CS 213 – Spring 2016

Lecture 15 – Mar 8

Serialization and Versioning

Persistence and Serialization

- Data that is used by a program can be saved (and retrieved) in many ways:
  - Save the data in a file in some ad hoc format
  - Save the data in a database
  - Save the state of objects in a program, which implicitly saves the data that is being used by the program

- Any combination of these options may be adopted by a software system, depending on the sharing of data between applications, the programming language used, and whether the program is an application that runs on the user's local machine, or is a remote service

- Serialization is a way of implementing persistence that saves entire Java objects, i.e. class and state

Why/What is Serialization

- Serialization is especially useful in saving "work in progress" between user sessions of an application that runs locally - that is, there is a notion of the "state" in which the user left the application at the end of a session, from where they want to start off the next session

- Serialization in Java is implemented by saving objects to a stream in binary format - the stream is a file, or a byte array, or a stream associated with a TCP/IP socket

- Distributed programs can communicate objects by serializing them

- It is called serialization because each object is given a serial number in the stream: On the first encounter with an object, it is completely written out, along with a serial number. At every subsequent encounter, only the serial number is written.

Serialization Example: Set Up

```
package geometry;

public class Point implements
Comparable<Point > {
   int x,y;
   public Point(int x, int y) {

      this.x = x; this.y = y;
   }
   ...
 }
```

```
package geometry;

public class ColoredPoint
extends Point{
   String color;
   public ColoredPoint(int x, int y
                       String color) {

      super(x,y); this.color = color;
   }
   ...
}
```

Serialization Example: Set Up

```
package geometry;
import java.util.*;
public class GeomApp {
   private ArrayList<Point> points;
   public GeomApp() {
      points = new ArrayList<Point>();
   }
   public void addPoint(Point p) {
      points.add(p);
   }
   public void writePoints() {
      for (Point p: points) {
          System.out.println(p);
      }
   }
   public static void main(String[] args) {
      GeomApp gapp = new GeomApp();
      gapp.addPoint(new Point(1,2));
      gapp.addPoint(new ColoredPoint(2,3,"green"));
      gapp.addPoint(new ColoredPoint(3,4,"blue"));
      gapp.addPoint(new Point(4,5));
      gapp.addPoint(new Point(5,6));
      gapp.writePoints();
   }
}
```

Want to save points that have been loaded into GeomApp so they can be reloaded in next session: serialize GeomApp instance

Serialization Example: Storing Session (Points)

```
package geometry;
import java.io.*;
import java.util.*;

public class GeomApp implements Serializable {
   private ArrayList<Point> points;

   public static final String storeDir = "dat";
   public static final String storeFile = "points.dat";

   public GeomApp() { points = new ArrayList<Point>(); }
   public void addPoint(Point p) {points.add(p);}
   public void writePoints() {
      for (Point p: points) {System.out.println(p);}
   }

   public static void writeApp(GeomApp gapp)
   throws IOException {
      ObjectOutputStream oos = new ObjectOutputStream(
          new FileOutputStream(storeDir + File.separator + storeFile));
      oos.writeObject(gapp);
   }

   ...
}
```

To be serialized, must implement `java.io.Serializable` (an empty interface)

File in which to store objects

Open `java.io.ObjectOutputStream` to file and write object to it

Serialization Example: Storing Session (Points)

```
package geometry;
import java.io.*;import java.util.*;
public class GeomApp implements Serializable {
   private ArrayList<Point> points;

   public static final String storeDir = "dat";
   public static final String storeFile = "points.dat";

   ...

   public static void writeApp(GeomApp gapp) throws IOException {
      ObjectOutputStream oos = new ObjectOutputStream(
          new FileOutputStream(storeDir + File.separator + storeFile));
      oos.writeObject(gapp);
   }

   public static void main(String[] args) {
      GeomApp gapp = new GeomApp();
      gapp.addPoint(new Point(1,2));
      gapp.addPoint(new ColoredPoint(2,3,"green"));
      gapp.addPoint(new ColoredPoint(3,4,"blue"));
      gapp.addPoint(new Point(4,5));
      gapp.addPoint(new Point(5,6));
      gapp.writePoints();
      writeApp(gapp);
   }
}
```

The GeomApp object and its constituents (recursively) are serialized and written out to the points.dat file

Serialization Process Sequence

- When an object is serialized using the `writeObject` method of `ObjectOutputStream`, the class itself is serialized first, with the class name and signature (identifying information)

- The serialization of a class starts at the highest serializable superclass in its hierarchy, and walks down the inheritance chain, serializing each subclass in turn.

  For instance, say you serialize a `ColoredPoint` object.

  The highest superclass of `ColoredPoint` is `Object`, but `Object` is not serializable.

  The class below this is `Point`, which should implement the `Serializable` interface, in which is case it will be serialized.

  Following this, the `ColoredPoint` class will be serialized.

Serialization Process Sequence

- After the class has been serialized, all non-transient and non-static primitive field values in the object are written to the output stream (data can be marked "transient" to prevent them from being serialized, e.g. temporary, non-state related data)

  In serializing a `ColoredPoint` object, the values of the `int` fields `x` and `y` are written to the output stream.

- If a reference to an object is encountered, writeObject is called recursively on the referenced object, and a serial number is created - if the same reference is encountered again, then only the serial number is written

  In serializing a `ColoredPoint` object, the `String` field color is written out with a serial number, following which the `String` object is serialized recursively.

  Note: `String` MUST be marked `Serializable` as well (and it is)!

Serialization Process Sequence

Serializing the `GeomApp` object

oos.writeObject(gapp);

`GeomApp` is the highest serializable class in its hierarchy. The class is written out.

The `GeomApp` object has a single field, which is an object reference:

```
ArrayList<Point> points
```

The `ArrayList` class written out (it implements the `Serializable` interface)

The `points` field is given a serial number and written to output stream

Then the `ArrayList<Point>` object is recursively serialized

Serialization Process Sequence

Recursive serialization of the `ArrayList<Point>` object

Each contained `Point` and `ColoredPoint` object is serialized:

- The `Point` class is written out, then the first `Point` object

-The `ColoredPoint` class is written out, which will start with the `Point` (super) class to be written out, but since it's already written out, only `ColoredPoint` is written, followed by the two `ColoredPoint` instances

- The last two `Point` objects are written out

(Classes `Point` and `ColoredPoint` must implement the `Serializable` interface, otherwise they will not be serialized.)

Deserializing (Reconstructing) objects from storage

```
...
public class GeomApp implements Serializable {
   private ArrayList<Point> points;
   ...

   public static GeomApp readApp()
   throws IOException, ClassNotFoundException {
      ObjectInputStream ois = new ObjectInputStream(
          new FileInputStream(storeDir + File.separator + storeFile));
      GeomApp gapp = (GeomApp)ois.readObject();
      return gapp;
   }

   public static void main(String[] args) {
      GeomApp gapp = GeomApp.readApp();
      gapp.writePoints();
   }
}
```

Recreating an object requires recreating both class and object state, and transitively, the class and object states of all referenced objects

The sequence of class reconstruction is the same as that of serialization: start with highest superclass in hierarchy, then walk down the hierarchy reconstructing each class in turn

Stream-Unique Identifier (SUID)

- Say you first write the store method code in GeomApp, then serialize a GeomApp object.

- After serializing, say you add the load method in GeomApp. This changes the version of GeomApp, so when you try to deserialize (load) the previously serialized GeomApp object, you eot an exception:

  java.io.InvalidClassCastException
  geometry.GeomApp; local class incompatible:
  stream classdesc serialVersionUID = ...
  local class serialVersionUID = ...

- This basically says that the current (local) version of GeomApp, is different than the version found in the stream from which it is being deserialized (loaded), and because of this incompatibility, the deserialization fails

- The versions are found to be different because they have different "stream unique identifiers", or SUIDs.

Stream-Unique Identifier (SUID)

- The virtual machine determines the version of a Serializable class by generating a stream-unique identifier (SUID)

- Java has a utility program called serialver that determines the version id of an object:

```
> serialver geometry.Point
  Geometry.Point: static final long serialVersionUID = 6062634463206984397L
```

(In Eclipse, mouse over the warning to the left of a serializable class header, and it gives you options toadd a custom serial version ID or generated serial version ID)

Serialization and Versioning

- Serialization saves class name, version identifier, and other information necessary to reconstruct it

- Following version incompatibilities may arise:
  - Local class is newer than serialized version, i.e. serialized object was generated by an older version of class
  - Local class is older than serialized version, i.e. serialized object was generated by a newer version of class

- When either of these mismatches is encountered during deserialization, an InvalidClassCastException is thrown, that says the version (of serialized object) is not compatible with local class

- To fix the problem with the GeomApp class, just have Eclipse generate a default id, which is written as a field of the class:

```
static final long serialVersionUID = 1L
```

The JVM will use this value instead of the default value it would otherwise generate, so that the serialized version and the local version will be forced to have the same ID of 1.

Handling Version (In)Compatibilty

- Use the generated serial version UID, or the default SUID such as 1L – the 'L' is for long integer constant
- When you make minor changes to your class, determine if the changes would make the class incompatible – if not, retain the same version
- If a data field is added in the new version of a class:
  - When reading data serialized by the old version, this field will not be found
  - The field will be initialized to the default value for its type
- If a data field that was in the old version is deleted in the new version of a class:
  - When reading data serialized by the old version, this field will be discarded
- If you make changes that renders a class incompatible with earlier version, generate a new serial version UID, and update the `serialVersionUID` static field – whether a change makes a class incompatible or not is a design decision.