CS108, Stanford   Handout #32
Fall, 2008-09   Osvaldo Jiménez

Doc Save/Open XML

Thanks to Nick Parlante for much of this handout

Serialization / Archiving

- Common problem of writing/reading objects between memory and persistent storage
- State in memory -- objects
- Write objects to streamed state
    - To a disk file, or across the network, or to the system clipboard
    - The notion of "address space" does not hold in the streamed form -- there are no pointers. Just a block of bytes.
- Read
    - Read the streamed form, and re-create the object in memory
- There are many words for writing an object to a streamed form
    - Writing
    - Persisting
    - Pickling
    - Flattening
    - Streaming
    - Archiving

Old Style Memory/Disk Streaming

- Translate back and forth by hand
- Typically use an ASCII text format
    - Custom arrangement between your data structures and some ASCII format for reading and writing.
    - Write custom code to read and write between the memory form and the streamed form
    - e.g. `DBRecord`

Java Automatic Serialization

- `Serializable` interface
    - By implementing this interface, a class declares that it is willing to be read/written by the automatic serialization machinery.
    - This use of an interface is a bit of a hack, but it works. It marks which classes participate. If designed today, no doubt it would use a java annotation.
    - Serialization is not the most efficient in terms of cpu cost or space, but it is very easy.
- Automatic Writing
    - The system knows how to recursively write out the state of an object
    - Recursively follows pointers and writes those objects out too (they must in turn be `Serializable`). In this way, it writes out the whole tree of objects.
- Built-Ins
    - Most built ins know how to serialize: `int`, array, `Point`, ...
- Fields declared with the `transient` modifier are skipped by serialization
- Use the `transient` reserved word on an instance variable to prevent the serialization from recurring down a branch you do not want written out. A transient ivar comes back as `null` after reading.
- Override: `readObject()`, `writeObject()` -- to put in more customized reading/writing
- Versioning of the class

  - Serialization can detect version changes to the class when reading and refuse to read. Programmer can control this if they wish.
  - This make serialization fragile without care -- data written with class code version N will fail to read back into the class version N+1 by default.

Strategy -- Data Struct

- Create a little public struct class that contains plain data you want to send
- Make it `Serializable`
- Use object streams below to read and write it easily

ObjectOutputStream

- Create an object output stream wrapped around any other stream. Then can write objects onto that stream.
- e.g. `out = new ObjectOutputStream(` <regular file or socket output stream> `);`
- This is the "decorator" pattern -- wrapping something of interface X in another thing, also of interface X

out.writeObject(obj)

- Suppose "out" is an `ObjectOutputStream`
- `out.writeObject(obj);`
- This one line calls the automatic serialization machinery to write out everything rooted at the given object.
- Classes
  - Each written object will be identified by its class -- the reading code will need those same classes to read the stream.
  - The written class should be `public` (so the receiver can know it)
  - The written class should not be inner, since that will try to write the outer object too. It can be a nested (`static`) class however.
- Arrays work
  - For a collection of things, it may be easier to arrange all the things into a single array that can be written in one operation.
- Transient
  - Fields should be declared `transient` if they should not be written. They will read back in as `null`.
- MVC
  - Write out the data model, not the view.
  - May also want to write out a simplified or canonical version of the data model -- so you can revise your real internal data model over time, without breaking file compatibility.

No Duplicates / Automatic Detection

- The automatic serialization detects duplicates in the stream -- objects that are written more than once. That is, a single object that is written multiple times is only recorded once in the stream.
- If a second or later instance of an object is written to the stream, a reference to the first instance is instead put in the stream, instead of a 2nd copy.
- The reading code uses this information to correctly re-create the pointer graph in memory.

out.writeUnshared(obj)

- `writeUnshared(obj)` -- writes out a fresh copy of the given object, even if that object was previously written to the stream. However, the no-duplicates property will still hold for objects referenced from inside the given object.

ObjectInputStream

- Create an `ObjectInputStream` wrapped around any type of stream
- `ObjectInputStream in = new  ObjectInputStream( <input stream> );`

in.readObject()

- CT type
  - Read back with the same compile time type it was written (`Object[]` or `String[]`)
- Class
  - If a class was written that is not present at read-time, there will be an error.
  - If the class has the same name but a changed implementation there will be an error.
  - It's safest to serialize classes that are stable everywhere such as `Array` and `Point`

GUI Document Theory

- GUI Shell -- same every time
  - The outer GUI shell of frames, buttons, toolbars that surrounds the interesting data
  - Don't save the GUI shell in each document -- you can re-create it when needed
- Core Model/View system
  - There is a core data model the user sees and edits through some View
  - The **model** is what we need to save. The rest we can re-create when the file is opened later.
- Dirty Bit
  - True if the model has been changed since the last save or open
  - Controls whether you get the "Do you want to save" query when closing the window or opening a new file into that window
  - Dirty bit: false just after open and just after save, changes to true on any edit and for a new, unsaved document

File Edit/Save/Open Operations

- These are rules for doc/open/save that you have seen many times in applications you use to create documents.
- Edit
  - Any edit operation ... calls setter on model, sets `dirty=true` in the model. The setter also typically calls `repaint()`, causing the pixels to update to show the new model state.
  - Looking at objects, selecting, scrolling ... none of these should set the `dirty` bit
- Save
  - Write the data model out, omit the shell
  - Set `dirty=false`
- Open
  - Create a new, empty shell
  - Read the file model data, creating and setting up new view objects based on the read in model (consider bottlenecking through the same method through which the user adds new data).
  - Set `dirty=false`
- New
  - New is very similar to of Open, and just don't read in a file
- `File lastSaved` -- the document stores its last saved or opened File (a Java `File` object is the pathname of a file, not an open file stream). When doing a save operation, it automatically sets

`lastSaved` to the written file. A "Save as" disregards the `lastSaved` file and prompts for a new one, which then becomes the `lastSaved`. A new document has a `lastSaved` of `null`, so we know to prompt the user for a file when the new document is saved for the first time.

Java "Bean" Style

- A simple, standard way of packaging some data in an object in a way that makes it easy for others to use that data.
- Java Beans come down to a convention on how methods are named
- The bean class exposes a public set of "properties"
- Suppose we want to have a `String` property called `url`, then the class must have a getter that looks exactly like `String getUrl()` and a setter that looks exactly like `void setUrl(String)`.
- The type (e.g. `String`) must be the same in the getter and setter.
- So given a bean class, there is a default way to deduce what properties it exposes by looking at the pattern of public methods starting with "get" and "set".
- A bean class should have a zero-arg constructor that creates a bean in a minimally functional "empty" state. In the new/empty state, the getters should work, although they may return default null/0/false/"" type values. Use the setters to put in real values.

XML

- XML is a standard, text based way to encode structured information -- it's a great interchange format to move data between systems.
- XML is not so great for day-to-day storage -- it's bulky and somewhat slow. However, it is great at interchange and if you just need a format without thinking about things too much.
- Note that XML is a mostly a **syntax** definition -- just because two applications read and write XML does not make them automatically compatible. There's still the real work of making them agree about the **semantics** of the XML elements. With that limitation in mind, XML is still a valuable technology that helps data handling and interchange (although it's gotten a lot of hype these last few years!).
- Java has many detailed ways to generate and parse XML (all languages now have XML support)
- We are just going to use Java's automatic bean <-> XML facility

XML Encoder/Decoder - Serialization Alternative

- http://java.sun.com/products/jfc/tsc/articles/persistence4/
- `XMLEncoder` -- write a java bean as XML
- `XMLDecoder` -- given XML, read the data back into a bean
- Uses the idea of a java bean -- in a way, a very elegant design. Use the public getter/setter interface exposed by an object as the basis to automatically write its data to XML.
- The bean class and its getters and setters should be `public` -- the encode/decode uses the public interface of the class.
- To count as a property to write, the property must have both a getter and a setter, and they must use the same type. Other methods are ignored.

Encode(obj) -- Bean -> write XML

- `XMLEncoder encode()` queries the bean by calling its getters, and writes those values out (e.g. a method named `getFoo()` yields a `String` which is written out). Only a property with getter/setter is written out.
- If a value is unchanged from the default for that class of object, `encode()` does not write the value out

  - The encoder figures this out by making a new empty version of the object with the default constructor, and comparing the property to write vs. the property in the new empty object. It only needs to write out the property if it differs from the default -- clever!

- By default it writes all the properties -- it's possible to customize the list of properties to write (see example `xmlOut.setPersistenceDelegate()` below).
- The encoding is recursive -- if a property is a java bean, it is in turn written out.

Decode(obj) -- read XML -> Bean

- `XMLDecoder decode()` -- given XML, recreate the bean object
- Given xml, look at the class to create
- Creates a new "empty" instance of that class, and then call setters (e.g. `setFoo()`) to build it back up to the original state
- In the example below, the default color for a dot is black. Compare the color property of the first two dots in the XML. They have the default black color, so the system figures out that it doesn't have to write anything for that property.

Smart about Duplicates -- Graph Save

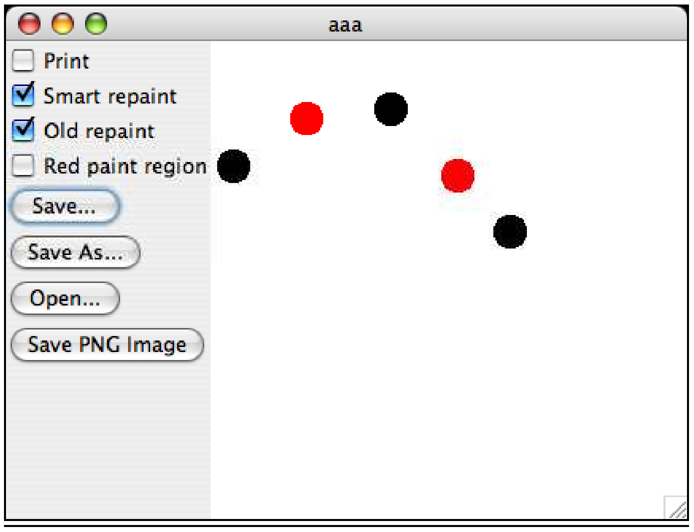
- The encoder is smart about only including one copy of an object that is written to the stream multiple times. Note how the color red in the example below is only encoded once. The second appearance of the Red object is just written as a reference to the first red object.
- Essentially, the decoder starts with one "root" object, and follows links to save all the reachable objects, while being smart about the case where an object has already been saved.
- The "read" decode operation recreates the object graph in memory, with all the pointers correct.
- Java "serialization" also has this feature of being smart about saving the graph of objects.

Advantages Of XML Encode/Decode

- You have a bunch of java objects forming your data model
- In your code, you make decisions about what getter/setter properties to expose as the public data of the model
- The XML encoder leverages that work -- using the same properties, so you don't have to think about your save format.
- Note that you can upgrade your data model objects and maintain backward compatibility -- so long as the new model object supports all the old properties (i.e. you maintain compatibility with your old getter/setter design), you can read your "old" data model files with no change -- neat!
- The main disadvantage is that XML is a bulky format -- if you have a huge volume of data, XML is costly. However, for quick and reliable results, XML can be a good choice.
- Also, for debugging, XML has the nice quality that it is easy to look through to see what is going on.

Dots Save/Open/XML Example

- The model in the dots program is a `List` of `DotModel` objects.
- In this case, the model is passive -- it sits there and does storage. The swing code calls getters/setters on the model, calls `repaint()`, does file i/o using the model data. (This is not as active as the `TableModel` which manages its own list of listeners.)



```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_05" class="java.beans.XMLDecoder">
 <array class="DotModel" length="5">
  <void index="0">
   <object class="DotModel">
    <void property="x">
     <int>14</int>
    </void>
    <void property="y">
     <int>78</int>
    </void>
   </object>
  </void>
  <void index="1">                // THE RED DOT
   <object class="DotModel">
    <void property="color">
     <object id="Color0" class="java.awt.Color">
      <int>255</int>
      <int>0</int>
      <int>0</int>
      <int>255</int>
     </object>
    </void>
    <void property="x">
     <int>60</int>
    </void>
    <void property="y">
     <int>48</int>
    </void>
   </object>
  </void>
  <void index="2">
   <object class="DotModel">
    <void property="x">
     <int>113</int>
    </void>
    <void property="y">
     <int>42</int>
    </void>
   </object>
  </void>
  <void index="3">
   <object class="DotModel">
    <void property="color">
     <object idref="Color0"/>
    </void>
    <void property="x">
     <int>155</int>
    </void>
    <void property="y">
     <int>84</int>
    </void>
   </object>
  </void>
  <void index="4">
   <object class="DotModel">
    <void property="x">
     <int>188</int>
    </void>
    <void property="y">
     <int>119</int>
    </void>
   </object>
  </void>
 </array>
</java>
```

Save to PNG

- Create image object
- Create `Graphics` pointing to it
- Call `paintAll()` -- very unusual to **call** paint, but this is the one case where we do it
- See `saveImage()` below

Dots Model Code

```
// DotModel.java
/*
 Contains the data model for a single dot: x,y, color
 Uses the "bean" getter/setter style, so works with the XML encode/decode.
 Has a zero-arg constructor.
*/

import java.awt.Color;

public class DotModel {
    private int x;
    private int y;
    private Color color;

    public DotModel() {
        x = 0;
        y = 0;
        color = Color.BLACK;
    }


    // standard getters/setters for x/y/color
    public Color getColor() {
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }

    // Convenience setters for clients

    // Moves x,y both the given dx,dy
    public void moveBy(int dx, int dy) {
        x += dx;
        y += dy;
    }

    // Sets both x and y
    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

File Save DotPanel Code

```
    /**
     Accessors for the dirty bit.
     */
    public boolean getDirty() {
        return dirty;
    }
    public void setDirty(boolean dirty) {
        this.dirty = dirty;
    }


    /**
     Saves out our state (all the dot models) to the given file.
     Uses Java built-in XMLEncoder.
     */
    public void save(File file) {
        try {
            XMLEncoder xmlOut = new XMLEncoder(
                    new BufferedOutputStream(
                            new FileOutputStream(file)));

            // Could do something like this to control which
            // properties are sent. By default, it just sends
            // all of them with getters/setters, which is fine in this case.
            //  xmlOut.setPersistenceDelegate(DotModel.class,
            //       new DefaultPersistenceDelegate(
            //           new String[]{ "x", "y", "color" }) );


            // Make a DotModel array of everything
            DotModel[] dotArray = dots.toArray(new DotModel[0]);

            // Dump that whole array
            xmlOut.writeObject(dotArray);

            // And we're done!
            xmlOut.close();
            setDirty(false);
            // cute: only clear dirty bit *after* all the things that
            // could fail/throw an exception
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }


    /**
     Reads in all the dots from the file and set the panel
     to show them.
     */
    public void open(File file) {
        DotModel[] dotArray = null;
        try {
            // Create an XMLDecoder around the file
            XMLDecoder xmlIn = new XMLDecoder(new BufferedInputStream(
                    new FileInputStream(file)));
            // Read in the whole array of DotModels
            dotArray = (DotModel[]) xmlIn.readObject();
            xmlIn.close();

            // Now we have the data, so go ahead and wipe out the old state
            // and put in the new. Goes through the same doAdd() bottleneck
            // used by the UI to add dots.
            // Note that we do this after the operations that might throw.
            clear();
            for(DotModel dm:dotArray) {
                doAdd(dm);
            }
            setDirty(false);

        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }


    /**
     Saves the current appearance of the DotPanel out as a PNG
     in the given file.
     */
    public void saveImage(File file) {
        // Create an image bitmap, same size as ourselves
        BufferedImage image = (BufferedImage) createImage(getWidth(), getHeight());

        // Get Graphics pointing to the bitmap, and call paintAll()
        // This is the RARE case where calling paint() is appropriate
        // (normally the system calls paint()/paintComponent())
        Graphics g = image.getGraphics();
        paintAll(g);
         g.dispose();  // Good but not required--
         // dispose() Graphics you create yourself when done with them.

        try {
            javax.imageio.ImageIO.write(image, "PNG", file);
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```