 0;
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
```

In fact, this is the use of a while loop for definite iteration, because it is clear at the start how many numbers will be printed. However, we cannot use a for-each loop, because they can only be used when iterating over collections. Later we will meet a third, related loop—the *for loop*—that would be more appropriate for this particular example.

To test your own understanding of while loops aside from collections, try the following exercises.

Exercise 4.30 Write a while loop (for example, in a method called **multiplesOfFive**) that prints out all multiples of 5 between 10 and 95.

Exercise 4.31 Write a while loop to add up the values 1 to 10 and print the sum once the loop has finished.

Exercise 4.32 Write a method called **sum** with a while loop that adds up all numbers between two numbers **a** and **b**. The values for **a** and **b** can be passed to the **sum** method as parameters.

Exercise 4.33 *Challenge exercise* Write a method **isPrime(int n)** that returns *true* if the parameter **n** is a prime number, and *false* if it is not. To implement the method, you can write a while loop that divides **n** by all numbers between **2** and **(n–1)** and tests whether the division yields a whole number. You can write this test by using the modulo operator (**%**) to check whether the integer division leaves a remainder of 0 (see the discussion of the modulo operator in Section 3.8.3).

Exercise 4.34 In the **findFirst** method, the loop's condition repeatedly asks the **files** collection how many files it is storing. Does the value returned by **size** vary from one check to the next? If you think the answer is no, then rewrite the method so that the number of files is determined only once and stored in a local variable prior to execution of the loop. Then use the local

Copyright © 2016. Pearson Education, Limited. All rights reserved.