Comp 533 - Assignment 5: Extendible and Multi-Platform Object (De)Serialization in GIPC

Date Assigned: April 6, 2017

Completion Date: Thu May 4, 2017

This assignment embodies several concepts including basic serialization, serialization-extensibility, multi-class recursive descent parsing, logical vs. physical structures, multi-platform serialization, reflection, textual vs binary representations, type-dependent vs type-independent serializers, space and time performance of different serializers, and graph creation and manipulation algorithms.

You will build an extensible (de) serialization mechanism, which requires you to set up a complex recursive-descent parsing/unparsing scheme involving objects of different classes. The serializer will also allow non-tree data structures to be serialized and de-serialized. In addition, it will support a reflection-based scheme that is more flexible than Java's and can support heterogeneous-platforms. Your serialization scheme will communicate both binary and textual representations of objects. The former is more efficient while the latter is easier to debug. Unlike the RMI serializer, your serializer will (de) serialize the logical rather than physical structure of an object. This will allow serialization of an instance of I1 to be deserialized into an instance of I2. Such a feature is particularly useful for communication among programs written in different languages.

You will not be able to serialize (and deserialize) arbitrary objects. The objects you can handle will be called *serializable* objects. An extendible serialization scheme together with different sections given below address the various sets of objects you should handle and provide an incremental path for implementation.

Here are some references for this assignment:

Parsing and Grammars	PowerPoint PDF YouTube	More Inheritance Chapter
Java Object Communication	PPT PDF YouTube Mix	
Issues in Serialization	PPT PDF	Serialization Demo

	YouTube-1 YouTube-2 YouTube-3 YouTube-4 Mix-1 Mix-2 Mix-3 Mix-4	

Tags

To guide your implementation and to autograde it (possibly) you are required to (a) put a special annotation called Tags (`util.annotations.Tags`) and (b) define methods following signatures given to you. You are encouraged to choose your own method and class names as long you follow these constrains. The values of the tags you pass to the Tags annotation are defined in interface `util.annotations.Comp533tags`. You can use import static to easily refer to them.

The use of tags is illustrated below for both classes and interfaces.

This assignment refers to a type with an unknown name having known tag T as <T>.

External Value-Serializers and Tracing

Value serializers encode and decode value serializations of objects of different types. They implement an interface following the constraints given below:

```
@Tags({VALUE_SERIALIZER})
public interface <Your Type Name> {
  void objectToBuffer (Object anOutputBuffer,
      Object anObject, <Your Collection Type> visitedObjects)
      throws NotSerializableException;

  Object objectFromBuffer(Object anInputBuffer,
      Class aClass, <Your Collection Type> retrievedObjects)
      throws StreamCorruptedException, NotSerializableException;
}
```

The methods in this interface are similar to the ones in the Serializer interface (discussed in the logical serializer section) except both of them take extra arguments specifying the (a) buffer to be used for serializtion or deserialization, passed by the caller of the methods, and (b) the collection of objects visited or retrieved so far, to support non-tree structures. The deserialzer method takes a third extra argument specifying the class of the object to be returned.

The actual type of buffers will be StringBuffer and ByteBuffer in the textual and binary serialization schemes, respectively. To determine which serialization scheme should be

used, a value serializer should use the **instance of** operation on the passed input or output buffer.

In each class implementing it, trace the start and end of the methods `objectToBuffer` and `objectFromBuffer` using the following calls:

```
ExtensibleValueSerializationInitiated.newCase(…)

ExtensibleValueSerializationFinished.newCase(…)

ExtensibleBufferDeserializationInitiated.newCase(…)

ExtensibleBufferDeserializationFinished.newCase(…)
```

The traces of these can be turned on by calling `ExtensibleSerializationTraceUtility.setTracing()`.

All trace classes specific to this assignment are in: `port.trace.serialization.extensible.`

Serializer Registry

Create a static class, called a serializer registry, that allows various serializers to be registered and found.

The class should have the tag given below, and methods with the name and signature also given below:

```
@Tags({SERIALIZER_REGISTRY})
public class <Your Class Name>  …{

public static void registerValueSerializer (Class aClass,
<VALUE_SERIALIZER> anExternalSerializer);
}
public static <VALUE_SERIALIZER> getValueSerializer (Class aClass) {

//other methods
…
}
```

The same value serializer can be registered for multiple types, as shown in the next section.

This class will also allow type-independent serializers (mentioned below) to be registered and found: dispatching serializer, null serializer, Bean serializer, list pattern serializer, and array serializer. For each of these serializers it will define static editable properties. A class has a static or instance editable property P of type T if it has a static or instance getter and setter for it, with the following signatures:

T getP();

void setP(T newVal);

The static properties it will have are DispatchingSerializer, ArraySerializer, BeanSerializer, ListPatternSerializer, Enum serializer, NullSerializer. The dispatching serializer is of type <DISPATCHING_SERIALIZER>. The others are of type <VALUE_SERIALIZER>. This means, for instance, that the serializer registry will have the following methods for the Array serializer:

```
public static void registerArraySerializer (<VALUE_SERIALIZER>
anExternalSerializer);
public static <VALUE_SERIALIZER> getArraySerializer ();
```

Predefined Value Serializers

You must implement and register value serializers for the base classes (String, Integer, Short, Long, Double, Float, Boolean), three implementations of java.util.Colllection (java.util.HashSet, java.util.ArrayList, and java.util.Vector) two implementations of java.util.Map (java.util.Hashtable, java.util.Hashmap). As Java automatically converts between primitive and wrapper values, you will also be able to handle **int** and **double** primitive values.

The nature of the predefined value serializers and their tags is illustrated in the predefined registrations below:

<VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(Integer.**class**,
new <INTEGER_SERIALIZER());

VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(Short.**class**,
new <SHORT_SERIALIZER>());

VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(Long.**class**,
new <LONG_SERIALIZER>());

<VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(Boolean.**class**,
new <BOOLEAN_SERIALIZER>());

<VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(Double.**class**,
new <DOUBLE_SERIALIZER>());

<VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(Double.**class**,
new <FLOAT_SERIALIZER>());

<VALUE_SERIALIZER_REGISTRY>..*registerSerializer*(String.**class**,
new <STRING_SERIALIZER>());

<VALUE_SERIALIZER_REGISTRY>.*registerSerializer*(HashSet.**class**,
new <COLLECTION_SERIALIZER>());

```
<VALUE_SERIALIZER_REGISTRY>.registerSerializer(ArrayList.class,
new < COLLECTION_SERIALIZER ());

<VALUE_SERIALIZER_REGISTRY>.registerSerializer(Vector.class,
new <COLLECTION_SERIALIZER> ());

<VALUE_SERIALIZER_REGISTRY>.registerSerializer(HashMap.class,
new < COLLECTION_SERIALIZER> ());

<VALUE_SERIALIZER_REGISTRY>.registerSerializer(Hashtable.class,
new <COLLECTION_SERIALIZER> ());
```

Consistent with what was said before:

new <Tag Name>()

means

new C()

where C is a class with the tag <Tag Name>.

Dispatching Object Serializer

To serialize a (non-null) serializable object of class C, you will need to send the name of the class followed by a representation of the object value. Thus the full serialization of an object consists of its class name serialization and its value serialization. To de-serialize an object you will convert the class name to a class and then instantiate the class and change its instance variables.

The value serializers handle (de)serialization of only values, not full object serialization. Relatedly, deserialization method in the value serializers takes as an argument the class of the object to be deserialized. To handle serialization and deserialization of class names, dispatching to the appropriate value serializers, and passing the class to the deserialization method, we need yet another serializer, called the dispatching serializer:

```
@Tags({DISPATCHING_SERIALIZER })
public interface <Your Type Name> {
  void objectToBuffer (Object anOutputBuffer,
      Object anObject, <Your Collection Type> visitedObjects)
      throws NotSerializableException;

  Object objectFromBuffer(Object anInputBuffer,
      <Your Collection Type> retrievedObjects)
      throws StreamCorruptedException, NotSerializableException;
}
```

It is exactly like the type-specific value serializer except the deserializer method does not take the class argument. The dispatching serializer interface will be implemented by a single class. *Give this class the same tag as the interface.*

The dispatching serializer will not only call the value serializer but also will be called by the latter, consistent with recursive-descent parsing. This is in the spirit of, for example, the parser of a <statement> calling the parser of an <if statement>; and the latter, in turn, calling the former to parse the <then part> and <else part> of the <if statement>.

As mentioned above, to support this serializer, you need the following methods in your serializer registry:

```
public static <DISPATCHER_SERIALIZER> getDispatchingSerializer();

public static void  registerDispatchingSerializer
      (<DISPATCHER_SERIALIZER>  newVal);
```

Value Representation of Base and Composite Types

Integer, double, Boolean and String values are based types and the others are composite types. You are free to choose how these are represented.

The representation of a composite type will essentially be a concatenation of the representation of its components. Some of these serializers will include the number of components before the concatentation of the component serializations.

To guide your implementation and your recursive calls, it might be useful to create a grammar describing your serialization scheme. Here is an outline:

<Object Serialization> → <ClassName> <Value Serialization>
<Value Serialization> → <Int Serialization> | <Null Seralization>| <Collection Serialization> | …
<Collection Serialization> → <Collection Size> <Object Serialization>*

The dispatching serializer will be responsible for <Object Serialization> and the value serrializers for <Value Serialization>.

As mentioned above, based on the kind of buffer that is passed, your value serializers should create the binary or textual representation. Support text serialization first and then add support for the more efficient binary once your recursive steps are working.

Binary Serialization: ByteBuffer provides putInt(), getInt(), putShort(), getShort(), putLong(), getLong(), putDouble(), getDouble(), putBoleean(), getBoolean() for encoding and decoding value serializations of base integer, double and Boolean values. To put a string, you can call the getBytes() method to convert it into a string and then use put(byte[]) method of ByteBuffer. Similarly, to extract a string of length l, you can allocate a byte array of length l and call get(byte[]) method to fill the array, which can be passed to a String constructor.

Textual Serialization: Your value serializer should be able to manipulate StringBuffer instances in addition to ByteBuffer instances. You can convert int, double and Boolean

values into strings using the Java toString() method and use appropriate parsing methods in wrapper classes (such as Integer.parseInt()) to deserialize these values.

Null Serializer

The null value is an instance of all classes, so it needs special handling. Define and register a null serializer for handling null values. The tag of its class should be <NULL_SERIALIZER>.

The null serializer is free to use any scheme for serializing and de-serializing values. For example it can use `"Null Class"` for the type name and `"null"` as the value to follow the format described above.

The dispatching serializer will need a way to determine the type name to be used for null values. You can add a method to the interface of the null serializer that returns this name, such as:

public String getNullClassName();

You can also take the simpler approach of using some predefined constant in the dispatching serializer.

Textual and Binary Logical Serializer Selection

You will define two classes that implements the following GIPC interface:

```
package serialization;
import java.nio.ByteBuffer;
public interface Serializer {
      public ByteBuffer  outputBufferFromObject(Object object);
      public Object objectFromInputBuffer(ByteBuffer inputBuffer) ;
}
```

The first method serializes an object into a byte buffer, and the second does the reverse.

These two classes are called logical serializers as they use the logical rather than physical structure of the serialized objects. They will support textual and binary serializations, and have the tags ***LOGICAL_TEXTUAL_SERALIZER*** and ***LOGICAL_BINARY_SERALIZER*** respectively. The factories that instantiate them will have the tags ***LOGICAL_TEXTUAL_SERALIZER_FACTORY*** and ***LOGICAL_TEXTUAL_SERALIZER_FACTORY*** respectively.

Thus, the class supporting textual serialization will have the header:

```
@Tags({LOGICAL_TEXTUAL_SERALIZER})
public class <Your Name> implements Serializer {
…
}
```

and the factory instantiating this serializer will have the header:

```
@Tags({LOGICAL_SERALIZER_FACTORY})

public class <Your Name> implements SerializerFactory {
…
}
```

Your serializer will be used by GIPC when you register an instance of this factory by calling the setter method of `serialization. SerializerSelector.` The customization code given for the previous assignment in the package `examples.gipc.counter.customization` shows how serializers can be cdhanged.

The difference between the textual and binary logical serializers is that they will pass a StringBuffer and ByteBuffer, respectively, to the dispatching seralizer methods.

Either logical serializer, should, however, return and receive ByteBuffer objects from GIPC components that access it. As you have seen, a ByteBuffer is a wrapper around an array, with markers indicating the position, limit and capacity. The serialization methods should access these markers in an appropriate fashion. In particular, the deserialization methods should not assume that the first byte to be read is the first byte in the array.

To minimize copying and data structure allocation the serialization method in a logical serializer should return the same ByteBuffer (stored as an instance variable) each time it is called. This means, before it starts, it should set the position of the buffer to 0 and its limit to the capacity. *Before returning the buffer to GIPC, it should make a flip() call to the buffer it writes so that it can be read by GIPC.*

Type-Independent Serializers

The scheme, as described above, does not extend to classes for which serializers have not been registered, or allow a serializer to handle types whose names it does not know. Extend this scheme to support serialization of enums, arrays, and objects that follow the Bean and list patterns described in the class material. We will refer to these serializers as type-independent serializers.

In your value-serializer registry, define four new registration methods to handle these three kinds of objects, as illustrated below:

```
<VALUE_SERIALIZER_REGISTRY>.registerEnumSerializer(new
<EMUM_SERIALIZER>());
<VALUE_SERIALIZER_REGISTRY>.registerArraySerializer(new
<ARRAY_SERIALIZER> ());
<VALUE_SERIALIZER_REGISTRY>.registerBeanSerializer(new <BEAN_SERIALIZER>
());
<VALUE_SERIALIZER_REGISTRY>.registerListPatternSerializer(
    new <LIST_PATTERN_SERIALIZER>());
```

The tags of the four serializers are given below. Define also getter methods for these four properties.

It is possible for an object such as an ArrayList to be serialized both by a type-dependent serializer and a type-dependent one. Choose the type dependent one.

Your Bean and List-pattern serializers should allow only Bean and List instances of java.io.Serializer to be serialized. Encountering some other kind of object should throw the exception `java.io.NotSerializableException`. You should make an exception for null, which is not a Serializable. Thus, you should allow null to be sent, as indicated above.

Enum Serializer

To handle enums, you can use: Class.isEnum() to determine if a value of type class is an enum class. Also given an enum object, you can call, e.getEnumConstants() to get the enum constants. The toString() method of these constants gives the String representation of each constant. You can also cast enum object, e, to Enum and execute Enum.valueOf(e,enumStringRepresentation) to get the enum object associated with enumStringRepresentation.

Array Serializer

You can call the isArray() method on the class of an object to determine if it is an array. You can call Array.newInstance(Class componentClass, int size) to instantiate a new array. Array.length(Object array), Array.get(Object array, int index)and Array.set(Object array, int index, Object value) can be used to read and write and elements of instantiated arrays.

Bean Serializer

To handle beans, you should use the java.beans.BeanInfo class and invoke the invoke the getPropertyDescriptors() method in it to get each property and its read and write method. You can assume that BeanInfo returns properties in the same order on all computers – so you do not need to write property names.To get the BeanInfo of a class, invoke the method Introspector.getBeanInfo(class) method.

Your Bean serializer should not die when Bean get read/write methods misbehave; it should just skip them in serialization. Do print the error, if any. A missing write method (e.g. for the Class property) is not an error - such a property is inherently transient. Also, it should not serialize objects whose read methods are associated with the util.misc.Transient annotation (not be confused the Java beans annotation with the same name), which is checked by invoking RemoteReflectionUtility.isTransient(Method) call.

List-Pattern Serializer

To support the list pattern, use the ObjectEditor class util.misc.RemoteReflectionUtility. The relevant methods are isList(), listGet(), listAdd(), and listSize() methods.

Basic Reflection

The method getClass() can be invoked on any object to get the class of an object, and the method getName() can be invoked on a class to get its name. The static method Class.forName(String) can be used to convert a class name of an unknown class to a Class object, and the method newInstance() can be invoked on a class to create a new instance of the class. The instantiated class must have a null constructor, which will be assumed for classes of all composite objects.

The predefined classes can be instantiated directly in your code as in:

new Integer(anInt)

new ArrayList();

InitSerialized

After a Bean or List-Pattern object is serialized, invoke the method initSerializedObject() in it, if such a method exists, which can be done by invoking RemoteReflectionUtility. invokeInitSerializedObject(Object).

Non-Tree Data Structures

Using the composite types, it is possible to create structures that are general graphs rather than trees. In a tree, a node has a single parent, while a graph allows multiple parents. This means that in a graph, the same node may be visited multiple times in a descent through the serialized/de-serialized structure. The de-serialized object should be isomorphic to the serialized object. This means that the serialization should contain a reference rather than a value when a component is visited again, and this reference should be converted to a memory address of an existing deserialized object at the destination. If you ensure that the serialization and deserialization traversals serialize and de-serialize corresponding objects in the same order, this scheme can be easily and efficiently implemented. As in the case of the null value, you must define some way to pass a reference, such as a special type name and a position. This can be hardwired in your dispatching serializer, which will fill and consume visited objects and retrieved objects. You can use hashtables or lists as the collections storing these objects.

Alternate Serialization

Allow an object implementing an interface to be de-serialized as an instance of another class that implements the same interface. For instance, allow an ArrayList to be deserialized as a Vector. The alternate class is registered by giving the classes of the serialized and deserialized objects, as in:

<VALUE_SERIALIZER_REGISTRY>.registerDeserializingClass(ArrayList.class, Vector.class)

Alternate serialization along with type-independent serializers makes your scheme multi - platform.

Serialization Tester

The GIPC class examples.serialization.SerializationTester includes a main method that has test cases with a comment delineating the two parts.

Submission

Single-Process Traces and Output: Create a YouTube video demonstrating your code running serialization.examples.SerializationTester with tracing turned on. Use traces and the output to show that the method runs correctly with your code. Demonstrate both the textual and binary serialization schemes.

Multi-Process Sample Code Traces and Output: Modify the example in the customization package by replacing the custom serializer with your serializer. Turn on tracing of serialization steps. Run the modified client and server and use the output of the server and client console windows to show that your implementation currently implements the required semantics. Again, do this for both textual and binary logical serialization. Compare the sizes of the byte buffers created in the two schemes implemented and the default scheme by looking at the output of the last `ObjectSerializationFinished` trace in each of the three schemes.

Multi-Process Simulation Experiments: Run the 3-client 500-command, simulation experiments on your local computer using both the default GIPC implementation and your new implementations of the serializer and compare the timings of the bytebuffers created in the three schemes.

As usual, also submit the code on Sakai.