420-406-AB Data Structures
Winter 2019

May 2nd, 2019
Worth 10%

Project: Serialization

(Due on on Léa)

Overview

For the project, you will be adding *serialization* to a selection of objects and data structures we studied in class. Serialization is the process of translating an object of data structure residing 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.

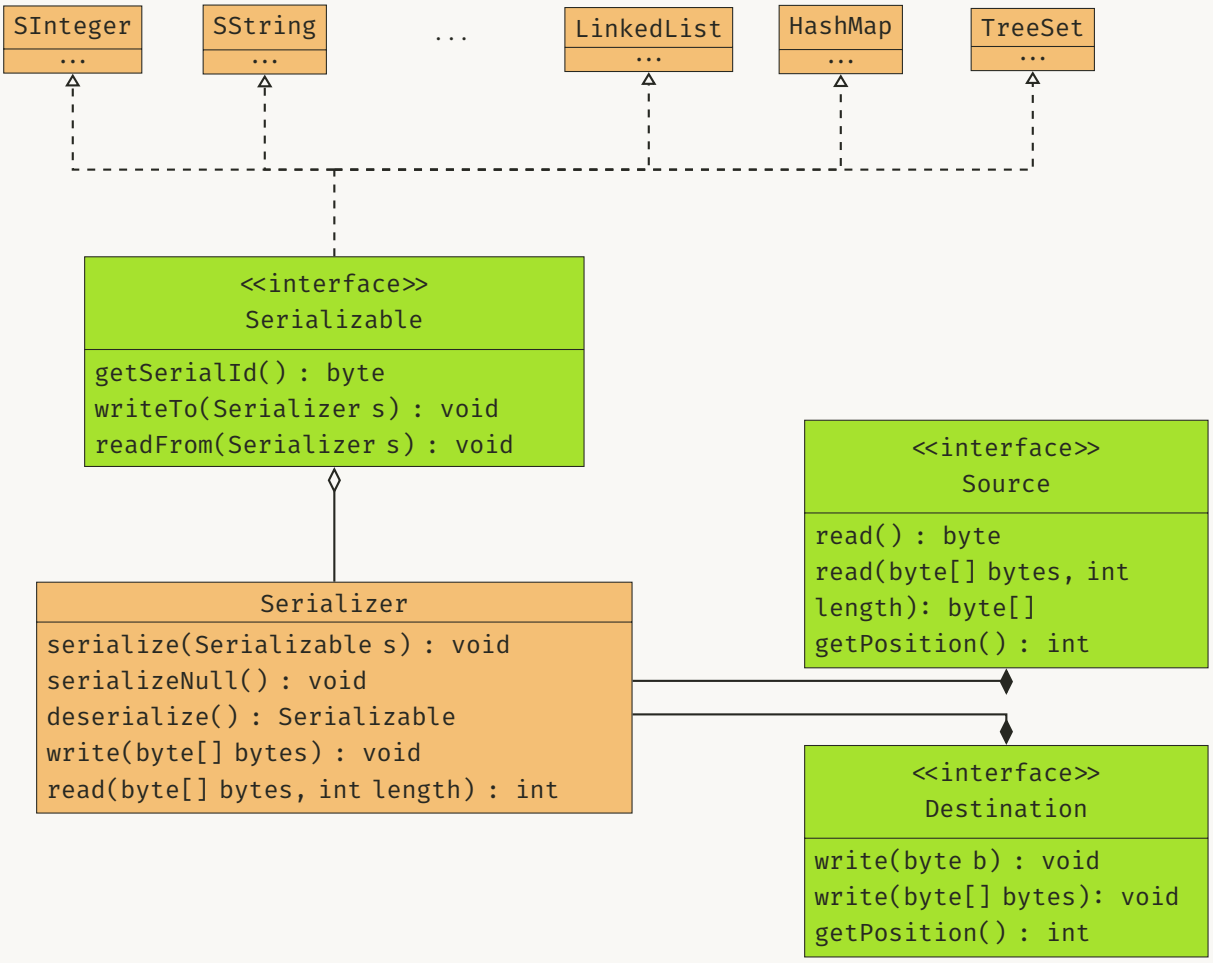
You will write serialization and deserialization operations for some of the classes in this course.

Example. Here is the serial representation of the string “hello world”.

0x02	0x0000000b	0x68	0x65	0x6c	0x6c	0x6f	0x20	0x77	0x6f	0x72	0x6c	0x64
id	length	h	e	l	l	o		w	o	r	l	d

1 Project Design

The design of the project is given in the following UML class diagram. You will be writing classes that implement the `Serializable` interface and will update the `Serializer` class. You will need to use these classes to demonstrate that your serialization works.



1.1 `Serializable` interface

Classes to be serialized will need to implement the `Serializable` API. It has the following operations:

Prototype	`byte getSerialId()`
Purpose	Get this classes unique serialization ID, used to identify the object when deserializing.
Pre-conditions	None.
Post-conditions	None.

Prototype	`void writeTo(Serializer s)`
Purpose	Write the binary representation of the object to the provided serializer.
Pre-conditions	A serialization header has been written to the output (see below).
Post-conditions	A binary representation of the object has been written to the output. This representation is sufficient to create an identical copy when deserializing.

Prototype	`void readFrom(Serializer s)`
Purpose	Read the binary representation of the object from the provided serializer.
Pre-conditions	The serialization header has been read from the input, and validated (see below).
Post-conditions	The binary representation has been read from the input and the object state has initialized as an identical copy of the serialized version.

1.2 `Serializer` class

While each class is in charge of generating their serial representation, a `Serializer` will manage the interaction with the input source and output destination. All serializable classes will be passed a serializer to write/read their representations to.

The `Serializer` API is:

Prototype	`void serialize(Serializable s)`
Purpose	Write a record to the output destination for the serializable.
Pre-conditions	None.
Post-conditions	A record header is written to the output destination, followed by the serial representation.

Prototype	`Serializable deserialize()`
Purpose	Read a record from the input source and return the object it represents.
Pre-conditions	None.
Post-conditions	The record header is read and validated, the object is constructed and the serial representation is used to initialize it.

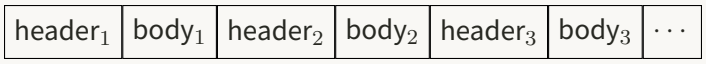
Prototype	`void write(byte[] bytes)`
Purpose	Write the provided bytes to the output destination.
Pre-conditions	None.
Post-conditions	The bytes are written to the output destination.

Prototype	`byte[] read(int n)`
Purpose	Read the next `n` bytes from the input source.
Pre-conditions	None.
Post-conditions	The next `n` bytes are read from the input source and returned.

Prototype	`byte[] read()`
Purpose	Read the next sequence of bytes from the input source.
Pre-conditions	None.
Post-conditions	The number of bytes, `n`, is read from the input source. Then, the next `n` bytes are read from the input source and returned.

1.3 The record header

Each serialized object will have a *record header* that is written to the output destination during serialization and then read from the input source during deserialization:



So the output is a series of records separated by headers. The header will contain the object's class *serial ID* and in Section 3 you will add more.

1.4 Serial IDs

Each `Serializable` class must have a unique ID given to it. In this project, we will use `byte` IDs, but in practice larger numbers are used[1]. This ID serves two important purposes in deserialization:

1. It is a *sanity* check, which means that when deserializing data, we can check that the thing we're about to deserialize is what we expect to be deserializing. Recall that binary data has many interpretations. This ID will help us make sure we have what we want.

[1] Java's provided serialization uses `longs`.

2. We will use the serial ID to create an instance of the correct class. We need this instance to call the `readFrom()` method and initialize the object from it's serial representation.

The second item means we need some mechanism to create objects. This can be done in many ways, but a straightforward one is to have a static method inside the `Serializer` that will switch on the id and create the object:

```
public static Serializable getSerializableById(byte id) {
     switch (id) {
         case 0×01:
             return new SInteger();
         case 0×02:
             return new SString();
          ...
         default:
             throw new SerializationException("Unknown serial ID.");
     }
 }
```

2 `Serializable` Classes

Here is a list of classes to make `Serializable`. Some I've done already as examples for you to study in order understand how to design serial representations for the others. All package names are relative to `ca.qc.johnabbott.cs406` in the starter project.

Class name	Notes
`serialization.util.SInteger`	**Provided.** Serializable wrapper class for integers.
`serialization.util.SString`	**Provided.** Serializable wrapper class for `String`. Serial representation is an `int` and a `char[]`.
`serialization.util.Box`	**Provided.** A generic class that uses the fact that its content is serializable.
`serialization.util.SDate`	Serializable wrapper class for `java.util.Date`. Hint: use a date's numeric representation.
`serialization.util.Grade`	A simple data classes using fields you can already serialize.
`collections.Either`	A generic interface, whose implementations store one of two possible values of two distinct types. To ensure that all implementing classes are serializable, add the line `extends Serializable` to the interface declaration and implement the required methods.
`collections.list.LinkedList`	Serialization should store all the necessary data to reconstruct the linked list. Deserialize without calls to `add(elem)` or `add(pos, elem)`.
`collections.map.HashMap`	Serialization should store all the necessary data to reconstruct the hash map, including the number of buckets and the maximum load factor. Deserialize without calls to `put(key, value)` or even recomputing the bucket. Hint: store and load all colliding records together.
`collections.set.TreeSet`	Serialization should store and load not only the set data, but also the tree structure they are in. Deserialize without calls to `add(elem)`. Hint: use one of the tree traversal algorithms to write and subsequently reconstruct the tree.

For each class, describe its serial representation in the class comment at the top of the `.java` file.

Advice. Although your submission must meet the above restrictions, i.e.: not calling operations like `add( .. )` and `put( .. )`, maybe your first attempt at implementation can call them to see if you're on the right track.

3 Optimizing References

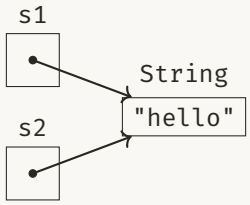
Recall from the course that the building blocks of data structures (and classes) are references, and that we can use references to create object aliases. We want a way to represent aliases,

both to ensure identical original and deserialized data and to save on the size of the output, by removing duplicates.

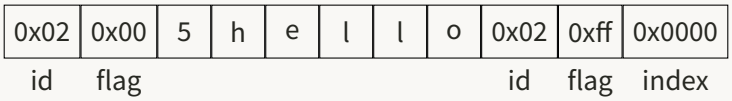
We will implement this by adding information to the record header. Each header will now contain a *flag* to indicate if it the body is a serial representation or an alias. If it's an alias, then the body is simply an integer indicating the byte index of the record containing the object. In the `Serializer` class, I've added constants:

```
public static final byte ALIAS_MARKER = (byte) 0×ff;
public static final byte ORIGINAL_MARKER = 0×00;
```

Example. For the following,



serializing `s1` then `s2`, we would get:



The last field, the byte index, would depend on where in the output destination `s1` had been serialized.

3.1 Hints

I suggest you add two maps to the `Serializer` class. The first is a `Map<Serializable,Integer>` that will store the byte index of each object that has been serialized to the output destination. The second is a `Map<Integer,Serializable>` that will store the deserialized objects by their byte index in the input source.

The first map is a `IdentityHashMap`[2], the second is the usual `HashMap`. The difference between these two classes is that, when searching by key, the identity version uses $=$ instead of `.equals()`. When serializing, you will keep track of the things you've previously serialized, and when you are asked to serialize something you've already done you can perform the optimization.

[2] Here you will need to use the Java libraries, since we did not implement the `IdentityHashMap`.

4 Optimizing Immutables

A class that has no *mutators*, i.e.: methods that change the state of the object after initiaization, is called *immutable*. Storage of immutables can also be optmized: since they never change, any two immutables that are equal will always be equal, therefore you only need to store a single copy! Use this fact about immutables when deserializing to reduce the number of objects created in memory.

4.1 Hints

Coming soon...

Requirements

Your program *must* meet the following requirements:

1. Your program should be clear and well commented. It must follow the "420-406 Style Guidelines" (on Léa).
2. Start from the provided classes.
3. Implement `Serializable` for the remaining classes in Section 2. Document the format that you use in code comments.
4. For `LinkedList<T>` deserialization, do not call `add(elem)` or `add(pos,elem)`.
5. For `HashMap<K,V>` do not call `put(key,value)`.
6. For `TreeSet<T>` do not call `add(elem)`.
7. Update `Serializer` to optimize references as described in Section 3.
8. Update `Serializer` to optimize immutables as described in Section 4.
9. Submit using Git by following the instructions (on Léa).