Head First Java: Chapter 14 Notes

September 12, 2019

Serialization

Serialization is a process that "*flattens*" a Java object such that its states is saved into a file that can be revived later on. This is useful for passing objects between different Java programs. There are four steps to the serialization of a Java object into a file:

1. Instantiate a `FileOutputStream` object — The `FileOutputStream` class object in Java is the object with the method to create files from a serialized object. There is a variety of ways to instantiate a `FileOutputStream`, but commonly the constructor takes a `String` argument of the file name where the object is to be saved as.

2. Instantiate a `ObjectOutputStream` object — The `ObjectOutputStream` class object serialize the objects to be saved into a file. When instantiating an `ObjectOutputStream` the constructor takes arguments that depends on the final destination of the serialized object. In this case because the object is to be saved into a file, the `ObjecttOutputStream` will need to be *chained* to a `FileOutputStream`, which would be the argument for the constructor.

3. Writing the object to the `ObjectOutputStream` — The object to be serialized is written to the `ObjectOutputStream` by calling its `writeObject()` method. Multiple objects can be written to an `ObjectOutputStream`, and the order matters, and when the objects are read back out of the file it will be read in the order they were written in.

4. Close the `ObjectOutputStream` — By calling the `close()` method of the `ObjectOutputStream`, the entire stream chain is closed (in this case, the `FileOutputStream` as well).

Serializing and saving a specific object to a file means that the object's *state* is saved, and can be revived later and/or elsewhere. The state of an object are its instant variables, which can be primitives or object references. In the case of primitive instance variables, the values of the primitives (*e.g.* `int 24`, `double 874.0045`) are written into the serialization. For object reference instance variables, however, the objects the variables are also serialize, and if those objects has object reference instance variables, objected being referenced by those variables are also serialize, and so forth, until the entire object graph is written into the serialization.

In order to utilize serialization, the `Java.io.*;` package needs to be imported, and each object class to be serialize must implement the `Serializable` interface. Note that if an instance variable of the object to be serialize references a object where its class did not implement `Serializable`, the serialization will fail. In other words, all object classes in the object graph of the object to be serialized must all implement `Serializable` for the serialization to be successful. It is possible to exclude instance variables of an object class from being serialized by declaring them to be `transient`. Instance variables declared `transient` will be ignored during serialization. This can be used to prevent serialization exceptions as described above. During deserialization, ignored `transient` instant variables (which now have no values because they were not serialized at the first place) will default to a value of `null`.

Deserialization in Java takes five steps, which is essentially the reverse of serialization:

1. Instantiate a `FileInputStream` object — Just like its writing counterpart, a `FileInputStream` object is able to read bytes from a file into serialization that can be deserialized. The constructor can take various arguments, such as a file name.

2. Instantiate a `ObjectInputStream` object — Again, like its writing counterpart, the `ObjectInputStream` object deserialize the serialization back to an object by calling its methods. It cannot read files directly so it must be chained to a `FileInputStream` or other objects that is able to read whatever form the serialized object currently takes.

3. Reading the object from `ObjectInputStream` — By calling the `readObject()` method of a `ObjectInputStream` object, the serialization will be deserialized back into an object. Note that despite what class the original object belongs to, the deserialized object now will be of class `Object`.

4. Casting the deserialized object — Because objects deserializes into `Object` class, they must be cast back to their original class.

5. Close the `ObjectInputStream` — By calling the `close()` method of the `ObjectInputStream`, the entire stream chain is closed (in this case, the `FileInputStream` as well).

Deserialization is handled by the JVM, and although the object is deserialized into class `Object`, the JVM will need to have access to the original (not a polymorph of) class of the object. This applies to all object class within the serialized object's object graph. Note that the process of deserialization does not run the object's constructor, as that would reinitialize all instance variable values. Another note is that static instance variables are not serialized in the way other instance are, as the value of a static variable is associated with the *class* itself, and not an instance of the class. Here is an example of a generic serialization-deserialization code block:

```
import java.io.*;

// Class and method declaration here.

    FileOutputStream fos = new FileOutputStream(‘‘object.ser’’);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(one);
    oos.writeObject(two);
    oos.writeObject(three);
    oos.close()

    // Other code here.

    FileInputStream fis = new FileInputStream(‘‘object.ser’’);
    ObjectInputStream ois = new ObjectInputStream(fis);
    Object newOne = ois.readObject();
    Object newTwo = ois.readObject();
    Object newThree = ois.readObject();
    ActualClass realNewOne = (ActualClass) newOne;
    ActualClass realNewTwo = (ActualClass) newTwo;
    ActualClass realNewThree = (ActualClass) newThree;
    ois.close();
```

The above code block is verbose for clarity, but several of the code lines are often combined into as single line in practice (for example, the reading of the object from the ObjectInputStream and casting can be combined.) Also, most methods/constructors used in serialization/deserialization throw exceptions (usually IOException and its subclasses), so related code needs to be wrapped in a try-catch block.

Because serialized objects are often used in a later date, version control is essential. The class of the serialized objects could have changed in between serialization and deserialization (these changes can include, for example: deleting instance variables; changing the type of instance variables; changing whether or not instance variables are transient/static; moving the class up or down the inheritance tree, etc.) To prevent class version conflicts, serialVersionUID, which is a unique identifier of a class based on, among other things, the identity of the class, and version of JAVA/JVM, and so on, is used as a fingerprint of sorts. During deserialization, the JVM checks the serialVersionUID of the class the JVM has access to, and the class in the serialization. If the serialVersionUIDs are different, the deserialization will fail. This can be circumvented by finding out the `serialVersionUID` of the class the JVM currently has access to by using the JDK command line tool `serialver` (which takes the name of the class of interest as an argument), which will return a `serialVersionUID` variable declaration (such as: `private static final long serialVersionUID = -6042309455930068204L`). This can be used to manually set the `serialVersionUID` of a class even after a user has changed it, so that the JVM would recognize that the modified class has the same `serialVersionUID` as the class it has access to, and go ahead with the deserialization despite the two classes really did changed. Note that if there still actually are conflicts between the original class and modified class, deserialization will still fail; manually setting the `serialVersionUID` merely gives the user an opportunity to account for differences between the original class and modified class and had came up with code to compensate for that (*e.g.* adding default values for new instance variables added to the class after it was serialized.)

Writing and Reading Text Files

If objects need to be saved and later be used by *non-Java* programs, then serialization will not suffice. In this case, one way to accomplished this is to write the data into a text file. Writing text files require not a `FileOutputStream` but a `FileWriter` object, also available in `java.io.*`. The process is similar except no object stream step is needed. Reading from a text file require similar processes, in this case a `FileReader` object is used instead, by calling methods such as `readLine()`, which reads an entire line of text from the file (in order), which can then be assigned to a `String` variable or other objects.

The `File` Class

The Java class `File` is an object representation of a file (or more accurately, a location on disk), similar to a file handle in Python. The object to not have any internal values representing the content of the file, but only provide access to it. The `File` class comes with many methods similar to functions of the Python `os` package, such as making a new directory (`mkdir()`), listing contents of a directory (`list()`), Boolean check if `File` object is a directory (`isDirectory()`), return absolute path of the `File` object location on disk (`getAbsolutePath()`), and deleting the file/location referred to by the `File` object (`delete()`). `File` objects can be passed to other stream-related objects such as `FileInputStream` and `FileOutputStream` rather than a `String` file name. This is a good practice that provides more security and better exception handling.

Buffers

A buffer in Java is a temporary holding place to group items together until the buffer is full. There are many types of buffer in Java; one example would be `BufferedWriter`, which temporarily stores `Strings` that was read/will be written later. For writing text files, using a `BufferedWriter` significantly increases efficiency both in code and also in memory usage. To send the buffered content of a buffer to the next chained stream before the buffer is full, as it is often the case, the `flush()` method can be called. A `BufferedReader` is the analogous object that is more efficient at reading text files (by using the `readLine()` method, for example.)