The Game of Lolo
An Exploration of Object-Oriented Programming

Timothy A. Budd
Oregon State University

May 4, 2000

Introduction

The game of Lolo is a classic Nintindo game. Lolo is a game in the puzzle tradutuib. Lolo is a creature that lives in a two-dimensional world of grids. Also in the world are walls, water, trees, blocks, and hearts. Lolo can move around the world in any of the four compass directions, but can only occupy an empty square. Lolo cannot jump over walls, or run into trees. When lolo moves into a square in which there is a Heart the score is incremented. The objective is to capture as many hearts as possible. Figure 1 shows a typical Lolo world.

Trees, Walls and water are immovable, but Lolo does have the power to slide a block. When Lolo pushes on a block, if the adjacent square Lolo is pushing the square into is empty, then the block is moved one square. Moving blocks is the major technique used to make a Lolo puzzle. Consider the world shown in Figure 1. To reach some of the hearts on the left of the picture Lolo must slide one block to the left, then another block to the top, capture a heart, move another block upwards, and so on. Only then can all the hearts be accessed.

We will develop the Lolo game in a series of small steps. Each step will bring out a different aspect of Object-Oriented programming.

Step 1: Getting Started – Building a Universe

As we emphasize in the textbook, an object-oriented program can often be envisioned as a universe of interacting agents. Each object in this universe has certain behavior, and certain responsibility. Most object-oriented programs begin by creating an initial set of objects, and setting them in motion.

Figure 2 is our first version if a simple Lolo game. Like all Java programs, execution begins at the static procedure named main. A static procedure, you will recall, is one that exists "outside" of any instances, and hence will exist even before an instance of the class is created. In this case our main program simply creates one instance of the class LoloGame and displays it.

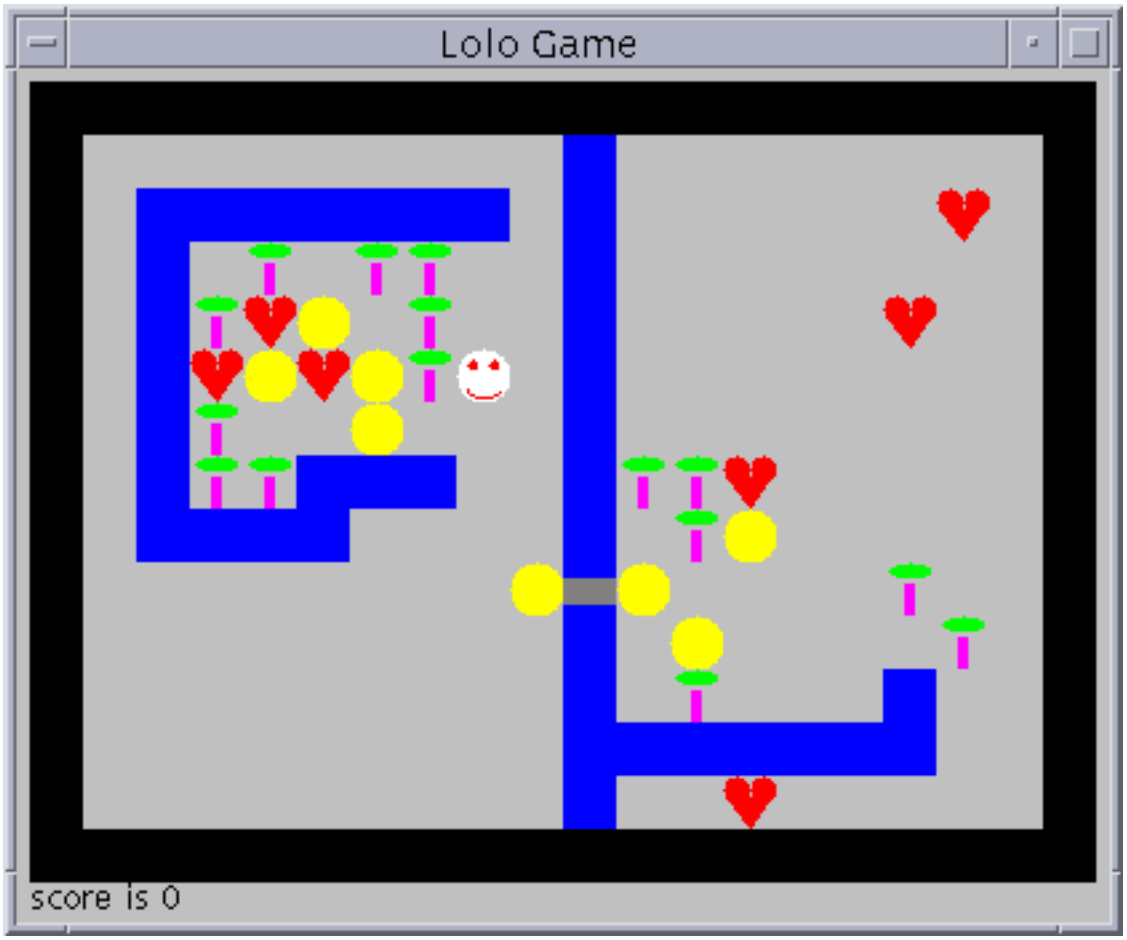

Figure 1: Screen Shot of Lolo World

```
import java.awt.*;

class LoloGame extends Frame {
    public static void main (String [ ] args) {
        LoloGame world = new LoloGame();
        world.show();
    }

    public LoloGame () {
            // awt initialization
        setTitle("Lolo Game");
        setSize(20 + BoardWidth * Cell.CellWidth,
            40 + BoardHeight * Cell.CellHeight);
            // game specific initialization
        for (int i = 0; i < BoardWidth; i++)
            for (int j = 0; j < BoardHeight; j++)
                board[i][j] = new Cell();
        board[loloX][loloY] = lolo;
    }

    private static final int BoardWidth = 20;
    private static final int BoardHeight = 15;
    protected static Cell [ ] [ ] board =
            new Cell[BoardWidth][BoardHeight];
    protected int score = 0;
    private int loloX = 10;
    private int loloY = 10;
    private Lolo lolo = new Lolo();

    public void paint (Graphics g) {
        for (int i = 0; i < BoardWidth; i++) {
            for (int j = 0; j < BoardHeight; j++) {
                board[i][j].paint(g,
                    10 + Cell.CellWidth * i,
                    30 + Cell.CellHeight * j);
            }
        }
        g.drawString("score is " + score,
            10, 40 + BoardHeight * Cell.CellHeight);
    }
}
```

Figure 2: Our Initial Lolo Game

```
import java.awt.*;

class Cell {
        // paint a cell starting at x and y
    public void paint (Graphics g, int x, int y) {
        g.setColor(Color.lightGray);
        g.fillRect(x, y, CellWidth, CellHeight);
    }

    public final static int CellWidth = 10;
    public final static int CellHeight = 10;
}

class Lolo extends Cell {
    public void paint (Graphics g, int x, int y) {
        g.setColor(Color.white);
        g.fillRect(x, y, CellWidth, CellHeight);
    }
}
```

Figure 3: The class Cell

The class LoloGame is declared as an instance of the Java library class Frame. This is the way that Java programs can create a graphical window. Part of the initialization of the class (performed in the constructor) will set the title of this window, and set its size.

There are a number of data fields declared as part of this class. Those labelled static are like static functions, they exist outside of any instance. Those labeled final cannot be changed once they are assigned. A value that is both static and final is commonly used to create a symbolic constant. The data fields BoardWidth and BoardHeight, which establishes the dimension of the Lolo world, are examples of this.

The playing board will be a two-dimenisional array of cells. The class Cell is shown in Figure 3. At least initially the only behavior for a cell will be to draw itself.

The Java AWT library draws the contents of a window by invoking the method named paint. The argument passed to this method is an instance of the class Graphics, which provides a number of useful graphical operations. The paint method in class LoloGame simply loops over each of the cells, asking them to paint themselves. A text message giving the current score is then drawn at the bottom of the screen. Note that the AWT, like most computer windowing systems, places the origin (0,0) at the upper right corner, and values increase as they move DOWN the window.

The class Cell (Figure 3) defines a pair of symbolic constants that represent the height and width of each cell in the Lolo world. The only responsibility given to an instance of the class is to paint its image on a window. The paint method is given both a Graphics object and an x and y coordinate pair that represents the upper left corner of the cell. At least

initially we will simply use a solid block of color to represent the the various different forms. The method fillRect in class Graphics allows one to draw a solid rectangle of color. A cell is drawn as a lightGray color (note the use of another symbolic constant, this one defined in the Java library class Color) and the Lolo piece is a white block.

The class Lolo is formed using inheritance from class Cell, in just the same way that the class LoloGame was extended from Frame. We will have more to say about inheritance and its uses in Step 3.

You should be able to take the classes shown in Figures 2 and 3, compile them and execute. However, the resulting system is not very interesting, since we do not yet have any game specific behavior, nor any way to allow the user to interact with the game.

While not absolutely necessary, the reader may wish to investigate how we can use the Java Graphics library to make a better representation for our game pieces. This topic is explored in Appendix A.

Step 2: Event-Driven Execution

Most graphical interfaces use a style of execution that is termed *event-driven* control flow. An event-driven program responds to user-generated actions. Users can generate actions by means of the mouse, the keyboard, or other activities (an example of another activity might be inserting a disk into a disk drive; however our programs will not explore this feature).

In Java events are handled using a technique called a *listener*. A listener is an object with sole responsibility to sit and wait, to "listen" as it were, for a specific event to occur. In this sense a listener is like a sentry in a castle guard-tower. Each listener looks for just a single type of event. When the listener senses that the event it is waiting for has occurred, it executes a method to respond. Typically this method will then interact with the main program.

We will create two listeners for our Lolo game. One will listen for window events, and the second will listen for keys being pressed. You may have noticed (depending upon your platform) that the Lolo program we created in Step 1 was difficult to halt. On Unix systems you needed to type control-C to halt the program. On other platforms you needed another platform-specific set of commands. Our first listener simply senses a click in the window close box, and when it occurs it halts the program. It can be written as follows:

```
import java.awt.event.*;

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

The class WindowAdapter is found in the Java library in the awt.event subdirectory, which is why we need a special import statement. The method windowClosing will be executed when

the user clicks in the close box. In this case, we respond by exiting the program, using the method System.exit (another part of the Java standard library).

To attach this to our program, we simply create an instance of this class, and register it with the system. This all happens in the constructor for LoloGame:

```
    public LoloGame () {
        ...
            // make listeners
        addWindowListener(new CloseQuit());
        addKeyListener(new KeyReader());
    }
```

Our second listener is slightly more complicated, because we want it to interact with other elements of the game. The Java library provides a class called KeyAdapter that will listen for key-press events. Just as with the window listener, we can add actions specific to our game by making a subclass of KeyAdapter:

```
class LoloGame extends Frame {
    ...

    class KeyReader extends KeyAdapter {
        public void keyPressed(KeyEvent e) {
            char key = e.getKeyChar();
            ... // do game specific actions
        }
    }
}
```

By placing this class *inside* the LoloGame class we create what is termed an *inner class*. The advantage of an inner class is that it is permitted to perform actions and access data values from the surrounding class. In this case the data value we want is the variable lolo.

We will use a set of four letters to move Lolo either upwards, downwards, left or right. The letter q will be an alternative way to halt the program. To move lolo we simply replace the square currently being occupied with a new empty cell, then replace the board position of the new square with the lolo piece:

```
    public void keyPressed(KeyEvent e) {
        char key = e.getKeyChar();
        board[loloX][loloY] = new Cell();
        switch (key) {
            case 'j': // move left
                loloX = loloX - 1;
                break;
            case 'k': // move right
```

```
                loloX = loloX + 1;
                break;
            case 'i': // move upwards
                loloY = loloY - 1;
                break;
            case 'm': // move downwards
                loloY = loloY + 1;
                break;
            case 'q': System.exit(0); // stop game
        }
        board[loloX][loloY] = lolo;
        repaint();
    }
```

After the piece has been moved, a call is made on the method repaint. This will schedule the window for a refresh operation. Since the value of the board has changed, the refresh will draw the new board.

Try adding this code to your lolo game. Try entering keypresses, and note how the Lolo piece moves around the window. On many platforms there will be an annoying flicker each time the window is redrawn. In Appendix B we suggest one way that this can be eliminated. What happens if you try to move Lolo off the edge of the playing board?

Step 3: Creating Useful Objects via Inheritance

We have actually seen already how we are going to create a large collection of useful objects. In Figure 3 we created the class Lolo using inheritance from the parent class Cell. By using inheritance, we are asserting that the Lolo piece is a type of cell, and in particular this means that it can be stored in the array board, which we declared as maintaining a collection of Cell values.

In this section we continue this theme, creating new classes for each of the different types of objects in our world. Classes will represent water, trees, walls, and so on. (You may already in Appendix A have explored the graphical operations necessary to draw the images for these objects).

These objects will be distinguished not only by their graphical representation, but by their behavior. In large part we can encapsulate this behavior into a single method, which will both answer the question "can Lolo move into this square?", and perform the actions necessary when Lolo does move into a square. We will call this method loloCanMove and add it to the definition of class Cell:

```
class Cell {
        ...
    public boolean loloCanMove (int direction, int i, int j) {
        return true;
```

```
    }
}
```

The arguments passed to this method will be the direction that Lolo is moving, and the i and j board coordinates for the cell. (Note that these are different from the graphics coordinates).

The behavior of the various different categories of Cell can be defined by the following table:

category	*image*	*can lolo move?*
EmptyCell	gray	yes, always
Wall	black square	no, never
Tree	green	no, never
Water	blue	no, never
Heart	red	yes, and score is increased
Stone	brown	yes if stone can slide, otherwise no
Bridge	brown	yes if approached from right direction

A typical example is the class Wall, which is defined as follows:

```
class Wall extends Cell {
    public void paint (Graphics g, int x, int y) {
        g.setColor(Color.black);
        g.fillRect(x, y, CellWidth, CellHeight);
    }

    public boolean loloCanMove (int direction, int i, int j) {
        return false;
    }
}
```

Ignoring for the moment the more complex behavior of stones and hearts, define now the classes for all these categories of objects.

In order to make use of the loloCanMove behavior, we need to change the key press listener to test a square before it moves into it. Since the same action is repeated four times with only minor variations, we can abstract it into a function:

```
private class KeyReader extends KeyAdapter {
    private boolean test (int direction, int newx, int newy) {
        return board[newx][newy].loloCanMove(direction, newx, newy);
    }

    public void keyPressed(KeyEvent e) {
        char key = e.getKeyChar();
```

```
        board[loloX][loloY] = new Cell();
        switch (key) {
            case 'j':
                if (test(Cell.left, loloX-1, loloY)
                    loloX = loloX - 1;
                break;
            case 'k':
                ...
        }
    }
}
```

Make this change, then revise the initialization of the game board to place a wall around the edge of the playing surface. You might try placing other items in various places as well. Note how this now solves the problem of Lolo running off the edge of the playing area.

An important feature to note is that we have declared the board to be class Cell, but in fact it is holding values that are created as instances of subclasses of Cell. This capability is termed *polymorphism*. When we pass the message loloCanMove to a board piece, the method executed will be determined by the current value in the cell (for example, an instance of Water or Heart), and not by the declared class of the board. This seemingly simple idea turns out to be extremely powerful.

For example, you might now go back and reconsider the operations of the listener objects. This technique only works because our listeners will override a method that is defined in a parent class found in the Java standard library. This is yet again an example of polymorphism.

Step 4: Adding the Game Logic

Much of the game logic now consists in changing the behavior of the method loloCanMove in each of the various subclasses. We will describe the various modifications needed, and leave the implementation to the reader:

Bridge Actually it is easier to split this into two classes, Horizontal bridge and vertical bridge. This makes both the printing easier and the game logic. This method uses the direction field to decide whether or not it is legal for Lolo go pass. If Lolo approaches from the right direction, then the method will return true, otherwise it will return false.

Heart This method will add 1 to the static field LoloGame.score before returning true.

Stone This method uses both the x and y coordinate and the direction. It tests whether or not the adjacent square (where adjacent is determined by the direction) is empty. If so, then it moves the stone to the adjacent square, and returns true. Otherwise, it returns false.

This is sufficient to create a workable game. In the next section we will describe various features that can be used to improve the game.

Step 5: More Objects

There are other objects that can be envisioned for the Lolo game. One example might be giant Balls. Like a square, Lolo can push on a ball. Unlike a square, a ball can roll as long as there are empty squares. Perhaps a sequence of balls can be rolled together. Perhaps when a ball strikes a stone it shatters both the ball and stone, leaving only an empty square.

The blocks as we have designed them can only be pushed into empty squares. Another form of block could be created that could be pushed into water, and when it hits water it becomes a bridge (or an empty square). In this way Lolo can create a bridge where there is none already existing.

Use your imagination to create other types of objects.

Step 6: Improvements

While after step 4 we have a working game, there are various improvements that can be made.

The Java system makes it very easy to incorporate and play sound clips. If you can make or find such clips you can add sounds for the various Lolo activities – moving into a new square, pushing a block, crossing a bridge, finding a heart, and so on.

Next, there is the technique used to create the initial game board. In the description in Step 2 we hard-wired the initial positions of the pieces into the program logic. A more flexible alternative that allows the board description to be easily changed is to read a textual description of a board. We can represent each piece by a character, for example A for wall, W for wall, T for tree, R for rock, H for vertical bridge and = for horizontal bridge, and so on. Using this encoding the game shown in Figure 1 can be described by a set of strings, as in the following:

```
WWWWWWWWWWWWWWWWWWWW
WEEEEEEEEEAEEEEEEEEW
WEAAAAAAAEAEEEEEEHEW
WEAETETTEEAEEEEEEEEW
WEATEBETEEAEEEEEHEEW
WEAHEEBTLEAEEEEEEEEW
WEATEEBEEEAEEEEEEEEW
WEATTAAAEEAETHEEEEEW
WEAAAAEEEEAETGEEEEEW
WEEEEEEEEB=BEEEETEEW
WEEEEEEEEEAEBEEEETEW
WEEEEEEEEEAETEEEAEEW
WEEEEEEEEEAAAAAAAEEW
```

```
WEEEEEEEEEAEEHEEEEEW
WWWWWWWWWWWWWWWWWWW
```

An array of such values can be initialized in Java in a single statement, as in the following:

```
String [ ] initialBoard = {
"WWWWWWWWWWWWWWWWWWWW",
"WEEEEEEEEEAEEEEEEEEW", ...
"WWWWWWWWWWWWWWWWWWWW"};
```

A method can then be written that will use such an array and reading the values initialize the board array, creating the different types of values as necessary.

The next level of complexity would be to store the initial values in a file, and read the array of strings one line at a time from the file.

In the original Lolo game there are levels of play of increasing difficulty. We can incorporate this feature by having a Stairwell object. The stairwell object returns false in response to the loloCanMove command as long as there are hearts that remain on the playing board. Once all the hearts have been gathered the stairwell permits Lolo to move onto the square, and in response loads the next level of play.

The text area at the bottom of the window can be used for more than simple scoring. You could display messages here when Lolo tries to perform an illegal action, for example running into a tree.

Once you have multiple levels of play it becomes annoying to always have to restart at the beginning. The bit of complexity asks the user for a name, and when play ends it will save the current name and level in a file. When the user begins at a later point execution can commence at the largest completed level.

Note that when Lolo leaves a square it is changed to an empty square. Most of the time this is correct. The one place this seems odd is when Lolo crosses a bridge – which somehow suddenly becomes solid ground. This can be fixed by having the Lolo piece remember the type of square it is moving onto, and then restoring the cell once the lolo piece moves off.

If Lolo is given a face (see Appendix A) an interesting effect is to have the eyes change as the direction of movement changes.

A Giving Lolo a face

Simple square blocks of color do not make very interesting game pieces. Although the Graphics class provides only a few primitive operations, even these can be used in a variety of interesting ways.

The method fillOval takes the same arguments as fillRect, and fills an oval that fits into the given rectangle. We can use this to draw the face of Lolo as a white circle, with two red eyes:

```
class Lolo extends Cell {
```

```
    public void paint (Graphics g, int x, int y) {
        g.setColor(Color.white);
        g.fillOval(x, y, CellWidth, CellHeight); // face
        g.setColor(Color.red);
        g.fillOval(x+4, y+4, 4, 4); // left eye
        g.fillOval(x+12, y+4, 4, 4); // right eye
        g.drawArc(x+4, y+12, 12, 6, 190, 160); // smile
    }
}
```

To make the "mouth" we have used the method drawArc, which defines a bounding box by the first four arguments, (as with fillRect and fillOval, the bound is defined by the upper left corner and a width and height). The next argument is the starting angle for the arc, and the final argument is the number of degrees to draw.

Other images can be drawn in a similar fashion. A tree can be drawn as a green oval over a brown rectangle. A bridge is drawn using three rectangles, two for the water and one for the bridge.

A Heart can be drawn using two circles and a triangle. The latter can be formed using a more general method called fillPoly, which takes an array of x values and an array of y values, and fills the polygon between the points:

```
class Heart extends Cell {
    public void paint (Graphics g, int x, int y) {
        g.setColor(Color.red);
            // assumes CellWidth and height are 20
        g.fillOval(x, y, 10, 10); // left circle
        g.fillOval(x+10, y, 10, 10); // right circle
        int [ ] xpts = new int[3]; // make triangle
        int [ ] ypts = new int[3]; // for body
        xpts[0] = x+1;
        ypts[0] = y+8;
        xpts[1] = x+10;
        ypts[1] = y+20;
        xpts[2] = x+18;
        ypts[2] = y+8;
        g.fillPolygon(xpts, ypts, 3); // draw triangle
    }
}
```

More advanced users may wish to investigate the ability to load and print gif or jpeg images, which allows for a much wider range of possibilities for the pieces.

B Removing The Flicker by Double Buffering

Many graphical applications will have an annoying flicker because the window is first repainted in the background color, and then repainted according the user specifications. The first repainting can be eliminated by overriding the update method as well as the paint method. Simply rename the paint method as update, then add the following as a new paint method:

```
public void paint (Graphics g) {
    update(g);
}

public void update (Graphics g) {
    // user defined painting routines
}
```