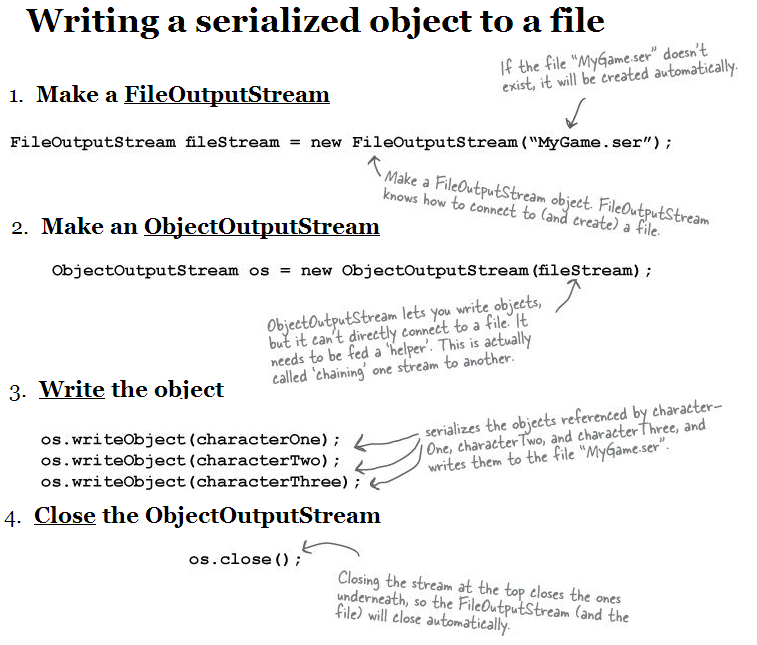
**B”H**

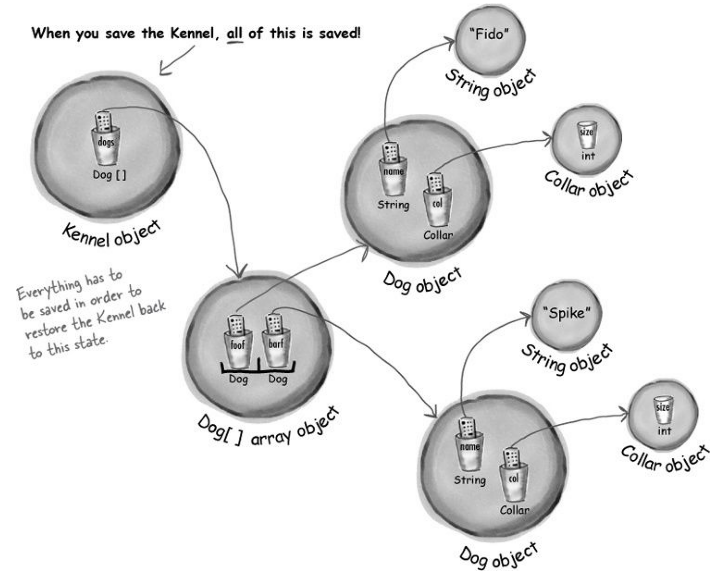
**Chapter 14**

**Serialization and File I/ O: Saving Objects**

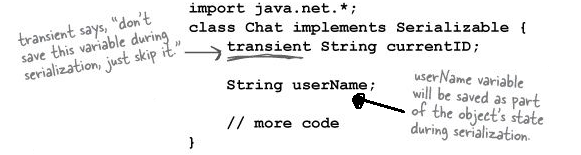
* **Note this book doesn’t discuss storing in data in a database, which is a lot more powerful and flexible though a bit more work to achieve.**
* So excluding the database option, if your data will be used by only the Java program that generated it: use **serialization.**
* If your data will be used by other programs: Write a plain text file. Write a file, with delimiters that other programs can parse. For example, a tab-delimited file that a spreadsheet or database application can use.



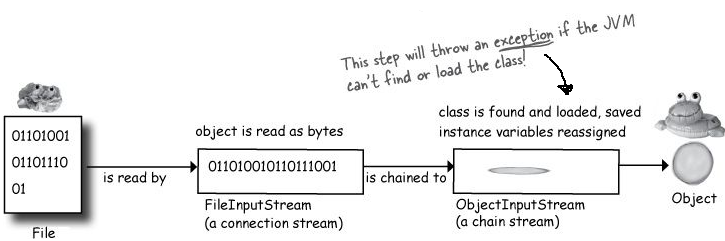
* **Connection streams** represent a connection to a source or destination (file, socket, etc.) while **chain streams** can’t connect on their own and must be chained to a connection stream.
* Often, it takes at least two streams hooked together to do something useful — one to represent the connection and another to call methods on.
* Connection streams are usually too low-level. FileOutputStream, for example, has methods for writing bytes. But we don’t want to write bytes! We want to write objects, so we need a higher-level chain stream.
* When we call writeObject() on the ObjectOutputStream, the object gets pumped into the stream and then moves to the FileOutputStream where it ultimately gets written as bytes to a file.
* See the Decorator pattern (chapter 3 of Head First Design Patterns)
* When an object is serialized, all the objects it refers to from instance variables are also serialized. And all the objects those objects refer to are serialized. And all the objects those objects refer to are serialized - it happens automatically!
* Serialization saves the entire object graph. All objects referenced by instance variables, starting with the object being serialized.



* If you want your class to be serializable, implement Serializable
* The Serializable interface is known as a marker or tag interface, because the interface doesn’t have any methods to implement. Its sole purpose is to announce that the class implementing it is serializable.
* If any superclass of a class is serializable, the subclass is automatically serializable even if the subclass doesn’t explicitly implement Serializable.
* See GameSaverTest.java which has simple demo of serialization and de-serialization
* Either the entire object graph is serialized correctly or serialization fails. You can’t serialize a Pond object if its Duck instance variable refuses to be serialized (by not implementing Serializable).
* Mark an instance variable as **transient** if it can’t (or shouldn’t) be saved.

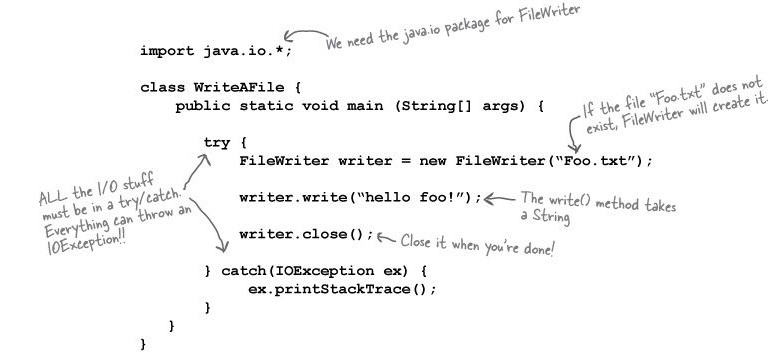


* It could be that the class designer simply forgot to make the class implement Serializable. Or it might be because the object relies on runtime-specific information that simply can’t be saved - like network connections, threads, or file objects.
* If the class itself is extendable (i.e. not final), you can make a serializable subclass, and just substitute the subclass everywhere your code is expecting the superclass type.
* When an object is deserialized and its superclass is not serializable, the superclass constructor will run just as though a new object of that type were being created. If there’s no decent reason for a class to not be serializable, making a serializable subclass might be a good solution.
* If you serialize an object, a transient reference instance variable will be brought back as null.
* Serialization is smart enough to know when two objects in the graph are the same. In that case, only one of the objects is saved. For example, if you have two different Cat objects in the Kennel, but both Cats have a reference to the same Owner object. The Owner won’t get saved twice.
* Deserialization is a lot like serialization in reverse:
  1. Make a **FileInputStream**
  2. Make an **ObjectInputStream**
  3. Read the objects
  4. Cast the objects
  5. Close the **ObjectInputStream**
* See GameSaverTest.java which has simple demo of serialization and de-serialization

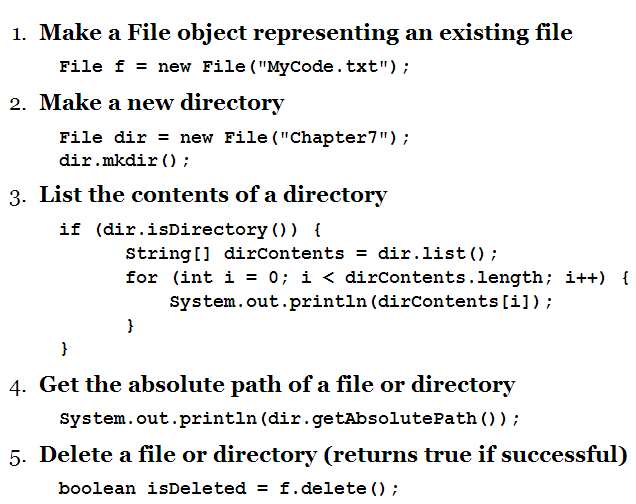


* If the object has a non-serializable class somewhere up its inheritance tree, the constructor for that non-serializable class will run along with any constructors above (above only) that (even if they’re serializable). Once the constructor chaining begins, you can’t stop it, which means all superclasses, beginning with the first non-serializable one, will reinitialize their state.
* Remember, static means “one per class” not “one per object”. Static variables are not saved, and when an object is deserialized, it will have whatever static variable its class currently has..

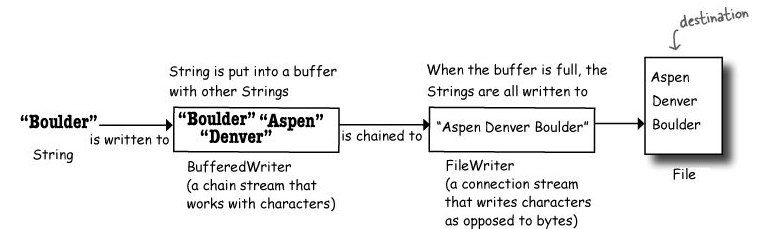
**Saving to a text file**



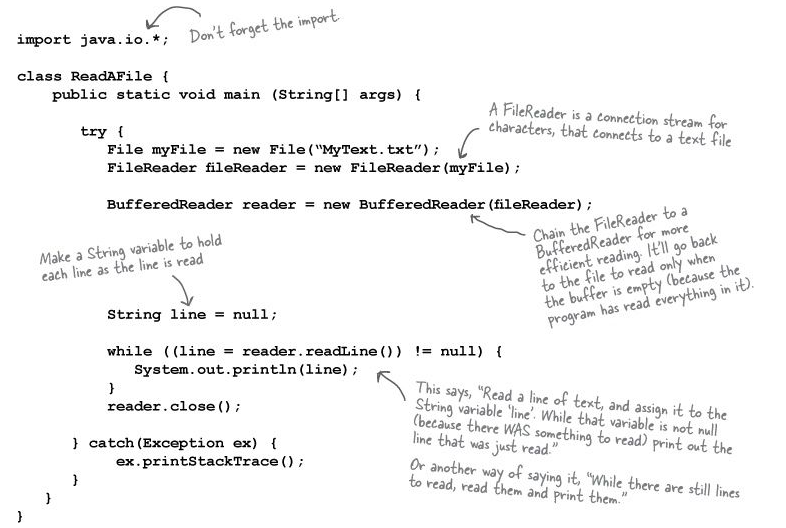
* Run and see: java QuizCardBuilder where you can create and save a quiz card set in a .txt file
* Run and see: java QuizCardReader where you can load the a quiz card set .txt file and play the quiz
* The java.io.File class represents a file on disk, but doesn’t actually represent the contents of the file.
* Think of a File object as something more like a pathname of a file (or even a directory) rather than the actual file itself.
* One very useful thing about a File object is that it offers a much safer way to represent a file than just using a String file name. For example, most classes that take a String file name in their constructor (like FileWriter or FileInputStream) can take a File object instead. You can construct a File object, verify that you’ve got a valid path, etc. and then give that File object to the FileWriter or FileInputStream.
* Some things you can do with a File object:



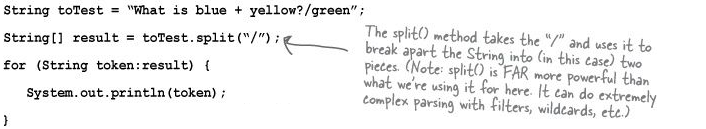




* Buffers give you a temporary holding place to group things until the holder (like a cart) is full. They’re much more efficient than working without them. You can write to a file using FileWriter alone, by calling write( someString), but FileWriter writes each and every thing you pass to the file each and every time. That’s overhead you don’t want or need, since every trip to the disk is a Big Deal compared to manipulating data in memory. By chaining a BufferedWriter onto a FileWriter, the BufferedWriter will hold all the stuff you write to it until it’s full. Only when the buffer is full will the FileWriter actually be told to write to the file on disk. If you do want to send data before the buffer is full, you do have control - just flush it: writer.flush()



* String split() lets you break a String into pieces.



**Using the serialVersionUID**

* **Changes to a class that can hurt deserialization:** 
  + Deleting an instance variable
  + Changing the declared type of an instance variable
  + Changing a non-transient instance variable to transient
  + Moving a class up or down the inheritance hierarchy
  + Changing a class (anywhere in the object graph) from Serializable to not Serializable
  + Changing an instance variable to static
* **Changes to a class that are usually OK:**
  + Adding new instance variables to the class
  + Adding classes to the inheritance tree
  + Removing classes from the inheritance tree
  + Changing the access level of an instance variable has no affect on the ability of deserialization to assign a value to the variable
  + Changing an instance variable from transient to non-transient (previously-serialized objects will simply have a default value for the previously-transient variables)
* Each time an object is serialized, the object (including every object in its graph) is ‘stamped’ with a serialVersionUID for the object’s class.
* It’s computed based on information about the class structure.
* If you think there is ANY possibility that your class might evolve, put a serial version ID in your class. When Java tries to deserialize an object, it compares the serialized object’s serialVersionUID with that of the class the JVM is using for deserializing the object.
* If the two numbers don’t match, the JVM assumes the class is not compatible with the previously-serialized object, and you’ll get an exception during deserialization.
* So, the solution is to put a serialVersionUID in your class, and then as the class evolves, the serialVersionUID will remain the same and the JVM will say, “OK, cool, the class is compatible with this serialized object.” even though the class has actually changed. In other words, you are taking responsibility for any issues that come up when an older object is brought back to life with a newer class.
* To get a serialVersionUID for a class, use the serialver tool that ships with your Java development kit.

