Eric Roberts
CS 106A

Handout #47
May 16, 2012

Array Lists and Files

ArrayLists and Files

Eric Roberts
CS 106A
February 12, 2010

The `ArrayList` Class

- Although arrays are conceptually important as a data structure, they are not used as much in Java as they are in most other languages, partly because the `java.util` package includes a class called `ArrayList` that provides the standard array behavior along with other useful operations.
- The main difference between a Java arrays and an `ArrayList` is that `ArrayList` is a Java class rather than a special form in the language. This design has the following implications:
  - All operations on `ArrayList`s are specified as method calls.
  - You get the number of elements by calling the `size` method.
  - You use the `get` and `set` methods to select individual elements.
- The next slide summarizes the most important methods in the `ArrayList` class. The notation `<T>` in these descriptions indicates the element type.

Methods in the `ArrayList` Class

`boolean add(<T> element)` Adds a new element to the end of the `ArrayList`; the return value is always `true`.
`void add(int index, <T> element)` Inserts a new element into the `ArrayList` before the position specified by `index`.
`<T> remove(int index)` Removes the element at the specified position and returns that value.
`boolean remove(<T> element)` Removes the first instance of `element`, if it appears; returns `true` if a match is found.
`void clear()` Removes all elements from the `ArrayList`.
`int size()` Returns the number of elements in the `ArrayList`.
`<T> get(int index)` Returns the object at the specified index.
`<T> set(int index, <T> value)` Sets the element at the specified index to the new value and returns the old value.
`int indexOf(<T> value)` Returns the index of the first occurrence of the specified value, or –1 if it does not appear.
`boolean contains(<T> value)` Returns `true` if the `ArrayList` contains the specified value.
`boolean isEmpty()` Returns `true` if the `ArrayList` contains no elements.

Generic Types

- The ability of a Java collection class to define an element type is a relatively recent extension to the language, but an extremely important one. In Java, classes such as `ArrayList` that allow the user to specify different element types are called ***generic types***.
- When you declare or create an `ArrayList`, you should always specify the element type in angle brackets. For example, to declare and initialize an `ArrayList` variable called `names` that contains elements of type `String`, you would write

```
ArrayList<String> names = new ArrayList<String>();
```

- The advantage of specifying the element type is that Java can then know what type of value the `ArrayList` contains. That information makes it possible for the compiler to check that calls to `put` and `get` use the correct types.

Restrictions on Generic Types

- In Java, generic specifications can be used only with object types and not with primitive types. Thus, while it is perfectly legal to write

```
ArrayList<String> names = new ArrayList<String>();
```

  it is not legal to write

```
ArrayList<int> numbers = new ArrayList<int>();
```

- To get around this problem, Java defines a ***wrapper class*** for each of the primitive types:

`boolean` ⟷ `Boolean`	`float` ⟷ `Float`
`byte` ⟷ `Byte`	`int` ⟷ `Integer`
`char` ⟷ `Character`	`long` ⟷ `Long`
`double` ⟷ `Double`	`short` ⟷ `Short`

Boxing and Unboxing

- Wrapper classes used to be much harder to use than they are today. Recent versions of Java include a facility called ***boxing and unboxing*** that automatically converts between a primitive type and the corresponding wrapper class.
- For example, suppose that you execute the following lines:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(42);
int answer = list.get(0);
```

  - In the second statement, Java ***boxes*** the `int` value 42 inside a wrapper object of type `Integer`.
  - In the third statement, Java ***unboxes*** the `Integer` to obtain the original `int`.
- Java's automatic conversions make it *appear* as if one is storing primitive values in an `ArrayList`, even though the element type is declared to be a wrapper class.

Reading Data from Files

- Applications that work with arrays and array lists often need to work with lists that are too large to enter by hand. In many cases, it is easier to read the values of a list from a data file.
- A ***file*** is the generic name for any named collection of data maintained on the various types of permanent storage media attached to a computer. In most cases, a file is stored on a hard disk, but it can also be stored on a removable medium, such as a CD or flash memory drive.
- Files can contain information of many different types. When you compile a Java program, for example, the compiler stores its output in a set of ***class files***, each of which contains the binary data associated with a class. The most common type of file, however, is a ***text file***, which contains character data of the sort you find in a string.

Text Files *vs.* Strings

Although text files and strings both contain character data, it is important to keep in mind the following important differences between text files and strings:

1. ***The information stored in a file is permanent.*** The value of a string variable persists only as long as the variable does. Local variables disappear when the method returns, and instance variables disappear when the object goes away, which typically does not occur until the program exits. Information stored in a file exists until the file is deleted.
2. ***Files are usually read sequentially.*** When you read data from a file, you usually start at the beginning and read the characters in order, either individually or in groups that are most commonly individual lines. Once you have read one set of characters, you then move on to the next set of characters until you reach the end of the file.

Reading Text Files

- When you want to read data from a text file as part of a Java program, you need to take the following steps:
  1. Construct a new **`BufferedReader`** object that is tied to the data in the file. This phase of the process is called ***opening the file***.
  2. Call the **`readLine`** method on the **`BufferedReader`** to read lines from the file in sequential order. When there are no more lines to be read, **`readLine`** returns **`null`**.
  3. Break the association between the reader and the file by calling the reader's **`close`** method, which is called ***closing the file***.
- Java supports other strategies for reading and writing file data. These strategies are discussed in Chapter 12.

Standard Reader Subclasses

- The **`java.io`** package defines several different subclasses of the generic **`Reader`** class that are useful in different contexts. To read text files, you need to use the following subclasses:
  - The **`FileReader`** class, which allows you to create a simple reader by supplying the name of the file.
  - The **`BufferedReader`** class, which makes all operations more efficient and enables the strategy of reading individual lines.
- The standard idiom for opening a text file calls both of these constructors in a single statement, as follows:

```
BufferedReader rd = new BufferedReader(new FileReader(filename));
```

The **`FileReader`** constructor takes the file name and creates a file reader, which is then passed on to the **`BufferedReader`** constructor.

Reading Lines from a File

- Once you have created a **`BufferedReader`** object as shown on the preceding slide, you can then read individual lines from the file by calling the **`readLine`** method.
- The following code fragment uses the **`readLine`** method to determine the length of the longest line in the reader **`rd`**:

```
int maxLength = 0;
while (true) {
   String line = rd.readLine();
   if (line == null) break;
   maxLength = Math.max(maxLength, line.length());
}
```

- Using the **`readLine`** method makes programs more portable because it eliminates the need to think about the end-of-line characters, which differ from system to system.

Exception Handling

- Unfortunately, the process of reading data from a file is not quite as simple as the previous slides suggest. When you work with the classes in the **`java.io`** package, you must ordinarily indicate what happens if an operation fails. In the case of opening a file, for example, you need to specify what the program should do if the requested file does not exist.
- Java's library classes often respond to such conditions by ***throwing an exception***, which is one of the strategies Java methods can use to report an unexpected condition. If the **`FileReader`** constructor, for example, cannot find the requested file, it throws an **`IOException`** to signal that fact.
- When Java throws an exception, it stops whatever it is doing and looks back through its execution history to see if any method has indicated an interest in "catching" that exception by including a **`try`** statement as described on the next slide.

The `try` Statement

- Java uses the `try` statement to indicate an interest in catching an exception. In its simplest form, the `try` statement syntax is

```
try {
    code in which an exception might occur
} catch (type identifier) {
    code to respond to the exception
}
```

  where *type* is the name of some exception class and *identifier* is the name of a variable used to hold the exception itself.

- The range of statements in which the exception can be caught includes not only the statements enclosed in the `try` body but also any methods those statements call. If the exception occurs inside some other method, any subsequent stack frames are removed until control returns to the `try` statement itself.

Using `try` with File Operations

- The design of the `java.io` package forces you to use `try` statements to catch any exceptions that might occur. For example, if you open a file without checking for exceptions, the Java compiler will report an error in the program.
- To take account of these conditions, you need to enclose calls to constructors and methods in the various `java.io` classes inside `try` statements that check for `IOException`s.
- The `ReverseFile` program on the next few slides illustrates the use of the `try` statement in two different contexts:
  - Inside the `openFileReader` method, the program uses a `try` statement to detect whether the file exists. If it doesn't, the `catch` clause prints a message to the user explaining the failure and then asks the user for a new file name.
  - Inside the `readLineArray` method, the code uses a `try` statement to detect whether an I/O error has occurred.

The `ReverseFile` Program

```
import acm.program.*;
import acm.util.*;
import java.io.*;
import java.util.*;

/** This program prints the lines from a file in reverse order */
public class ReverseFile extends ConsoleProgram {

   public void run() {
      println("This program reverses the lines in a file.");
      BufferedReader rd = openFileReader("Enter input file: ");
      String[] lines = readLineArray(rd);
      for (int i = lines.length - 1; i >= 0; i--) {
         println(lines[i]);
      }
   }

/*
 * Implementation note: The readLineArray method on the next slide
 * uses an ArrayList internally because doing so makes it possible
 * for the list of lines to grow dynamically.  The code converts
 * the ArrayList to an array before returning it to the client.
 */
```

The `ReverseFile` Program

```
/*
 * Reads all available lines from the specified reader and returns
 * an array containing those lines.  This method closes the reader
 * at the end of the file.
 */
  private String[] readLineArray(BufferedReader rd) {
     ArrayList<String> lineList = new ArrayList<String>();
     try {
        while (true) {
           String line = rd.readLine();
           if (line == null) break;
           lineList.add(line);
        }
        rd.close();
     } catch (IOException ex) {
        throw new ErrorException(ex);
     }
     String[] result = new String[lineList.size()];
     for (int i = 0; i < result.length; i++) {
        result[i] = lineList.get(i);
     }
     return result;
  }
```

The `ReverseFile` Program

```
/*
 * Requests the name of an input file from the user and then opens
 * that file to obtain a BufferedReader.  If the file does not
 * exist, the user is given a chance to reenter the file name.
 */
  private BufferedReader openFileReader(String prompt) {
     BufferedReader rd = null;
     while (rd == null) {
        try {
           String name = readLine(prompt);
           rd = new BufferedReader(new FileReader(name));
        } catch (IOException ex) {
           println("Can't open that file.");
        }
     }
     return rd;
  }
}
```

Exercise: Write `readScoresArray`

- In section last week, you were asked to assume the existence of a method

```
double[] readScoresArray(String filename)
```

  that reads score data from the specified file, one value per line, and returns an array of doubles containing those values.