CS 213 – Software Methodology

Sesh Venugopal

Notes on Serialization and Versioning

Data Persistence

- Data that is used by a program can be saved/retrieved in various ways:
    - Save the data in a file in some ad hoc custom format, or some standard format such as JSON or XML

    - Save the data in a database

    - Save the *state of objects* in the program, which implicitly saves the data that is in the fields of various classes/instances

- Various factors may determine which of these (could be more than one) options are used in a software system:

    - Complexity of implementation (depends on type of data)

    - Sharing of data between applications (need a standard way. e.g JSON/XML)

    - Programming language used

    - Whether application is single or multiuser (simultaneous)

    - Whether application is running locally or is a remote service

Persistence : Serialization

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

Why/What is Serialization

- Serialization is especially useful in saving "work in progress" between user sessions of an application that runs locally:
  - 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 continue in 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: Setup

```
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: Setup

```java
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 ObjectOuputStream(
            new FileOutputStream(storeDir + File.separator + storeFile));
        oos.writeObject(gapp);
    }
    public static void main(String[] args) {
        ...
    }
}
```

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)

```java
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

Deserializing (Reconstructing) objects from storage

```java
...
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) throws Exception {
        GeomApp gapp = GeomApp.readApp();
        gapp.writePoints();
    }
}
```

Stream-Unique Identifier (SUID)

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

- After serializing, 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 get 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 to add 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.