420-406-AB Data Structures
Winter 2015

March 30th, 2015
Worth 3%

Lab VI: Serialization

(Due on April 13th, 2015)

Overview

Serialization is the process of translating a data structure (or object) in memory into a series of bytes that can be stored to file or communicated over a network. The inverse process, converting the data structure from bytes back into memory, is called deserialization. In this lab you will write serialization and deserialization methods for the some of the data structures in this course.

For an example of serialization, the binary representation of the string "and now for something completely different", in hexadecimal, is[1]

```
0000000      0000    2a00    6e61    2064    6f6e    2077    6f66    2072
0000020      6f73    656d    6874    6e69    2067    6f63    706d    656c
0000040      6574    796c    6420    6669    6566    6572    746e
0000056
```

The first column above is the byte index. The first 4 bytes of the serialization, hexadecimal $0x0000002A$, or decimal 42, records the number of characters in the `String`. Interpreting the bytes as ASCII symbols we get:

```
0000000  nul nul nul   *   a   n   d  sp   n   o   w  sp   f   o   r  sp
0000020    s   o   m   e   t   h   i   n   g  sp   c   o   m   p   l   e
0000040    t   e   l   y  sp   d   i   f   f   e   r   e   n   t
0000056
```

1 Binary Files

Serialization requires reading and writing binary data. For this lab, we will use Java's NIO (New IO) library to read from and write to binary data. Read through the following sections of the "Java NIO Tutorial" by Jakob Jenkov (`http://tutorials.jenkov.com/java-nio/index.html`):

- Java NIO Overview.
- Java NIO Channel.
- Java NIO Buffer.

You can skip over the topics concerning "Selectors". You can also see how to use the Java NIO in the `String` class serializer provided.

[1] displayed using the Linux command 'od'

2 Serializers

In this lab, each data type to be serialized will have an associated class, called it's *serializer* that performs the necessary operations to serialize and deserialize objects of that type. Each of these serializer classes have operations:

Operation	`void set(T x)`
Purpose	Set the object to be serialized
Pre-conditions	None.
Post-conditions	The object to be serialized is `x`

Operation	`T get()`
Purpose	Get the object that was deserialized
Pre-conditions	None.
Post-conditions	The object that was deserialized is returned

Operation	`int serialize(SeekableByteChannel channel)`
Purpose	Create a binary serial representation of the object and write it to `channel`.
Pre-conditions	The `channel` is open and ready for binary writing. The "position" is at the end of the channel (for appending).
Post-conditions	The binary serial representation of the object is written to `channel`. The position is at the end of the channel (for appending). The total number of bytes written is returned.

Operation	`int deserialize(SeekableByteChannel channel)`
Purpose	Read the binary serial representation of the object from `channel` and recreate the object in memory.
Pre-conditions	The `channel` is open and ready for binary reading. The position is at the beginning of the serial representation.
Post-conditions	The binary serial representation is read from `channel` and the object is recreated. The position in `channel` is after the serial representation. The total number of bytes read from the file is returned.

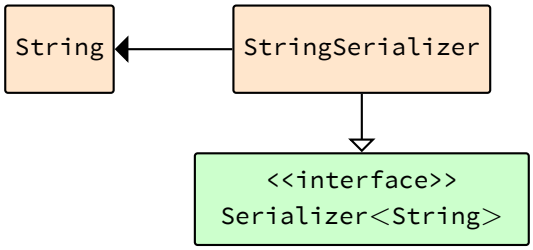
2.1 Serializer Interface

Since these operations are common to all serializers, we'll provide a Java interface for each of them to implement.

```
public interface Serializer<T> {
    // get/set the object to be (de)serialized
    public T get();
    public void set(T x);

    // methods for (de)serialization of the current object
```

```
    public int serialize(SeekableByteChannel channel)
                    throws IOException, SerializationException;
    public int deserialize(SeekableByteChannel channel)
                    throws IOException, DeserializationException;

}
```

For example, the `String` would be serialized by a class `StringSerializer` that implements `Serializer<String>`. This is shown in the following UML diagram:



So for example, to serialize a `String`, one would perform:

```
String s = "and now for something completely different";
StringSerializer serializer = new StringSerializer();
serializer.set(s);
serializer.serialize(channel);
```

For generic serializers, it is expected that you will need to provide a serializer object for type `T`:

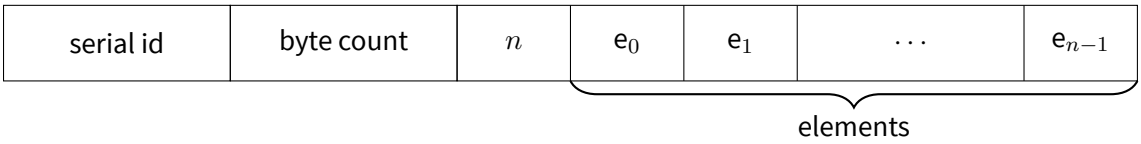
```
LinkedList<String> list = new LinkedList<String>();
list.add("foo");
list.add("bar");

StringSerializer stringSerializer = new StringSerializer();
LinkedListSerializer<String> listSerializer =
    new LinkedListSerializer<String>(stringSerializer);
listSerializer.serialize(channel);
```

3 Serialization Formats

3.1 `LinkedList<T>` class

The serial representation for the `LinkedList<T>` class will consist of the following information, written in order:

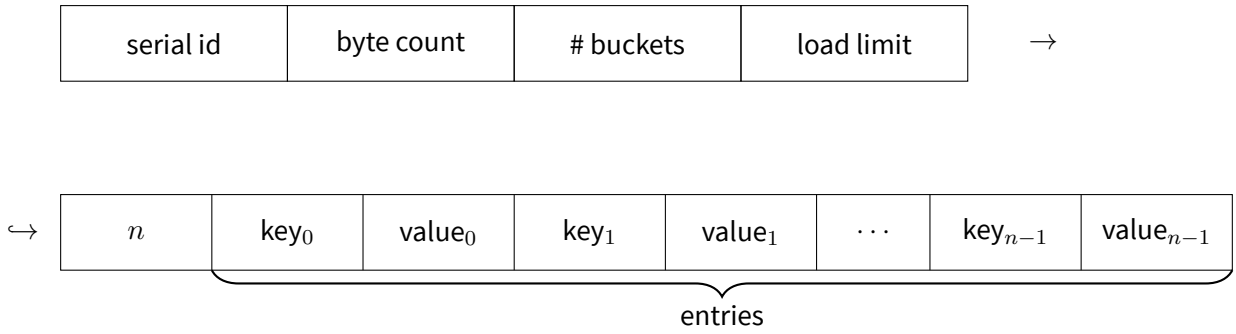
1. A header composed of a Serial ID and content byte count (see Section 4).
2. The size of the list: n.
3. The serialized elements in the list.



3.2 `HashMap<K,V>` class

The serial representation for the `HashMap<K,V>` class will consist of the following information, written in order:

1. A header composed of a Serial ID and content byte count (see Section 4).
2. The current number of buckets.
3. The load factor limit (see the implementation).
4. The size of the map.
5. The serialized entries in the map.



4 Checks

You will use two checks to ensure that the data you are deserializing from the binary file is really the data that was serialized by your code. Two values, Serial ID and byte count, will be written as the first items in the serialization. When deserializing, you will verify these values and throw a `DeserializationException` if they are incorrect.

Serial ID. The serial representation of each class should be assigned an identifier. The purpose of this identifier is to differentiate representations. For example, one could look at the identifier and know that a `LinkedList<T>` is being read instead of a `HashMap<K,V>`. Also, the Serial ID can be changed if the data structure code is updated and requires an update to the serial representation.

For this lab, I've given you serial IDs as constants in the `LinkedList<T>` and `HashMap<K,V>` classes.

Content byte count. This is the number of bytes needed to serialize the object, not including the Serial ID of the content byte count itself.

For the `String` class, the content byte count is the value `length()`, since this is number of 1 byte chars use to represent the string. For a `LinkedList<T>`, the number of bytes is more complicated; it is can be calculated by this formula:

$$\text{content byte count} = \texttt{sizeof(int)} + \sum_{i=0}^{n-1} \texttt{serialize(element}_\texttt{i}\texttt{)}$$

which is the storage for the list size and the size of each element's serial representation.

Note: the content byte count stored in the serial representation is different from the one returned by the `(de)serialize` method. The latter will be the total number of bytes for the entire serial representation *including* the Serial ID and content byte count.

5 Example

A map of type `HashMap<Character,Integer>` representing the individual character count of the word "aardvark" is: `[ 'a' => 3, 'r' => 2, 'd' => 1, 'v' => 1, 'k' => 1 ]`. If the number of buckets is 7 and the load factor limit is 0.75, then the serial representation is:

0x1234	41	7	0.75	5	r	2	d	1	v	1	a	3	k	1

6 Requirements

1. Implement serializers for classes `LinkedList<T>` and `HashMap<K,V>`.

2. Make sure that the pre-conditions and post-conditions to the (de)serialization methods are observed. You should be able to serialize and deserialize multiple objects to a binary file in sequence.

3. Write a short program to test your serializers. Include this in your submission.

7 Hints

- You should start testing your code on a structure that uses simple types, for example a `LinkedList<Character>`, since it's easier to interpret the binary file contents. Make sure you eventually test more complicated types, for example `HashMap<String,String>`.
- In the serialization process, to write the content byte count in the header of the representation you will need to move backwards in the file (since you only know the content byte count after serializing the elements/entries). You can do this by moving the *position* cursor to specific positions in the file. Use the `position()` and `positions(int)` methods of the `SeekableByteChannel` class[2].

Hand-in Checklist

- ☐ The program is clear and well commented.
- ☐ The project directory contains all of the source files, including your program to test the serialization.
- ☐ Zip the project folder and submit it using Léa.

[2] `http://docs.oracle.com/javase/7/docs/api/java/nio/channels/SeekableByteChannel.html`