Persistence

using the JDK 1.1

Part 1

Pedro Alves

- Preface
- What is Persistence?
  - Object lifetime
  - The need for Persistence
  - Persistence
  - Forms of Persistence
- File-based persistence - first approach
  - Streams
  - Objects
  - Storing/Retrieving objects
  - Problems
  - Conclusion

Preface

This month I will start a set of notes on Persistence. If you don't even know what Persistence is, don't worry - that's the first thing I'll cover. This is not going to be an advanced thesis on Persistence that only the author will be able to understand! I'll try to explain all the concepts in a straight simple way, illustrating them with simple examples whenever necessary. So, you don't need to be an experienced Java programmer - whatever that could be - to understand my text. Nevertheless, being familiarized with OO concepts and the Java API (mainly streams) will certainly help. And even you, the so-called Java guru, should read my notes, for your corrections to my possible inaccuracies shall be helpful and, in the end, you also may learn a little more.

If you have any doubts, suggestions, critics or whatever, feel free to contact me by e-mail. I will be grateful and publish any interesting ideas in the following month.

Enjoy!

Pedro Alves (phqa@camoes.rnl.ist.utl.pt)

What is Persistence?

Persistence is the ability of objects to exist beyond their normal lifetime.

Object lifetime

Some programmers may not be familiarized with the meaning of an object lifetime, but they know what happens to a variable when a program quits or when the scope in which the variable was declared finishes.

Consider this code sample:

```
{
  String str;
  str = new String("Persistence"); // a new object was born
  System.out.println(str);
}

// an object is going to die at the hands of the garbage collector
```

When you use the *new* operator, you are giving life to an object. That's what happens with *str*. When a scope (e.g., a function) ends, the JVM's garbage collector destroys any objects declared in that scope. After the closing '}' we can consider *str* destroyed. Its lifetime was only a few ms.

We can extend an object's lifetime beyond the scope of a particular function by declaring that object in the main class - the object becomes global. But what if the object is already global? How can we extend its lifetime? And, by the way, why should we want to do that?

The need for Persistence

Consider a program to handle accounts. We could have a class like this:

```
class Account {
  int accountID;
  float balance;
  String owner;
  ...
}
```

We don't want to loose our Account objects when the program exits. We want those objects' lifetime extended beyond the program scope. We probably want the objects to persist for years! We need Persistence.

Persistence

Now you should be able to understand the first definition of Persistence:

"Persistence is the ability of objects to exist beyond their normal lifetime."

Objects that accomplish this are said to be 'persistent objects'.

So, how do we implement Persistence? We'll have to save the internal state of an object (into a non-volatile memory) in such a way that it can later be retrieved. Here's a second definition for Persistence:

"Persistence is the ability of objects to store and retrieve their internal state."

An object is characterized by its fields and methods. But we don't need to save the methods because they won't change between runs of the program. We do need to save the fields (and more than just the fields, as we'll see later). In order to save these we have to convert them into a sequence of bytes - that's where we'll need the powerful Object Serialization API.

Forms of Persistence

As I said earlier, objects' internal state must be saved into a non-volatile memory. How this is accomplished may differ from application to application. Basically, we have 3 forms of Persistence:

File-based persistence

It's the simplest and cheapest way but also the least powerful. It's the ideal method for small applications, but forget it when dealing with large DBMSs (Database Management Systems). Nevertheless, this approach makes it easier to explain the concepts behind Persistence, and it will be my primary choice for these notes.

RDBMS

Relational databases use tables to store information. This has nothing to do with how information is stored in objects. And that's the big problem with this kind of Persistence - how to map objects into tables with simple SQL statements like 'insert' or 'update'. These DBMSs have the advantages of being widely known and of supporting concurrency control and transactional properties. The JDBC API helps too.

OODBMS

Object Oriented Database Management Systems are the final and most important solution to Persistence. Basically, their ability to store objects relieves the programmer of coding SQL statements or methods to

serialize objects. They also support concurrency control and transactional properties. Later I will present a few Java-related OODBMS projects.

File-based persistence - first approach

File-based persistence is the easiest approach to Persistence. Although it suffers from some problems that we'll examine later, it's the best solution to small applications with simple object structures. It's also the best way to learn the concepts behind Persistence and to notice some problems that may arise.

Streams

When we say 'files' in Java we say 'streams', more specifically 'file streams'. I assume that you are already familiarized with this subject. If you're not, take a rest on this text and try to learn how I/O operations are dealt in Java.

FileInputStream and *FileOutputStream* (derived from *InputStream* and *OutputStream*) receive a *File* object or a *String* object (the file name) in their constructor and have methods to read/write a byte or set of bytes:

```
FileInputStream(File file);
FileInputStream(String name);
read();
read(byte[]);

FileOutputStream(File file);
FileOutputStream(String name);
FileOutputStream(String name, boolean append);
write(byte[]);
```

Now we have the first problem - how to convert an object into a sequence of bytes?

We don't need to. We can use one of the most useful filter streams, the *DataInputStream/DataOutputStream*, over our file stream. Here's how it can be done:

```
// open a file stream to 'file.txt'
FileOutputStream fileOut = new FileOutputStream("file.txt");
// open a data stream 'on top of' the file stream
DataOutputStream dataOut = new DataOutputStream(fileOut);
// write to the data stream, which will be streamed 'into' the file stream
dataOut.writeInt(i);
dataOut.close();
```

Notice the line in which the *writeInt* method of *DataOutputStream* is called. If we had only the *write(byte[])* method we would have to do some boring work to convert an integer to a sequence of bytes. The data stream classes do it for us. Other useful methods are:

```
writeBoolean()
writeByte()
writeChar()
writeShort()
writeLong()
writeFloat()
writeDouble()
writeUTF()    // for writing Strings
```

Of course, the corresponding read methods are supplied by *DataInputStream.*

We can now read/write primitive types, but the only 'object' that we can read/write is the *String*. In the next section we will see how we can read and write other objects.

Objects

Suppose we had a simple class like this:

```
class Employee {
  int ID;
  String name;
}
```

We can easily write an object of this type to a file - we'll write its fields sequentially using the methods supplied by the *DataOutputStream* class. As the fields are encapsulated inside the class and because we want to follow the OOP rules we'd better define two methods in the class *Employee*, one for reading the fields from a stream and another for writing the fields to a stream. I said 'stream' and not 'file' because this way we have a more flexible class that can send its data, for example, through a network connection.

```
class Employee {
   int ID;
   String name;

   public void writeObject(DataOutputStream dataOut) {
      dataOut.writeInt(ID);
      dataOut.writeUTF(name);
   }

   public void readObject(DataInputStream dataIn) {
      ID = dataIn.readInt();
      name = dataIn.readUTF();
   }
}
```

If all our classes implement these methods then our distance towards Persistence will get shorter. Unfortunately, the standard Java classes do not implement these methods, but that's no reason to desperate - later I'll cover the Object Serialization API, it will help us solving the problem. For the remaining of this section, I'll only focus on Persistence in classes created by us. You may think right now that it won't be very useful, but, believe me, most of the times you'll want your own classes to be

persistent - it's also the best way to come upon some problems with Persistence.

Storing/Retrieving objects

From now on, classes that implement the *readObject/writeObject* methods will be called 'persistent classes'. This will be clear when we talk about the persistent interface.

So, we have a few persistent classes and some corresponding persistent objects and we now want to store them in a file. We don't have to worry about the objects and their internals because the *writeObject* method takes care of the details for us. But we still have to supply a data stream in which the object will write to/read from - you can easily do it in the *main* method.

As you probably know, when we are dealing with streams, we have to catch the IO exceptions and take the appropriate actions. The *readObject/writeObject* methods can throw these exceptions. We could handle those exceptions inside the methods, but we'd better handle everything 'outside', so we simply add the *throws* command:

```
public void writeObject(DataOutputStream dataOut) throws IOException
public void readObject(DataInputStream dataIn) throws IOException
```

We can handle the exception thrown by simply enclosing the call to the method with a *try/catch* statement:

```
try {
   obj.writeObject(dataOut);
} catch (IOException e) { System.out.println(e.toString()); }
```

Having said this, recall the last example:

```
class Employee {
   int ID;
   String name;

   // used for reading an object
   public Employee() {}

   public Employee(int ID, String name) {
      this.ID = ID;
      this.name = name;
   }

   public void writeObject(DataOutputStream dataOut) throws IOException { ... }
   public void readObject(DataInputStream dataIn) throws IOException { ... }

   public static void main(String args[]) {

      try {
         Employee emp1 = new Employee(123, "Pedro Alves");
         // create a data stream on top of a file stream
         DataOutputStream dataOut = new DataOutputStream(
           new FileOutputStream("Employee.prt"));
```

```
        // 'tell' the object to write itself to the stream
        emp1.writeObject(dataOut);

        dataOut.close();

        // create a data stream on top of a file stream
        DataInputStream dataIn = new DataInputStream(
          new FileInputStream("Employee.prt"));

        // create an 'empty' object
        Employee emp2 = new Employee();

        // 'fill' the object
        emp2.readObject(dataIn);

        dataIn.close();

        // compares the two objects
        if (emp1.equals(emp2))
          System.out.println("It worked!");

      } catch (IOException e) { System.out.println(e.toString()); }
   }
}
```

This program creates a 'filled' object, *emp1*, and saves it in a file. Then it creates an 'empty' object, *emp2*, and 'fills' it by reading the values from the file in which *emp1* was saved. Finally, it compares the two objects to check if they're equal.

Problems

Although this program works fine, it has a lot of problems. Here are the most critical:

- We have to create an empty object before we can read it from the file. That's because we only have access to the readObject method after creating the corresponding object. If we don't know the class of the object saved in the file, we can't use this method. We also have to implement a constructor with no parameters that is only needed for this purpose. We could use the other constructor with dummy values but that wouldn't be a nice programming style.

- If we have a more complex class with references to other objects, possibly non-persistent objects (the standard Java classes objects, for example), we would be in trouble implementing the *writeObject*/*readObject* methods.
  Suppose the following class:

  ```
  class Headache {
     BitSet set;
     Image img;
     Headache ache;
  }
  ```

  Any ideas for implementing *writeObject* and *readObject*?

- Persistence is not transparent to the programmer. The programmer has to mix 'normal' code with Persistence-details code, which leads to confusing, hard to reuse and maintain classes. I'll focus on this particular problem next month in the section '*Implementing a persistent store*'.

- Inheritance can be a problem too. If a class inherits from a persistent class, it becomes persistent too, because the *readObject/writeObject* methods are inherited from the superclass. But if we add some fields to the subclass that don't exist in the superclass, we have to override the *readObject/writeObject* methods. As usual, this is better explained with an example:

```
class A {
   int a;
   long b;

   public A(int a, long b) {
      this.a = a;
      this.b = b;
   }

   public writeObject(DataOutputStream dataOut) { ... }
   public readObject(DataInputStream dataIn) { ... }
}

class B extends A {
   int c;

   public B(int a, int c, long b) {
      super(a,b);
      this.c = c;
   }

   // we have to override writeObject because of 'c'
   public writeObject(DataOutputStream dataOut) {
      super.writeObject(dataOut);
      dataOut.writeInt(c);
   }

   // we have to override readObject because of 'c'
   public readObject(DataInputStream dataIn) {
      super.readObject(dataIn);
      c = dataIn.readInt();
   }
}
```

Conclusion

These are just a few problems that may arise, I won't discuss other major problems right now - it's too early. I'm sure that some of you have thought of other problems. If you want to share those with us, please contact me by e-mail, I'll be glad to publish them next month. Anyway, the '*Employee*' program works and it's a good starting point to higher achievements. I hope you've enjoyed it, come next month for more.

Pedro Alves (phqa@camoes.rnl.ist.utl.pt)