Chapter 12. Lists and Files

This chapter discusses two distinct but related issues: lists and files. *Lists* are similar to arrays, but they can grow or shrink during the execution of a program. This can be helpful in situations where the programmer cannot know the number of list elements that need to be represented, say because each user adds them at their own discretion. It is sometimes convenient to store such elements permanently on the computer rather than by expecting the user to enter them manually or by hard-coding them into the program. Languages use *files* for this purpose. This chapter introduces the use of lists and files in Java.

12.1. Example: An Interactive Quiz Program

In this chapter, the goal is to design and implement an interactive quiz or drill program that helps people learn visually-oriented materials. The particular domain will be Chinese characters. College-level readers of Chinese are generally believed to recognize from 3000-5000 characters, so learners of this language have a big character recognition task ahead of them. Paper flash cards are often used, but producing and maintaining stacks of these cards is challenging. This chapter sets the goal of developing a character drill program such as the one shown in Figure 12-1.

This application should maintain a list of Chinese characters, along with their PinYin pronunciation and English translation, and present the characters to the user one at a time in random order, giving them hints as needed. The user could press the “Give up” button, which would present the answer and move on to another character. It would be ideal if the user could add and/or subtract characters as needed over time without having to reprogram the application.

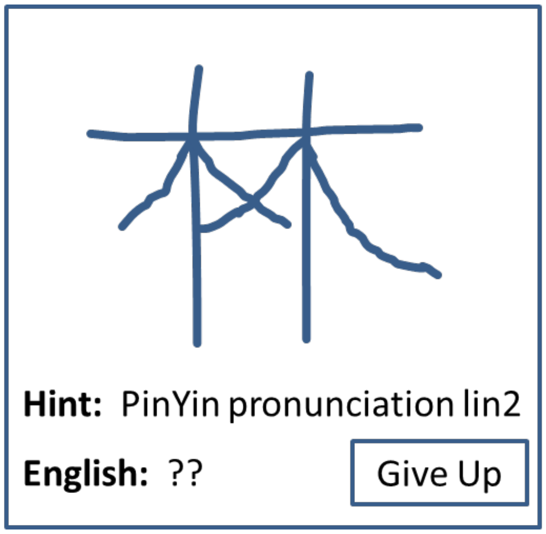


Figure 12-1. Character drill application sketch

We’ll attempt to design this application in such a way that it can be used for quizzes and drill programs in other domains that require visual recognition of lists of objects, e.g., identifying works of art and their creators, recognizing people’s faces, etc.

This chapter discusses the Java-based list and file technologies required to implement this vision. It uses a Java-based graphical user interface with an integrated Processing panel to present the images.

12.2. ArrayLists

In Chapter 8, we learned about Java’s *array* mechanism for storing sequences of values. An array that can hold `capacity` values with a specified `type` can be created with a statement of the form

```
type [] anArray = new type [ capacity ];
```

A drawback to this mechanism is that once allocated, the capacity of the array is fixed and cannot be changed. This implies that if we wish to store a sequence of values in an array, we must know approximately how many values there are in the sequence before we allocate the array. Otherwise:

• If the capacity of the array exceeds the number of values to be stored in it, then memory is wasted by the unused elements.
• If the capacity of the array is smaller than the number of values to be stored in it, then the problem of array overflow may occur.

To circumvent this inconvenience, the Java class hierarchy provides a variety of **collection classes**. Like an array, each collection class can store a group of values. Unlike an array, however, a collection can grow and shrink as a program runs. In this chapter, we examine one of these classes: `ArrayList`.

12.2.1. Collection Classes

The ArrayList is actually only one example of data structures provided by Java for storing collections of objects. Java provides three different *collection classes*:

- **Lists**, that store collections of objects, some of which may be the same
- **Sets**, that store collections of objects, with no duplicates
- **Maps**, that store collections of pairs, each of which associates a *key* with an *object*.

In the Java API, `List`, `Set` and `Map` are all *interfaces* that are implemented by other classes. The Java API describes each of these collection interfaces in depth, as well as the classes that implement them. In this section, we focus on implementations of the `List` interface as implemented by the `ArrayList` class. A few of the methods that must be defined to implement a `List` are given here:

- `add(object)` – Append *object* to the end of the given list.
- `get(index)` – Return the object at position *index* in the given list.
- `isEmpty()` – Return true if the given list is empty.
- `size()` – Return the size of the given list.

More information on these and other methods supported by the `List` interface, as well the methods supported by the other collection classes can be found in the Java API specification.[1] In this section, we focus on the `ArrayList` implementation of the `List` interface.

12.2.2. Introducing the `ArrayList` Class

As its name implies, `ArrayList` is a class that implements the `List` interface using an array. As such, it implements each of the `List` operations using an array of objects to store the collection's values. It also adds additional operations that are described in the API documentation.

[1] See http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html.

By using an `Object` array, an `ArrayList` can store *any* reference type, since all reference types are classes that (directly or indirectly) extend `Object`. The down side of this is that an `ArrayList` cannot *directly* store the primitive types (`int`, `char`, `double`, etc.), although it can *indirectly* store them by storing instances of their wrapper classes (`Integer`, `Character`, `Double`, etc.).

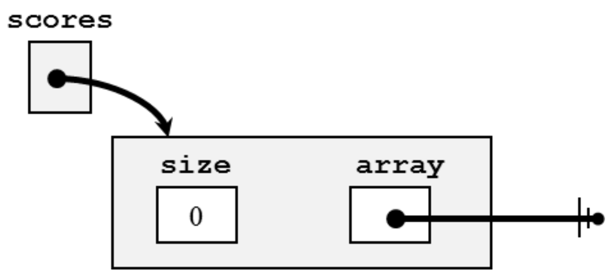
Consider the problem of reading in a list of integers scores and computing the standard deviation of those numbers. The formula for computing this quantity for a population of *n* values is as follows:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i + \mu)^2}$$

Because μ represents the average of the values, a program must make two passes over the data, once to compute the average μ and a second time to compare each value with μ as specified in the equation. Thus, the program must store the values in a suitable data structure. If the program knows the value of *n*, the size of the population, it can declare an array of integers of that size. If it does not know this value, it can use a List as shown here:

```
ArrayList<Integer> scores = new ArrayList<Integer>();
```

This statement declares and initializes a new `ArrayList` of integers. No size is specified. Note that the specification uses brackets (< & >) to indicate that the list contains objects of type `Integer`, where `Integer` is the reference type for integers; lists cannot contain primitive types like `int`. This statement defines `scores` as a reference to an empty `ArrayList` object that can be visualized as follows:

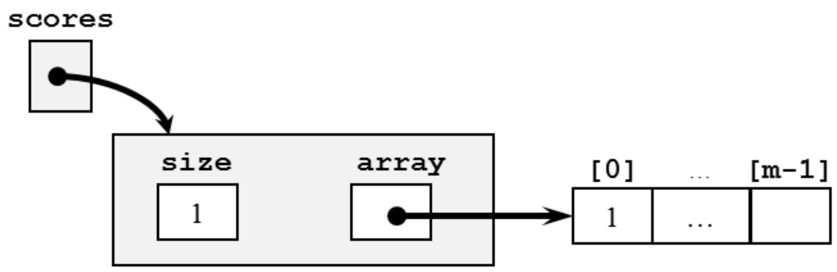


12.2.3. Adding Elements to a List

This list contains no integer objects initially, but the `add()` method can be used to add elements to the list, automatically allocating memory space for them as necessary. For example, if a program executes the following command:

```
scores.add(1);
```

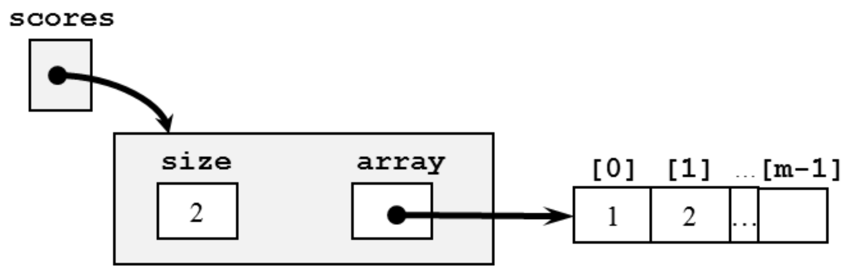
the `ArrayList` object allocates an array of some (implementation-dependent) size `m`, links the array component of the `ArrayList` to this new array and updates the size. The result can be visualized as follows:



Adding additional integers, up to the array capacity of `m` triggers no new memory allocation, but the value of size is incremented as each new item is added. For example, if the program continues with the following command:

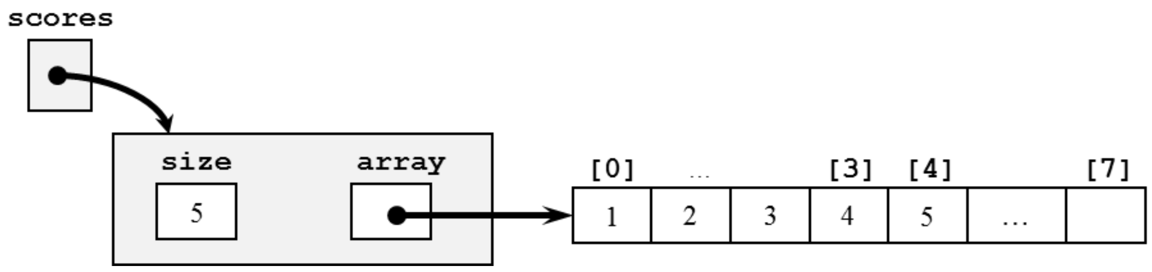
```
scores.add(2);
```

A new value, 2, is added to the second element of the array, which has already been allocated, and the size is incremented. The result can be visualized as follows:



This is all done automatically by the `ArrayList` object; the program simply adds element after element, letting the `ArrayList` deal with representation details.

When more array space is needed, the ArrayList object will add it automatically. For example, if the default array allocation were four elements and the user added a fifth element, e.g., 5, the `ArrayList` object would allocate a larger internal array. The result can be visualized as follows:



In this structure, four more array elements have been added and one of them has been used to store the fifth element of the `ArrayList`. The size value has also been updated to represent the number of elements in the `ArrayList`. Accomplishing this is actually a 3-step process:

1) A new array of size *`2m-1`* is allocated (m=4 in the example)
2) The values from the old array are (shallow) copied into the new array
3) The old array is replaced by the new array

Because all of this is carried out "behind the scenes" by the `add()` method, the details of how it is accomplished need not concern us here.

12.2.4. Accessing List Elements

The get() method accesses list elements at given indexes. For example, the following method call:

```
scores.get(2)
```

retrieves the value 3 from the list. Note that accessing list elements does not use the square brackets notation (`[]`) used to access array elements.

12.2.5. Using Lists as Arguments and Parameters

As with arrays, lists can be passed as arguments and declared as parameters. The following program allows a user to enter as many integer scores as they would like and then computes the average and variance of the data.

Code:

```
// This method assumes a non-null, non-empty list of values
public static double computeAverage(ArrayList<Integer> values) {
      int sum = 0;
      for (int i = 0; i < values.size(); i++) {
            sum += values.get(i);
      }
      return sum / values.size();
}

// The computeVariance() method is left as an exercise.

public static void main(String[] args) throws Exception {
      ArrayList<Integer> scores = new ArrayList<Integer>();
      Scanner keyboard = new Scanner(System.in);
      int score;
      System.out.println("Please enter scores (-1 to quit).");
      while (true) {
            System.out.print("\tScore: ");
            score = keyboard.nextInt();
            if (score == -1) {
                  break;
            }
            scores.add(score);
      }
      double average = computeAverage(scores);
      System.out.println("Average: " + average);
      System.out.println("Variance: " + computeVariance(scores, average));
}
```

Interaction:

```
Please enter scores (-1 to quit).
        Score: 1
        Score: 2
        Score: 3
        Score: 4
        Score: 5
        Score: -1
Average: 3.0
Variance: 1.4142135623730951
```

This code segment declares and fills the scores `ArrayList` as discussed above. Note that the user is allowed to enter as many scores and they would like, without having to indicate the total number of scores at the beginning. It then passes that `ArrayList` to the `computeAverage()` method, which accesses the elements of the `ArrayList` to compute the average. The `computeVariance()` method uses a similar approach to compute the standard deviation.

12.2.6. Working with Lists of Records

The same automatic behavior is implemented for `ArrayLists` of other object types. For example, consider creating a collection of Soldier objects, each implemented using the following Soldier class:

Soldier.java

```java
public class Soldier {

      private String myName, myRank, mySerialNumber;

      public Soldier(String line) {
            Scanner scanner = new Scanner(line);
            myName = scanner.next();
            myRank = scanner.next();
            mySerialNumber = scanner.next();
      }

      public String toString() {
            String result = myName + " " + myRank + " " + mySerialNumber;
            return result;
      }

}
```

Given this class, the following code creates an `ArrayList` of `Soldier` records, each of which represents the soldier's name, rank and serial number.

Code:

```java
ArrayList<Soldier> soldiers = new ArrayList<Soldier>();
soldiers.add(new Soldier("Flagg", "Colonel", "1234567"));
soldiers.add(new Soldier("Freedman", "Major", "2345678"));
soldiers.add(new Soldier("Kellye", "Lieutenant", "3456789"));
System.out.println(soldiers);
```

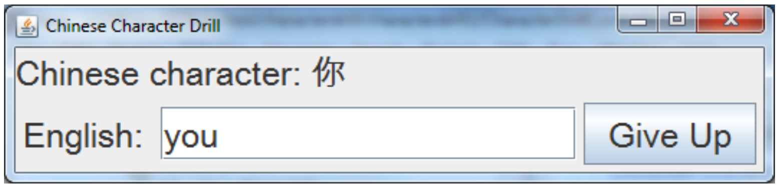
Output:

```
[Flagg Colonel 1234567, Freedman Major 2345678, Kellye Lieutenant 3456789]
```

This code creates and loads its `ArrayList` of class objects in the same way that the earlier example created and loaded its `ArrayList` of integers. The program is free to add as many soldier records as required. The default output for an `ArrayList` is to list the values within square braces (`[]`), printing each record according to its own `toString()` definition. In this implementation, `Soldier` records are programmed to display themselves by concatenating their name, rank and then serial number.

12.2.7. Revisiting the Example

The `ArrayList` collection class will allow our Chinese character drill program to include as many characters as required, without having to know the total number in advance. The first iteration of this application, shown below, implements a text-only version of the quiz program that presents the Unicode representation of the character and asks the user to enter the English translation of that character.



We'll look more closely at the GUI implementation later in the chapter. For now, we'll examine the `Character` and `CharacterDrill` classes in more detail.

Character.java

```
/**
 * Character models a single Chinese character, with English translation,
 * PinYin transcription and Chinese character (in Unicode).
 *
 * @author kvlinden
 * @version Spring, 2012
 */
public class Character {

      private String myEnglish, myPinyin, myCharacter;

      public Character(String english, String pinyin, String character) {
            myEnglish = english;
            myPinyin = pinyin;
            myCharacter = character;
      }

      public String getEnglish() {
            return myEnglish;
      }

      public String getPinyin() {
            return myPinyin;
      }

      public String getCharacter() {
            return myCharacter;
      }

}
```

This class is much like other classes built in the text before with the exception that `myCharacter` must be specified using Unicode[2]. This character is still represented as a string object here in the `Character` class, but when creating new `Character` objects, which we'll see in a moment, the String argument to the explicit-value constructor must use the character escape sequence `\u`.

The class that implements the actual drill/quiz program is shown here.

CharacterDrill.java

[2] For more details on Unicode, see http://unicode.org/.

```java
/**
 * CharacterDrill manages a simple quiz with hints based on a manually-
 * created list of Chinese characters.
 *
 * @author kvlinden
 * @version Spring, 2012
 */
public class CharacterDrill {

      private List<Character> myCharacters;
      private int myAnswerIndex, myHintCount;

      private Random myRandom;

      public CharacterDrill() {
            myCharacters = new ArrayList<Character>();
            myCharacters.add(new Character("I", "wo3", "\u6211"));
            myCharacters.add(new Character("you", "ni3", "\u4F60"));
            myCharacters.add(new Character("he", "ta1", "\u4ED6"));
            myCharacters.add(new Character("she", "ta1", "\u5979"));
            myCharacters.add(new Character("it", "ta1", "\u5B83"));
            myRandom = new Random();
            reset();
      }

      public void reset() {
            myAnswerIndex = myRandom.nextInt(myCharacters.size());
            myHintCount = 0;
      }

      public boolean guess(String guess) {
            return myCharacters.get(
                        myAnswerIndex).getEnglish().equalsIgnoreCase(guess);
      }

      public String getAnswer() {
            return myCharacters.get(myAnswerIndex).getEnglish();
      }

      public String getHintText() {
            myHintCount++;
            if (myHintCount == 1) {
                  return "Translate this character.";
            } else if (myHintCount == 2) {
                  return "PinYin: " +
                              myCharacters.get(myAnswerIndex).getPinyin();
            } else {
                  return "No more hints...";
            }
      }
}
```

This code uses an ArrayList to represent a list of characters for the drill. It declares this ArrayList in the default constructor, manually loads a collection of five characters and drives the test by selecting a random entry from this `ArrayList` for each round of the test. The Unicode representations of the five text characters are shown in lines 17-21.

Random behavior is implemented in Java using the `Random` class[3]. This program declares an object of this type in line 13 and initializes it in line 22. Each time the drill is reset, `myAnswerIndex` is set to a new random number between 0 and the `size()` of the `myCharacters` `ArrayList`, see line 27.

Hints are doled out by the `getHintText()` method based on `myHintCount`, which represents the number of hints that have been given out so far.

12.3. Files

Many computer users have had the unfortunate experience of having their word processor (or text editor) unexpectedly fail while they were editing a document. This is especially annoying because all of the information entered since the last save operation is lost. This happens because a word processor is an executable program, and the information being edited (documents, programs, input data, etc.) is stored in the section of main memory allocated to the word processor. When the word processing program terminates, this memory is deallocated and its contents are lost.

To minimize this problem, many word processors and text editors provide an *autosave* feature that periodically saves the information to secondary memory to minimize the amount of information lost should a power outage occur. Examples of secondary memory include hard disks, optical disks, CD-ROM, and USB drives.

Information that is saved in secondary memory must be stored in such a way that:

- It can be retrieved in the future
- It is kept separate from all other documents, programs, and so on, that are saved

To achieve these goals, secondary memory is organized into distinct containers called **files**, in which information can be stored. When a document must be edited, the word processor *loads* it from secondary memory to main memory by *reading* from the file containing the document. The operation of *saving* information to secondary memory involves *writing* the document to a file.

In addition to providing a stable place to store *documents* and *programs*, files can also be used to store *data*. If a large set of data values is to be processed, then those values can be stored in a file, and a program can be written to read these values from the file and process them. This capability is especially useful in testing and debugging a program because the data does not have to be reentered each time the program is executed.

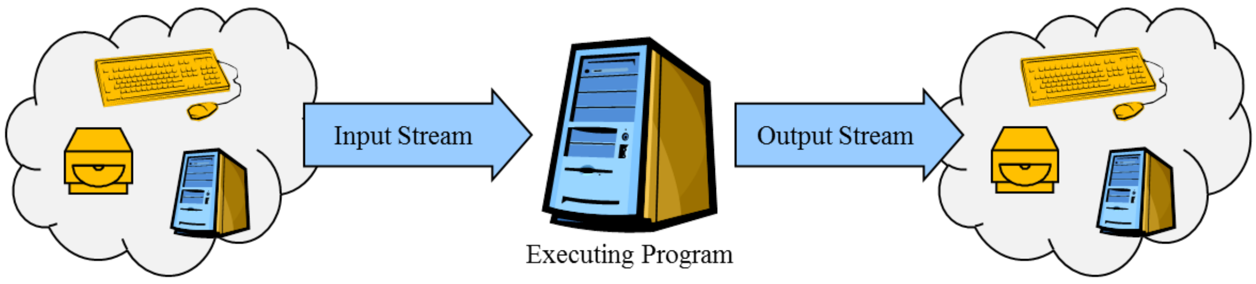
Files can be classified by the kind of data stored in them. Those that contain textual characters (such as the source code for a program, numbers entered with a text editor, etc.) are called **text files**. By contrast, files that contain non-textual characters (such as the binary code for a compiled program, or the control codes for a word processor) are called **binary files**. In this chapter, we will discuss Java's support for input and output using both kinds of files.

[3] See http://docs.oracle.com/javase/7/docs/api/java/util/Random.html.

12.3.1. Streams

There are many ways for a Java program to receive input and to produce output, each with their own strengths and weaknesses. In this section we go into more detail about the underlying principles of Java's Input/Output (I/O) classes.

All input and output in Java is accomplished using classes that are collectively known as *streams*. As the name implies, a Java stream is an abstraction for a means of *moving information* between a program and a device – a disk, a keyboard, a screen or window, and so on.



There are several ways that streams can be categorized. Perhaps the most basic division is between **input streams** and **output streams**. An input stream provides support for moving information from an input device to a program, and an output stream provides support for transferring information from a program to an output device.

The `System` class in the `java.lang` package provides three public class variables that are streams:

- **`System.in`** is an `InputStream` object associated with "standard input" (usually the computer's keyboard)
- **`System.out`** is a buffered `PrintStream` object associated with "standard output" (usually the computer's screen or active window)
- **`System.err`** is an unbuffered `PrintStream` object associated with "standard error" (usually) the computer's screen or console window).

The programs written in the text have made frequent use of the first two of these streams. When a program declares a new Scanner object to read user input from the keyboard, a declaration such the following is used:

```
Scanner keyboard = new Scanner(System.in);
```

Note that this code references the predefined `System.in` object, which refers to the keyboard. The `Scanner` class implements some useful text-scanning features to this basic system stream, e.g., `nextInt()` and `nextLine()`, but, as we will see in the next section, there are additional useful features that can be added.

Similarly, when a program prints text to the output console, a command such as the following is used:

```
System.out.println("Hello, world!");
```

12.3.2. Reading from a File

The `File` class provides a means of building a stream from a file to our program by sending the name of the file to its constructor[4]:

```
new File(filename)
```

This statement constructs a stream to our input file that we can use to read characters from the input file. Unfortunately, this operation is not very efficient because it is not buffered. *Buffering* I/O improves program performance. Without buffering, each read operation could cause values to be read directly from the disk and then returned to the program. However, disk accesses can take thousands of times as long as a typical CPU operation. With buffering, values are obtained from the disk in advance and stored in a section of main memory called a *buffer*, so that calls to read operations can obtain these values from main memory, which takes much less time than for disk accesses.

Fortunately, Java's `Scanner` class provides not only a set of useful reading features, but it also provides buffering. Thus, we can declare a `Scanner` class that "wraps" the `File` class as follows:

```
Scanner fileIn = new Scanner(new File(filename));
```

This nested statement declares and initializes a file input stream, identified by `fileIn`, that provides buffered read access to the specified file. We can now use this input file stream to read data from the given file in the same way that we have read input from the keyboard in previous programs.

When we are finished inputting the file data, it is a good practice to close the stream to the file using the close command.

```
fileIn.close();
```

Because streams are automatically closed when a program terminates, this is not strictly necessary for small applications, but larger applications where potentially many files may be open at once, require more careful management of the limited number of open streams supported by the operating system. Thus, it is good to make a practice of closing file streams when we are finished with them.

For example, imagine that we have stored a list of integer quiz scores in a text file named `scores.txt` that looks like this:

```
1
2
3
4
5
```

[4] Notice that in this example we are passing the filename as an argument, instead of a `File` object created from the filename. Either is acceptable since there are multiple constructors for the `FileReader` class, one of which takes a string as an argument, and one which takes a `File` object.

Given the existence of this file, the following code reads the values and computes their average.

```
List<Integer> scores = new ArrayList<Integer>();
Scanner fileIn = new Scanner(new File("scores.txt"));
while (fileIn.hasNext()) {
      scores.add(fileIn.nextInt());
}
fileIn.close();
System.out.println("average: ", computeAverage(scores));
```

This program declares an `ArrayList` of integers, opens a stream to the scores file, reads the values one-by-one and then computes the average (3.0). Note that auto-growing nature of the `ArrayList` works well when reading a file whose length is unknown. When `fileIn` has no more values, that is when `fileIn.hasNext()` is `false`, the loop terminates and the program closes the file.

12.3.3. Writing to a File

The `PrintWriter` class provides buffered output of primitive and `String` values using `print()` and `println()`. In the output stream context, buffering means that values to be written to a disk are temporarily stored in a buffer (in main memory), and are actually written to the disk at some later time when it is more efficient to do so. This means that not every call to `print()` or `println()` generates a disk access, which is potentially much faster since each print operation requires much more time than typical CPU operations. `PrintWriter` only writes its buffer to the disk when the `close()` method is called or the buffer becomes full.[5] Opening a file for write access in this manner automatically deletes the data that was originally in the file.

The following program demonstrates the process of writing to a file.

Code:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Data Records filename: ");
String filename = keyboard.nextLine();

PrintWriter fileOut = new PrintWriter(new File(filename));
System.out.println("Please enter scores (enter to quit).");
String line;
while (true) {
      System.out.print("\tScore: ");
      line = keyboard.nextLine();
      if (line.equals("")) {
            break;
      }
      fileOut.println(line);
}
fileOut.close();
System.out.println("File created: " + filename);
```

[5] The `PrintWriter` then writes the contents of the buffer to disk to make room for more values. Also, the `flush()` message can be used to tell a `PrintWriter` to write its buffer to disk immediately.

Interaction:

```
Data Records filename: src/c12lists/text/integerStatistics/output.txt
Please enter scores (enter to quit).
      Score: 1
      Score: 2
      Score: 3
      Score: 4
      Score: 5
      Score:
File created: src/c12lists/text/integerStatistics/output.txt
```

output.txt:

```
1
2
3
4
5
```

This program is similar to the score reading program discussed in the previous section, but rather than filling an `ArrayList`, it writes the user's input to the specified file. It reads the name of the file from the keyboard, opens a stream to that file and then writes each input from the user directly to the file. The presentation given above shows the code, the user interaction and the resulting file.

12.3.4. Working with Records

File I/O works for records just as it does for primitive values, where a record is a set of values associated with a single object. A program can read the information for a set of `Soldier` objects using the Soldier class discussed in the previous section. Each soldier record has a name, rank and serial number, all represented by `String` values.

Consider the following file, which contains one row for each Soldier object:

```
Flagg Colonel 1234567 Sammy
Freedman Major 2345678
Kellye Lieutenant 3456789 Able Kealani
```

The lines of this file specify the soldier's name, rank serial number and then a possibly empty list of nicknames. For example, Colonel Flagg is sometimes known as "Sammy". Major Freedman has no nicknames.

The following code reads the soldier information from the given file.

```
List<Soldier> soldiers = new ArrayList<Soldier>();
Scanner fileIn = new Scanner(new File(filename));
while (fileIn.hasNext()) {
      soldiers.add(new Soldier(fileIn.nextLine()));
}
fileIn.close();

System.out.println(soldiers);
```

This file-reading code works much the same way as the scores file-reading code discussed above except that it doesn't attempt to parse the records on each line of the file. Instead, it passes the full line (`fileIn.nextLine()`) unparsed to the `Soldier` class constructor method. The philosophy here is that the `Soldier` class knows what information it needs and how that information should be formatted in the file, so that operation is implemented in the `Soldier` class. The relevant constructor method is shown here:

```
public Soldier(String line) {
      Scanner scanner = new Scanner(line);
      myName = scanner.next();
      myRank = scanner.next();
      mySerialNumber = scanner.next();
      myNickNames = new ArrayList<String>();
      while (scanner.hasNext()) {
            myNickNames.add(scanner.next());
      }
      scanner.close();

}
```

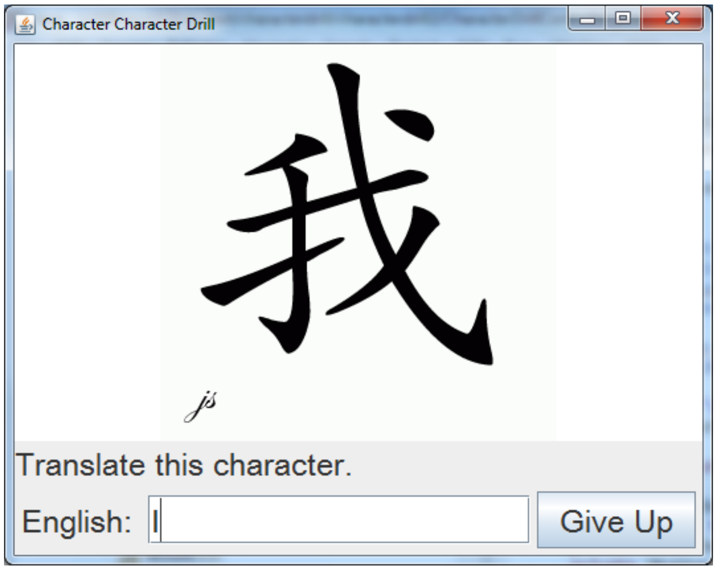
Note the following about this explicit-value constructor for the `Soldier` class:

- This method receives a string already read from the file (`line`) and scans it using a `Scanner` object constructed for that line (see `new Scanner(line)`). Note that we have used the `Scanner` to parse input from the keyboard, to parse input from files and now to parse individual string values. In all these cases, the `Scanner` receives a string of characters, either from the `System.in` stream, a file stream or a simple `String` object, and scans across those characters.
- The constructor is not very picky about its input values; it takes whatever it reads from the file without question. This is not a very wise choice given the volatility of files and errors that they can contain.
- The list of nicknames, potentially empty, is represented as an `ArrayList` of Strings. This constructor constructs an empty list and scans each nickname individual as appropriate. If there are no nicknames on the input line, then this `ArrayList` remains empty.

A program capable of reading soldier information from the keyboard and writing it to a suitably formatted file is relatively simple and left as an exercise for the reader.

12.3.5. Revisiting the Example

Files can give our Character Drill application the ability to load its test characters from a file, rather than hard-coding them into the program as was done in the previous iteration (see Section 12.2.7). An enhanced version of the program is shown here.



As can be seen here, this GUI implementation presents a full image of the Chinese character hand-drawn with higher-quality calligraphy. It stores the English translation, PinYin pronunciation and image filename (with full file path) as shown here:

```
I wo3 src/c12lists/text/characterdrill/data/wo3.gif
you ni3 src/c12lists/text/characterdrilldata/ni3.gif
he ta1 src/c12lists/text/characterdrilldata/ta1he.gif
she ta1 src/c12lists/text/characterdrill/data/ta1she.gif
it ta1 src/c12lists/text/characterdrill/data/ta1it.gif
```

The Character class is largely the same as it was in the last iteration with the exception that it now stores the name of the image file presenting the character’s calligraphy. The CharacterDrill class is also largely the same as it was with the exception that it now reads the character object’s information from a file; the method that implements this in the `CharacterDrill` class is shown here:

```
private List<Character> loadCharacters(String filename)
                                    throws FileNotFoundException {
      Scanner fileIn = new Scanner(new File(filename));
      List<Character> result = new ArrayList<Character>();
      while (fileIn.hasNext()) {
            result.add(new Character(fileIn.next(),
                                     fileIn.next(),
                                     fileIn.next()));
      }
      fileIn.close();
      return result;

}
```

Note that the method shown here can throw a `FileNotFoundException`, which is the exception class thrown by the `File` constructor when it cannot find the named file on the system. The remainder of the code is similar to the file-reading code discussed in Section 12.2.2.

A new is added to support the canvas required to present the character image. This panel is a rather unremarkable reuse of techniques presented in Chapter 11 with the exception that it uses *tinting* to fade the characters in and out of view. This is implemented in the `draw()` method as follows:

```
	public void draw() {
		if (myImage != null) {
			tint(255, myTintValue);
			image(myImage, width / 2, height / 2);
			myTintValue = constrain(myTintValue + 1, 0, 255);
		}

	}
```

The `CharacterPanel` initializes `myTintValue` to 0, uses it to set the tint value of the image being presented and then increments the tint value. Each time a new image is to be presenting, its image file is loaded and `myTintValue` is reset to 0.

The GUI itself is a bit more involved that the GUI in the previous iteration in that it is using a Processing panel to display the character image and it is fading each image in and out in pleasing manner. This behavior is implemented using the following GUI controller.

CharacterDrillController.java

```
/**
 * ChineseDrillController implements a simple Chinese character review
 * application. This version adds file handling and image files, but
 * drops support for Unicode character strings (because they are harder
 * to read from simple text files).
 *
 * @author kvlinden
 * @version Spring, 2012
 */
public class CharacterDrillController extends JFrame {

	private final static String DATA_PATH =
			"src/c12lists/text/characterdrill/characterdrill2/data/";

	private CharacterDrillPanel drillPanel;
	private JLabel hintLabel;
	private JTextField guessField;
	private JButton giveupButton;
	private CharacterDrill drill;

	public CharacterDrillController() throws Exception {
		setTitle("Character Character Drill");
		setDefaultCloseOperation(EXIT_ON_CLOSE);

		Font font = new Font("Arial Unicode MS", 0, 24);

		drill = new CharacterDrill(DATA_PATH, "characters.txt");

		JPanel guessController = new JPanel();
		guessController.setLayout(new FlowLayout());

		JLabel guessLabel = new JLabel("English: ");
		guessLabel.setFont(font);
		guessController.add(guessLabel);

		guessField = new JTextField(15);
		guessField.setFont(font);
		guessField.setActionCommand("Guess");
		guessField.addActionListener(new GuessFieldListener());
```

```
            guessController.add(guessField);

            giveupButton = new JButton("Give Up");
            giveupButton.setFont(font);
            giveupButton.addActionListener(new GiveupButtonListener());
            guessController.add(giveupButton);

            add(guessController, BorderLayout.SOUTH);

            hintLabel = new JLabel();
            hintLabel.setFont(font);
            hintLabel.setText(drill.getHintText());
            add(hintLabel, BorderLayout.CENTER);

            drillPanel = new CharacterDrillPanel();
            drillPanel.init();
            add(drillPanel, BorderLayout.NORTH);
            drillPanel.setImage(drill.getHintImageFilename());

      }

      class GuessFieldListener implements ActionListener {
            public void actionPerformed(ActionEvent ae) {
                  try {
                        if (drill.guess(guessField.getText())) {
                              drill.reset();
                              hintLabel.setText("Right! " +
                                                drill.getHintText());
                              drillPanel.setImage(
                                          drill.getHintImageFilename());
                        } else {
                              hintLabel.setText("Guess again - " +
                                                 drill.getHintText());
                        }
                        guessField.setText("");
                  } catch (Exception e) {

                  }
            }
      }

      class GiveupButtonListener implements ActionListener {
            public void actionPerformed(ActionEvent ae) {
                  try {
                        String answer = drill.getAnswer();
                        drill.reset();
                        hintLabel.setText("Answer: " + answer +
                                    " - Next: " + drill.getHintText());
                        drillPanel.setImage(drill.getHintImageFilename());
                        guessField.setText("");
                  } catch (Exception e) {
                        hintLabel.setText(e.getMessage());
                  }
            }
      }


```

```
    public static void main(String[] args) throws Exception {
        CharacterDrillController application = new CharacterDrillController();
        application.pack();
        application.setVisible(true);
    }
}
```

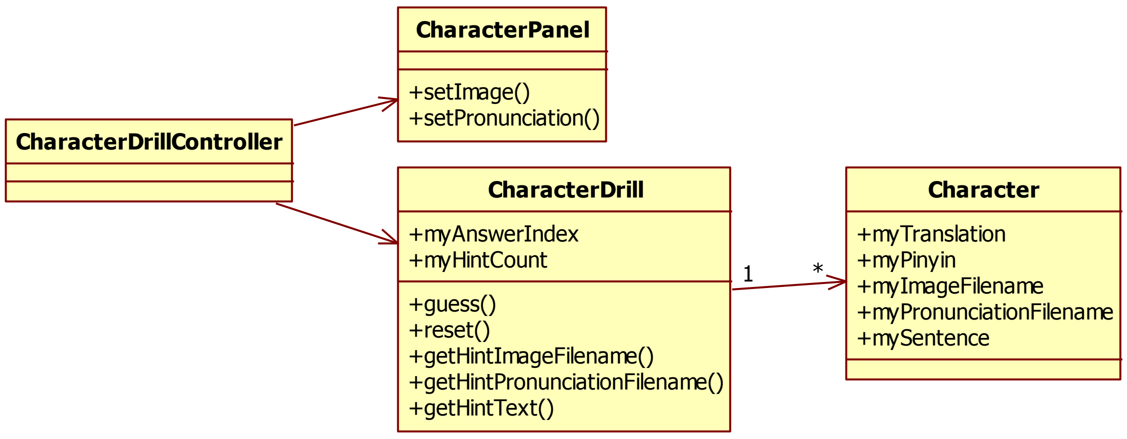
This code generally uses the GUI techniques discussed in Chapter 11.

12.4. The Example Revisited

As a final iteration of the character drill application, the following implementation makes two additions to the previous iteration:

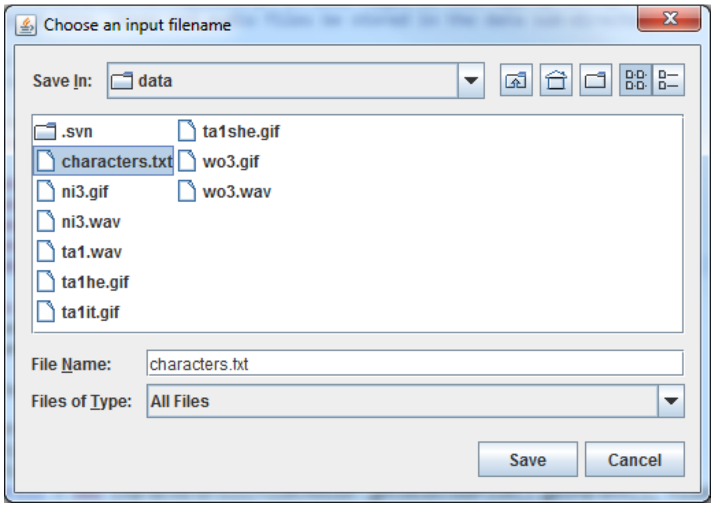
- It allows the user to specify the filename using a Swing file dialog box.
- It uses the Minim audio library to introduce a sound file for each character that, when played, demonstrates how the character should be pronounced. The appearance of the GUI remains unchanged, but now the user hears the pronunciation of the character as the image comes into view.

The class structure of the final application can be depicted as shown here:



The `CharacterDrillController` implements the GUI interface and declares and initializes the `CharacterPanel` and `CharacterDrill` classes. The `CharacterPanel` class is Processing-based and implements the graphical and audio output. The `CharacterDrill` class manages the quiz/drill itself, maintaining an `ArrayList` of `Character` objects.

The dialog box appears to the users as follows:



The dialog box is implemented in CharacterDrillController as follows:

```
JFileChooser fileChooser = new JFileChooser();
fileChooser.setDialogTitle("Choose an input filename");
fileChooser.showSaveDialog(this);
String path = fileChooser.getSelectedFile().getParent();
drill = new CharacterDrill(path, fileChooser.getSelectedFile().getName());
```

This code declares and initializes a `fileChooser` object, sets it header text and shows it to the user. It then gets both the path and the filename. The dialog box that is presented to the user is modal in that suspends the interface until the user has entered a filename. When the user has selected the file, the code segment constructs a new `CharacterDrill` object that reads its characters from the given file.

The code that implements the audio output is included in the Processing panel class:

CharacterDrillPanel.java
```
/**
 * CharacterDrillPanel supports the display of images and playing of
 * audio clips for the CharacterDrill application.
 *
 * You must include the Minim audio libraries in your build path; find
 * them in the standard code repository.
 *
 * Images taken from http://chineseculture.about.com
 *
 * @author kvlinden
 * @version Fall, 2009
 */
public class CharacterDrillPanel extends PApplet {

        private String myPath;
        private PImage myImage;
        private int myTintValue;
        private Minim myMinim;
        private AudioSnippet myPronunciation;



        public CharacterDrillPanel(String path) {
                myPath = path;
                myMinim = new Minim(this);
        }

        public CharacterDrillPanel(String path, String imageFilename,
                                        String pronunciationFilename) {
                myPath = path;
                setImage(imageFilename);
                myMinim = new Minim(this);
                setPronunciation(pronunciationFilename);
        }

        public void setImage(String filename) throws URISyntaxException {
                myImage = loadImage(myPath + filename);
                myTintValue = 0;
        }

        public void setPronunciation(String filename) {
                myPronunciation = myMinim.loadSnippet(myPath + filename);
        }

        public void playPronunciation() {
                myPronunciation.play();
        }

        public void setup() {
                size(300, 300);
                background(255);
                imageMode(CENTER);
        }

        public void draw() {
                if (myImage != null) {
                        tint(255, myTintValue);
                        image(myImage, width / 2, height / 2);
                        myTintValue = constrain(myTintValue + 1, 0, 255);
                }
        }

        public void stop() {
                myPronunciation.close();
                myMinim.stop();
        }

}
```

The highlighted segments of this code declare and initialize the Minim audio player and audio snippet objects and then play them in a separate thread on demand.