CMPSCI 187: Programming With Data Structures

Lecture #24: Files and a Case Study
David Mix Barrington
2 November 2012

Files and a Case Study

- Volatile and Non-Volatile Storage
- Storing and Retrieving Objects as Strings
- Serialization of Objects
- Serialized Song Lists
- The `SerSongList` Class
- The `SerSongsApp` Application

Volatile and Non-Volatile Storage

- When the main method of a class stops running, its variables all disappear. The memory words it influenced may have the same contents, but there are no longer pointers to those words and they can't be accessed in any normal way. We call those variables **volatile storage**. (But the Java `volatile` keyword is different.)

- If you want to retain any of the data in your program after it has finished, you must write it to **non-volatile storage**, a **file**. Of course you have used files in your programming -- Java source files, class files, pictures, videos, perhaps HTML source files for web pages or applets. But these interactions have primarily been through the operating system -- how do we deal with files in Java?

- The `Scanner` class gives us a means to interact with files and the console in the same way. The trivia game application in Chapter 2 read the game data from a text file, attaching a `Scanner` to a `FileReader` object and using the `nextLine` method of `Scanner` -- we have also used the `next` and `nextInt` methods.

Storing and Retrieving Objects as Strings

- Every piece of data in an object ultimately consists of primitives -- numbers, booleans, or characters. All of these things can be stored in files, but operating systems tend to think of files as sequences of bytes or words.

- Our objects all have `toString` methods to get descriptions of them as `String` objects. We can store `String` objects in text files as sequences of characters, but this *may or may not* allow us to recover the objects data later by reading the file. For example, if we store a sequence of `String` objects by just writing them to a file, we need some way to **parse** the file into its individual strings on reading it. Sometimes we use line breaks for this -- each line of the file is a different string. Or if we know that there are no commas in the strings we want to store (for example, if they are all numbers), we can make a **comma-delimited** list.

- Given any class, we could write a fromString method that takes the output of the toString method and parses that string to recover the object. Fortunately, Java gives us a way to avoid having that programming task for every class.

Serialization of Objects

- If a class is serialized, then there is a way to record all the fields of any object in the class as a series of machine words that can be written to a file, in such a way that these words can be read from a file and reassembled into an object. You could imagine that serializing a class could be complicated, if the objects consist of complex structures made from references, like lists or trees.

- From the programmer's standpoint, though, serializing a class couldn't be easier. All you need to do is add the words `implements Serializable` to the class header! To read and write objects, you have to attach an `ObjectOutputStream` to your `FileOutputStream`, and an `ObjectInputStream` to your `FileInputStream`. These classes have a `writeObject` and a `readObject` method respectively.

- Note that you must `import java.io.*` to use these classes, and enclose any calls to these methods in a `try-catch` block because they may throw checked exceptions.

Serialized Song Lists

- One of DJW's applications of lists was to maintain a playlist of songs, where a Song object stored the name of the song and its duration in seconds. Since it's reasonable for a user to expect a playlist to persist between uses of the program, in Section 6.8 they present a case study with playlists that may be stored in files.

- They alter their `Song` class into a `SerSong` class that implements `Serializable`. This really does mean changing only the header line. They also define a class called `SArrayIndexedList` which is identical to `ArrayIndexedList` except that it also implements `Serializable`.

- They then define a SerSongList class whose objects are serializable playlists, and a SerSongsApp class to allow the user to create and manipulate playlists. These extend the former example classes in that the user sees the lists as indexed beginning with 1, not 0, and that playlists now also have names.

The `SerSongList` Class

- All we need to do in order to have a serializable class is to declare that we implement the interface, and make sure that the underlying data structure, and the things we store in it, are also serializable.

```
public class SerSongList implements Serializable {
   protected String listName;
   protected int totalDuration = 0;
   protected SArrayIndexedList songList;
   // constructor, getters, setters

   public void add (int number, serSong song) {
      totDuration += song.getDuration( );
      if ((number <= 0) || (number > (songList.size( ) + 1)))
         songList.add(songList.size( ), song);
      else songList.add(number - 1, song);}

   // toString method to give list starting with 1, total
   // duration in minutes and seconds
```

The `SerSongsApp` Application

```
public class serSongsApp {
   public static void main (String[ ] args) {
      Scanner conIn = new Scanner(System.in);
      // local variables including FILENAME = “songs.dat”
      try{
         ObjectInputStream in = new ObjectInputStream(
                                new FileInputStream(FILENAME));
         songs = (SerSongList) in.readObject( );
         System.out.println(songs);{
      catch (Exception e) {
         // print message, ask for name of new songlist
      System.out.println (“\nSong name (press Enter to end): “);
      name = conIn.nextLine( );
      while (!name.equals(“”))
         // get song data, create SerSong, add where user wants
      // tell user list is being saved in FILENAME
      try{ObjectOutputStream out = new ObjectOutputStream (
                                   new FileOutputStream(FILENAME));
          out.writeObject(songs); out.close( );}
      catch (Exception e) {
         System.out.println(“Unable to save song information.”);}}}
```