We shall find that the **`ArrayList`** class makes it very easy to provide this functionality from our own class.

Notice that we are not being too ambitious in this first version. These features will be sufficient for illustrating the basics of creating and using the **`ArrayList`** class, and later versions will then build further features incrementally until we have something more sophisticated. (Most importantly, perhaps, we will later add the possibility of playing the music files. Our first version will not be able to do that.) This modest, incremental approach is much more likely to lead to success than trying to implement everything all at once.

Before we analyze the source code needed to make use of such a class, it is helpful to explore the starting behavior of the music organizer.

Exercise 4.1 Open the *music-organizer-v1* project in BlueJ and create a **`MusicOrganizer`** object. Store the names of a few audio files into it—they are simply strings. As we are not going to play the files at this stage, any file names will do, although there is a sample of audio files in the *audio* folder of the chapter that you might like to use.

Check that the number of files returned by **`numberOfFiles`** matches the number you stored. When you use the **`listFile`** method, you will need to use a parameter value of **`0`** (zero) to print the first file, **`1`** (one) to print the second, and so on. We shall explain the reason for this numbering in due course.

Exercise 4.2 What happens if you create a new **`MusicOrganizer`** object and then call **`removeFile(0)`** before you have added any files to it? Do you get an error? Would you expect to get an error?

Exercise 4.3 Create a **`MusicOrganizer`** and add two file names to it. Call **`listFile(0)`** and **`listFile(1)`** to show the two files. Now call **`removeFile(0)`** and then **`listFile(0)`**. What happened? Is that what you expected? Can you find an explanation of what might have happened when you removed the first file name from the collection?

4.4 Using a library class

Code 4.1 shows the full definition of our **`MusicOrganizer`** class, which makes use of the library class **`ArrayList`**. Note that library classes do not appear in the BlueJ class diagram.

Class libraries One of the features of object-oriented languages that makes them powerful is that they are often accompanied by *class libraries.* These libraries typically contain many hundreds or thousands of different classes that have proved useful to developers on a wide range of different projects. Java calls its libraries *packages.* Library classes are used in exactly the same way as we would use our own classes. Instances are constructed using **`new`**, and the classes have fields, constructors, and methods.

Code 4.1
The **Music-Organizer** class

```
import java.util.ArrayList;

/**
 * A class to hold details of audio files.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class MusicOrganizer
{
    // An ArrayList for storing the file names of music files.
    private ArrayList<String> files;

    /**
     * Create a MusicOrganizer
     */
    public MusicOrganizer()
    {
        files = new ArrayList<>();
    }

    /**
     * Add a file to the collection.
     * @param filename The file to be added.
     */
    public void addFile(String filename)
    {
        files.add(filename);
    }

    /**
     * Return the number of files in the collection.
     * @return The number of files in the collection.
     */
    public int getNumberOfFiles()
    {
        return files.size();
    }

    /**
     * List a file from the collection.
     * @param index The index of the file to be listed.
     */
    public void listFile(int index)
    {
        if(index >= 0 && index < files.size()) {
            String filename = files.get(index);
            System.out.println(filename);
        }
    }
```

Code 4.1 continued
The `Music-Organizer` class

```
    /**
     * Remove a file from the collection.
     * @param index The index of the file to be removed.
     */
    public void removeFile(int index)
    {
        if(index >= 0 && index < files.size()) {
            files.remove(index);
        }
    }
}
```

4.4.1 Importing a library class

The very first line of the class file illustrates the way in which we gain access to a library class in Java, via an *import statement*:

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

This makes the **`ArrayList`** class from the **`java.util`** package available to our class definition. Import statements must always be placed before class definitions in a file. Once a class name has been imported from a package in this way, we can use that class just as if it were one of our own classes. So we use **`ArrayList`** at the head of the **`MusicOrganizer`** class to define a **`files`** field:

```
private ArrayList<String> files;
```

Here, we see a new construct: the mention of **`String`** in angle brackets: **`<String>`**. The need for this was alluded to in Section 4.3, where we noted that **`ArrayList`** is a *general-purpose* collection class—i.e., not restricted in what it can store. When we create an **`ArrayList`** object, however, we have to be specific about the type of objects that will be stored in that particular instance. We can store whatever type we choose, but we have to designate that type when declaring an **`ArrayList`** variable. Classes such as **`ArrayList`**, which get parameterized with a second type, are called *generic classes* (we will discuss them in more detail later).

When using collections, therefore, we always have to specify two types: the type of the collection itself (here: **`ArrayList`**) and the type of the elements that we plan to store in the collection (here: **`String`**). We can read the complete type definition **`ArrayList<String>`** as "*ArrayList of String*." We use this type definition as the type for our **`files`** variable.

As you should now have come to expect, we see a close connection between the body of the constructor and the fields of the class, because the constructor is responsible for initializing the fields of each instance. So, just as the **`ClockDisplay`** created **`NumberDisplay`** objects for its two fields, here we see the constructor of the **`MusicOrganizer`** creating an object of type **`ArrayList`** and storing it in the **`files`** field.

4.4.2 Diamond notation

Note that when creating the **`ArrayList`** instance, we have written the following statement:

```
files = new ArrayList<>();
```