Serialization

A Java object normally expires when the program that created it terminates. Since no variable refers to it, the garbage collector reclaims its storage.

Problem: Want to make an object be persistent so that it can be saved between program executions.

A Possible Solution

If all of the instance variables (fields) in the object are of primitive types or Strings, we can use

- Methods from DataOutputStream (writeInt(), writeDouble(), writeUTF(), etc.) to store the object.

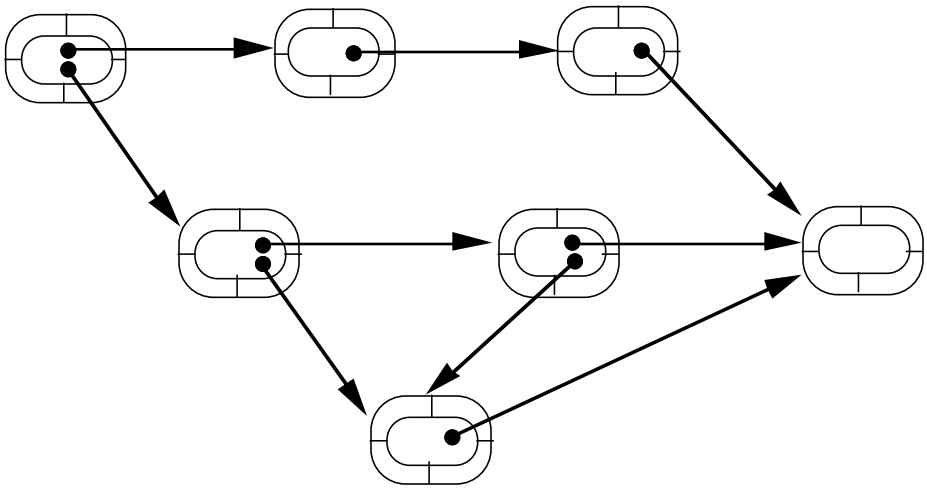
- Methods from DataInputStream (readInt(), readDouble(), readUTF(), etc.) to restore the object.

Difficulties

1. What if objects contain arrays of varying sizes?
2. What if objects belong to various subclasses of some class so that different objects have different sets of fields (consider subclasses of Employee)?
3. What if instance variables are references to other objects (composition), which have references to still other objects, and so on?

Imagine a graph of objects that lead from the object to be saved.

The entire graph must be saved and restored.



We need a byte-coded representation of objects that can be stored in a file external to Java programs, so that the file can be read later and the objects can be reconstructed.

Serialization provides a mechanism for saving and restoring objects.

Serialization

Serializing an object means to code it as an ordered series of bytes in such a way that it can be rebuilt (really a copy) from that byte stream.

The serialization mechanism needs to store enough information so that the original object can be recreated including all objects to which it refers (the object graph).

Java has classes (in the java.io package) that allow the creation of streams for object serialization and methods that write to and read from these streams.

Only an object of a class that implements the empty interface java.io.Serializable or a subclass of such a class can be serialized.

What is Saved

- The class of the object.
- The class signature of the object.
- All instance variables *not* declared **transient**.
- Objects referred to by non-transient instance variables.

If a duplicate object occurs when traversing the graph of references, only one copy is saved, but references are coded so that the duplicate links can be restored.

Uses of Serialization

- Means to make objects persistent.
- Means to communicate objects over a network.
- Means to make a copy of an object.

Saving an Object (an array of dominoes)

1. Open a file and create an ObjectOutputStream object.

```
ObjectOutputStream save =
        new ObjectOutputStream(
            new FileOutputStream("datafile"));
```

2. Make Domino serializable:

```
class Domino implements Serializable
```

3. Write an object to the stream using writeObject().

```
Domino [] da = new Domino [55];
// Create a set of 55 Domino's and  place them in array.
```



```
save.writeObject(da);     // Save object (the array)
save.flush();             // Empty output buffer
```

Restoring the Object

1. Open a file and create an ObjectInputStream object.

```
ObjectInputStream restore =
            new ObjectInputStream(
                new FileInputStream("datafile"));
```

2. Read the object from the stream using *readObject* and then cast it to its appropriate type.

```
Domino [] newDa;
      // Restore the object:
newDa = (Domino [])restore.readObject();
```

or

```
Object ob = restore.readObject();
```

When an object is retrieved from a stream, it is validated to ensure that it can be rebuilt as the intended object.

Validation may fail if the class definition of the object has changed.

Conditions

- A class whose objects are to be saved must implement interface Serializable, with no methods, or the Externalizable interface, with two methods.
- The first superclass of the class (maybe Object) that is not serializable must have a no-parameter constructor.
- The class must be visible at the point of serialization.

The implements clause acts as a tag indicating the possibility of serializing the objects of the class.

Main Serialization Methods

Saving Objects

public ObjectOutputStream(OutputStream out)
throws IOException

Special Exceptions:

SecurityException
untrusted subclass illegally overrides security-sensitive methods

public final void writeObject(Object obj)
throws IOException

Special Exceptions:

InvalidClassException
Something is wrong with a class used by serialization (contains unknown datatypes or no default constructor).

NotSerializableException
Some object to be serialized does not implement the Serializable interface.

public void flush() **throws** IOException

Writes any buffered output bytes and flushes through to the underlying stream.

public void close() **throws** IOException

Restoring Objects

public ObjectInputStream(InputStream in)
throws IOException, SecurityException

Special Exceptions:

StreamCorruptedException
stream header is incorrect

SecurityException
untrusted subclass illegally overrides security-sensitive methods

public final Object readObject()
throws IOException,
ClassNotFoundException

Special Exceptions:

ClassNotFoundException // not an IOException
Class of serialized object cannot be found.

InvalidClassException
Something is wrong with a class used for serialization (probably a definition change).

StreamCorruptedException
Control information in stream violates internal consistency checks.

OptionalDataException
Primitive data was found in the stream instead of objects.

public void close() **throws** IOException

Primitive Data

ObjectOutputStream and ObjectInputStream also implement the methods for writing and reading primitive data and Strings from the interfaces DataOutput and DataInput, for example:

writeBoolean	readBoolean
writeChar	readChar
writeInt	readInt
writeDouble	readDouble
writeUTF	readUTF

Some Classes that Implement Serializable

String	StringBuffer	Calendar	Date
Character	Boolean	Number	Class
Point	Component	Color	Font
Throwable	InetAddress	URL	ArrayList
LinkedList	HashSet	TreeSet	HashMap

Note: No methods or class variables are saved when an object is serialized.

A class knows which methods and static data are defined in it.

Drawing Shapes Example

Create a surface to draw shapes on, so that the current set of drawings can be saved in a file and restored later.
Use Swing Set graphics.

Components in the Program

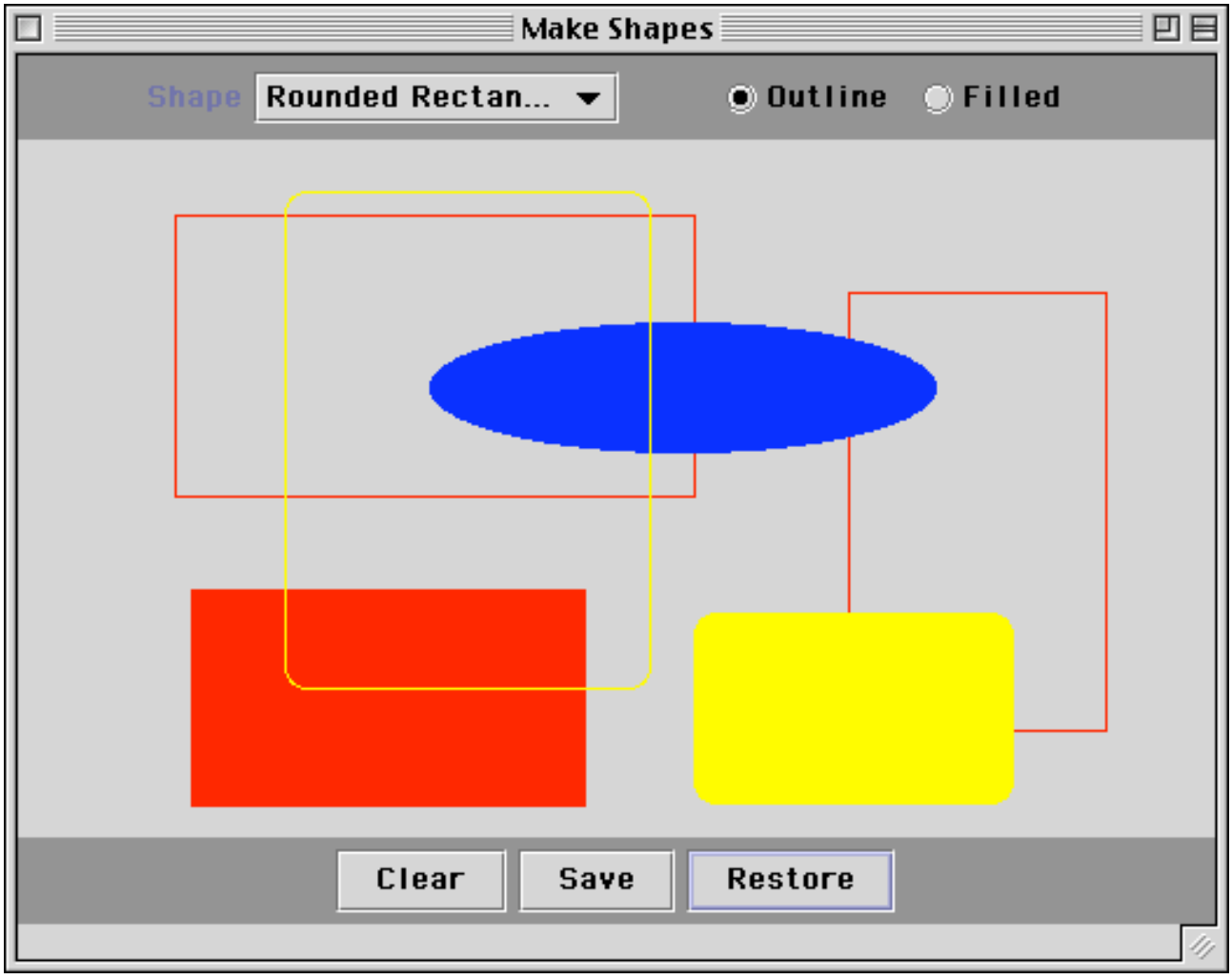
- JPanel acting as a drawing surface.
- JComboBox object that allows the user to pick a shape (rectangle, oval, or rounded rectangle).
- Radio buttons to choose between an outline shape or a filled shape.
- Save button that causes the current drawings to be written to a file.
- Restore button that recalls the saved drawings.
- Clear button that erases the drawing surface.

Shapes

The shapes will be objects that are each responsible for drawing themselves on the surface.

The three shapes will be instances of three concrete classes that extend an abstract class called Shape.

The state of the drawing surface is kept as a list of the shapes that have been drawn on it.



A shape object is determined by the point at its upper left corner, its width and height, and whether it is filled or not.

```
abstract class Shape implements Serializable
{
    protected Point spot;
    protected int width, height;
    protected boolean isFilled;

    abstract void draw(Graphics g);
}
```

Each of the concrete classes provides an implementation for the abstract method.

```
class Rectangle extends Shape
{
    Rectangle(Point p, int w, int h, boolean fill)
    {  spot = p;  width = w;  height = h;  isFilled = fill;  }

    void draw(Graphics g)
    {
        g.setColor(Color.red);
        if (isFilled)
            g.fillRect(spot.x, spot.y, width, height);
        else
            g.drawRect(spot.x, spot.y, width, height);
    }
}

class Oval extends Shape
{
    Oval(Point p, int w, int h, boolean fill)
    {  spot = p;  width = w;  height = h;  isFilled = fill;  }

    void draw(Graphics g)
    {
        g.setColor(Color.blue);
```

```
        if (isFilled)
            g.fillOval(spot.x, spot.y, width, height);
        else
            g.drawOval(spot.x, spot.y, width, height);
    }
}

class RoundRect extends Shape
{
    RoundRect(Point p, int w, int h, boolean fill)
    {   spot = p;  width = w;  height = h;  isFilled = fill;   }

    void draw(Graphics g)
    {
        g.setColor(Color.yellow);
        if (isFilled)
           g.fillRoundRect(spot.x, spot.y, width, height, 20, 20);
        else
           g.drawRoundRect(spot.x, spot.y, width, height, 20, 20);
    }
}
```

Event Handling

Window closing and mouse clicks will be handled by inner classes inside of the main class.

The JComboBox and the radio buttons will be handled by having the main class implement ItemListener.

The three single buttons (Clear, Save, and Restore) will be handled by having the class implement ActionListener.

Structure of the Main Class

```
public class MakeShapes extends JFrame
             implements ActionListener, ItemListener
{
    // Instance and class variables

    public static void main(String [] args) { … }

    MakeShapes() { … }

    public void itemStateChanged(ItemEvent e)
    { … }

    public void actionPerformed(ActionEvent e)
    { … }


// Inner Classes

    class WindowHandler extends WindowAdapter
    { … }

    class DrawPanel extends JPanel
    {
        DrawPanel() { … }

        public void paintComponent(Graphics g)
        { … }

        class MouseHandler extends MouseAdapter
        { … }
    }
}
```

Header and Fields in Main Class

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class MakeShapes extends JFrame
                        implements ActionListener, ItemListener
{
    private static String fileName; // external file

    private java.util.List shapeList;

    private JComboBox shape;
    private JRadioButton outline, filled;
    private JButton clear, save, restore;

    private JPanel drawPanel;

    private String currentShape = "Rectangle";
    private boolean isFilled = false;
        :
```

The default shape is an outline rectangle.

These parameters can be changed using the shape JComboBox and the radio buttons.

Constructor for the Frame

Constructor builds the GUI interface and registers some event handlers with appropriate components.

```
MakeShapes()
{
    setTitle("Make Shapes");
    shapeList = new ArrayList();

    addWindowListener(new WindowHandler());

    Container cp = getContentPane();
    setBackground(Color.lightGray);

    JPanel p1 = new JPanel();
    p1.setBackground(Color.gray);
    p1.add(new Label("Shape"));
    shape = new JComboBox();
    shape.addItem("Rectangle");
    shape.addItem("Oval");
    shape.addItem("Rounded Rectangle");
    p1.add(shape);
    shape.addItemListener(this);

    ButtonGroup bg = new ButtonGroup();
    outline = new JRadioButton("Outline", true);
    p1.add(outline);
    outline.addItemListener(this);
    bg.add(outline);

    filled = new JRadioButton("Filled", false);
    p1.add(filled);
    filled.addItemListener(this);
    bg.add(filled);
    cp.add(p1, "North");

    drawPanel = new DrawPanel();
    cp.add(drawPanel, "Center");
```

```
        JPanel p2 = new JPanel();
        p2.setBackground(Color.gray);
        clear = new JButton("Clear");
        clear.addActionListener(this);
        p2.add(clear);
        save = new JButton("Save");
        save.addActionListener(this);
        p2.add(save);
        restore = new JButton("Restore");
        restore.addActionListener(this);
        p2.add(restore);
        cp.add(p2, "South");
    }
```

Main Method

The main method instantiates a MakeShapes frame, sets its size and visibility, and provides a name for the external file that will be used for serialization.

As an extension to the program, the user could be asked to provide a name for the file.

```
    public static void main(String [] args)
    {
        MakeShapes ms = new MakeShapes();
        ms.setSize(500, 400);
        ms.setVisible(true);
        fileName = "SavedShapes";
    }
```

Inner Class for Window Closing

```
class WindowHandler extends WindowAdapter
{
    public void windowClosing(WindowEvent evt)
    {  System.exit(0);  }
}
```

DrawPanel

Contains the overriden *paintComponent* method, which draws each of the shapes in the list by calling their *draw* methods.

Note the application of dynamic binding

This inner class contains its own inner class, the mouse event handler.

```
class DrawPanel extends JPanel
{
    private Point pointA, pointB;

    DrawPanel()
    {
        addMouseListener(new MouseHandler());
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Iterator it = shapeList.iterator();
        while (it.hasNext())
        {
            Shape s = (Shape)it.next();
            s.draw(g);
        }
    }
```

Mouse Clicks (inside DrawPanel)

```
class MouseHandler extends MouseAdapter
{
    public void mousePressed(MouseEvent e)
    {
        pointA = new Point(e.getX(), e.getY());
    }

    public void mouseReleased(MouseEvent e)
    {
        pointB = new Point(e.getX(), e.getY());
        int x = Math.min(pointA.x, pointB.x);
        int y = Math.min(pointA.y, pointB.y);
        int w = Math.abs(pointA.x - pointB.x);
        int h = Math.abs(pointA.y - pointB.y);
        Shape newShape;
        if (currentShape.equals("Rectangle"))
            newShape =
                    new Rectangle(new Point(x,y), w, h, isFilled);
        else if (currentShape.equals("Oval"))
            newShape =
                    new Oval(new Point(x,y), w, h, isFilled);
        else
            newShape =
                    new RoundRect(new Point(x,y),w,h,isFilled);
        shapeList.add(newShape);
        drawPanel.repaint();
    }
}
```

JComboBox and Radio Button Handling

Methods in MakeShapes

```
public void itemStateChanged(ItemEvent e)
{
    if (e.getSource() == outline)
                                    isFilled = false;
    else if (e.getSource() == filled)
                                    isFilled = true;
    if (e.getSource() == shape)
            currentShape = (String)e.getItem();
}
```

Button Handling

Clear Button

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == clear)
    {
        shapeList.clear();
        drawPanel.repaint();
    }
```

We shown two ways of saving the list of Shapes.

1. Object by object.
2. Entire ArrayList at once.

Save Button (1)

```
else if (e.getSource() == save)
{
    try
    {   FileOutputStream fos =
                        new FileOutputStream(fileName);
        ObjectOutputStream oos =
                        new ObjectOutputStream(fos);

        oos.writeInt(shapeList.size());

        for (int k=0; k<shapeList.size(); k++)
            oos.writeObject(shapeList.get(k));

        oos.close();          // also flushes
    }
    catch (IOException ex)
    {
        System.out.println("Trouble writing list");
    }
}
```

Restore Button (1)

```
else if (e.getSource() == restore)
{
    try
    {   FileInputStream fis =
                         new FileInputStream(fileName);
        ObjectInputStream ois =
                        new ObjectInputStream(fis);
        int len = ois.readInt();
        shapeList.clear();
```

```
            for (int k=0; k<len; k++)
            {
                Object newShape = ois.readObject();
                shapeList.add(newShape);
            }

            ois.close();

            drawPanel.repaint();       // redraw the surface
        }
        catch (Exception ex)
        {
                System.out.println("Trouble reading list");
        }
```

Save Button (2)

```
    else if (e.getSource() == save)
    {
        try
        {   ObjectOutputStream oos =
                new ObjectOutputStream(
                        new FileOutputStream(fileName));
            oos.writeObject(shapeList);
            oos.close();
        }
        catch (IOException ex)
        {
            System.out.println("Trouble writing list");
        }
    }
```


Restore Button (2)

```
else if (e.getSource() == restore)
{
    try
    {   ObjectInputStream ois =
                new ObjectInputStream(
                    new FileInputStream(fileName));
        shapeList = (java.util.List)ois.readObject();
        ois.close();
        drawPanel.repaint();
    }
    catch (Exception ex)
    {
        System.out.println("Trouble reading list");
    }
}
```

MouseMotion

The next page shows an attempt to have the shapes drawn as the mouse is moved.

It works fairly well with Swing.

Add one line to the constructor for DrawPanel:

```
addMouseListener(new MotionHandler());
```

and put the next class inside DrawPanel.

```
class MotionHandler extends MouseMotionAdapter
{
    Shape prevShape;
    public void mouseDragged(MouseEvent e)
    {
        pointB = new Point(e.getX(), e.getY());
        int x = Math.min(pointA.x, pointB.x);
        int y = Math.min(pointA.y, pointB.y);
        int w = Math.abs(pointA.x - pointB.x);
        int h = Math.abs(pointA.y - pointB.y);

        Shape newShape;
        if (currentShape.equals("Rectangle"))
            newShape =
                new Rectangle(new Point(x,y),w,h,isFilled);
        else if (currentShape.equals("Oval"))
            newShape =
                new Oval(new Point(x,y), w, h, isFilled);
        else
            newShape =
                    new RoundRect(new Point(x,y),w,h,isFilled);

        shapeList.remove(prevShape);
        shapeList.add(newShape);
        drawPanel.repaint();
        prevShape = newShape;
    }
}
```

Class Versioning

What happens if the definition of a class has been altered between the time an object was serialized and is then deserialized?

Some changes have little or no effect on an object of the class:

- class variables
- class methods
- body of instance methods
- addition of an instance variable (use default value)

Some changes prevent the correct restoration of a previously serialized object.

Incompatible Changes

- name of the class
- type of an instance variable
- removing of an instance variable, which includes
  - changing its name
  - making it static or transient
- superclass of the class
- other changes dealing with *readObject*, *writeObject*, and changes between Serializable and Externalizable.

Version ID

To avoid incompatible changes, each class has

a version ID that is included, along with its fully qualified name, with each serialized object of the class.

This number is known as the stream unique identifier (SUID).

The SUID is a hash value of type **long** whose computation depends on the signature of the class members that are neither static nor transient.

The value can be set explicitly by giving a value to the **static final** field *serialVersionUID*.

The JDK comes with a utility command that provides the version ID of a class:

% **serialver Domino**

```
Domino:   static final long
                    serialVersionUID = -1744580429045726511L;
```

InvalidClassException

If an attempt is made to deserialize an object whose stored value of serialVersionUID disagrees with the value belonging to the current version of the class, an InvalidClassException is thrown.

Suppose we want to alter a class definition in a way that will have no substantial effect on the deserialization process but the version ID change prevents deserialization.

Solution: Define serialVersionUID explicitly in the class thereby ignoring the Java versioning mechanism.

If we define serialVersionUID on our own,
we take complete responsibility for version compatibility.

Example: SaveDominoes

The SaveDominoes class provides a way to add dominoes to a persistent List, which is serialized on exit from the main method.

Each time we start up the main method, an attempt is made to restore the previously serialized list of dominoes.

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

public class SaveDominoes
{
    private static String fileName = "DominoFile";
    private static List doms;

    public static void main(String [] args)
        throws IOException, ClassNotFoundException
    {
        doms = thawDominoes();

        BufferedReader in = new BufferedReader(
                            new InputStreamReader(System.in));

        while (true)
        {
            System.out.print("Enter p to print dominoes, "
                            + "a to add a domino, q to quit: ");

            char ch = in.readLine().charAt(0);

            switch (ch)
            {
              case 'p':
                 for (int k = 0; k < doms.size(); k++)
                       System.out.println(doms.get(k));
```

```
                break;

            case 'a':
                System.out.println("Enter domino as" +
                                   "integers on two lines:");
                int v1, v2;
                v1 = Integer.parseInt(in.readLine().trim());
                v2 = Integer.parseInt(in.readLine().trim());
                doms.add (new Domino(v1,v2,true));
                break;

            case 'q':
                freezeDominoes();
                return;
        }
    }
}

private static List thawDominoes()
        throws IOException, ClassNotFoundException
{
    try
    {   FileInputStream fis = new FileInputStream(fileName);
        ObjectInputStream objectIn =
                            new ObjectInputStream(fis);
        List result = (List)objectIn.readObject();
        objectIn.close();
        System.out.println("Thawed from file " + fileName);
        return result;
    }
    catch (FileNotFoundException ex)
    {
        System.out.println("New file");

        return new ArrayList();
```

```
        }
    }

    private static void freezeDominoes() throws IOException
    {
        FileOutputStream fos =
                                    new FileOutputStream(fileName);
        ObjectOutputStream objectOut =
                                    new ObjectOutputStream(fos);
        objectOut.writeObject(doms);
        objectOut.close();
        System.out.println("Frozen to file " + fileName);
    }
}
```

Domino Class

```
class Domino implements java.io.Serializable
{
    private int spots1, spots2;
    private boolean faceUp;

    static final int MAXDOTS = 9;
    private static int numDominoes=0;

    Domino(int v1, int v2, boolean up)
    {   if (0<=v1 && v1<=MAXDOTS) spots1 = v1;
        else spots1 = 0;
        if (0<=v2 && v2<=MAXDOTS) spots2 = v2;
        else spots2 = 0;
        faceUp = up;   numDominoes++;
    }

    Domino()
    {   this(0,0,false); }
```

```
    Domino(boolean up)
    {   spots1 =
            (int)((MAXDOTS+1)*Math.random());
        spots2 =
            (int)((MAXDOTS+1)*Math.random());
        faceUp = up;   numDominoes++;
    }

    int getHigh()
    {   if (spots1>= spots2) return spots1;
        else return spots2;  }

    int getLow()
    {   if (spots1<= spots2) return spots1;
        else return spots2;  }

    public String toString()
    {   String orient = faceUp ? "UP" : "DOWN";
        return "<" + getLow() + ", " + getHigh() + ">  "
                    + orient;
    }

    void flip()
    {   faceUp = ! faceUp; }

    boolean matches(Domino otherDomino)
    {   int a = otherDomino.getHigh();
        int b = otherDomino.getHigh();
        int x = getHigh();
        int y = getLow();
        return  a==x || a==y || b==x || b==y;
    }

    static int getNumber()
    { return numDominoes; }
}
```

Sample Output

```
% java SaveDominoes
New file
Enter p to print dominoes, a to add domino, q to quit: p
Enter p to print dominoes, a to add domino, q to quit: a
Enter domino as integers on two lines:
3
6
Enter p to print dominoes, a to add domino, q to quit: a
Enter domino as integers on two lines:
7
2
Enter p to print dominoes, a to add domino, q to quit: p
<3, 6> UP
<2, 7> UP
Enter p to print dominoes, a to add domino, q to quit: q
Frozen to file DominoFile

% java SaveDominoes
Thawed from file DominoFile
Enter p to print dominoes, a to add domino, q to quit: a
Enter domino as integers on two lines:
0
1
Enter p to print dominoes, a to add domino, q to quit: p
<3, 6> UP
<2, 7> UP
<0, 1> UP
Enter p to print dominoes, a to add domino, q to quit: q
Frozen to file DominoFile
```

Changing Domino

Now comment out the flip() instance method from class Domino. This change has no direct effect on the Domino objects, but note result of executing SaveDominoes with the new version of Domino.

% **serialver Domino** (without flip)

```
Domino:  static final long
        serialVersionUID = 32939169702587020 29L;
```

```
% java SaveDominoes
java.io.InvalidClassException: Domino;
Local class not compatible:
stream classdesc serialVersionUID=-17445804290457265 11
     local class serialVersionUID=32939169702587020 29
    at java.io.ObjectStreamClass.setClass(Compiled Code)
    at java.io.ObjectInputStream.inputClassDescriptor(
                                        ObjectInputStream.java)
    at java.io.ObjectInputStream.readObject(Compiled Code)
    at java.io.ObjectInputStream.readObject(Compiled Code)
    at java.io.ObjectInputStream.inputObject(Compiled Code)
    at java.io.ObjectInputStream.readObject(Compiled Code)
    at java.io.ObjectInputStream.inputArray(Compiled Code)
    at java.io.ObjectInputStream.readObject(Compiled Code)
    at java.io.ObjectInputStream.defaultReadObject(
                                        ObjectInputStream.java)
    at java.io.ObjectInputStream.inputObject(Compiled Code)
    at java.io.ObjectInputStream.readObject(Compiled Code)
    at java.io.ObjectInputStream.readObject(Compiled Code)
    at SaveDominoes.thawDominoes(SaveDominoes.java:21)
    at SaveDominoes.<init>(SaveDominoes.java:12)
    at SaveDominoes.main(SaveDominoes.java:75)
```

Taking Control

The InvalidClassException can be avoided by taking the following steps:

1. Add a line to the Domino class

   static final long serialVersionUID = 1999L;

2. Recompile Domino.
3. Remove the file DominoFile from the current directory.
4. Execute SaveDominoes and create a new domino List.
5. Avoid calling *flip* on the deserialized dominoes.

Now the program works correctly whether *flip* is in the class or not.

Adding instance variables to a class will also change the version ID without invalidating the serialized objects.

When such an object is deserialized, the new instance variables are created with their default values.