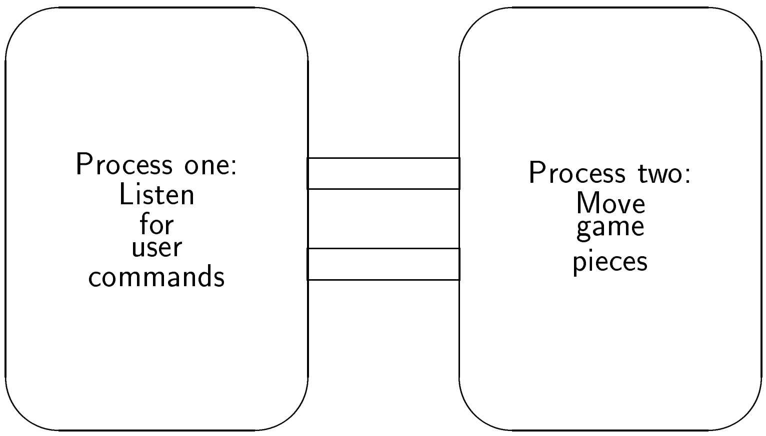
Chapter 6

Case Study: A Tetris Game

As we saw in the cannon game chapter (Chapter 4), an arcade game is a good illustration of the need for multiple threads of control. In a typical arcade game, the user is constantly entering information, and the game itself is constantly moving. In a conventional single threaded programming language, one or the other of these tasks must be selected as the primary structure. Either the main program is a large loop that listens for the users commands, and every now and then moves the game forward, or it is a large loop that moves the elements of the game, and every now and then listens for the users instructions. Either form will end up being clumsy and difficult to debug.

A much more elegant solution is provided by Java. Each of these two tasks can be assigned its own process, or *Thread* of control. Each thread is executed in parallel with every other thread. On thread is simply listening for the users commands, while the other thread is charged with moving the pieces.



Of course, there must be some sort of communication between the two threads, and this is where most of the complexity in multithreaded programming occurs (see Chapter *thread*.) In the Tetris game we will present here, the communication consists only of two integer values; the command to be performed and the current score.

In addition to demonstrating multiple threads of control, the Tetris game presented in this chapter will illustrate how to read keyboard events, and an important technique for preventing flickering in graphical displays, called *double buffering*.

6.1 The Tetris Game Class

The primary class for the Tetris game is shown in Figure 6.1, with the body of a number of member functions elided. As with all applications, execution begins with a static procedure named main. The main procedure creates an instance of the class Tetris, which is a type of Frame, and places this window on the display using the method show.

The constructor for the Tetris class resizes the main window, gives it a name, then creates and starts the second process. The game playing process will be an instance of a second class, named PieceMover. We will examine the structure of this class shortly. Once created, the thread of execution that will play the game is initiated by the issuing of the command start.

6.1.1 Communication Between Threads

Communication between the two threads of control is accomplished via two integer variables; one holding the current command to be executed, and one holding the current score. These variables are both set and accessed via a pair of procedures, as follows:

```
class Tetris extends Frame {
    ...

    public synchronized int getCommand (int nextCommand)
    {
        int oldCommand = currentCommand;
        currentCommand = nextCommand;
        return oldCommand;
        }

    public synchronized int addScore (int amount)
    {
        score = score + amount;
        return score;
    }
```

```
class Tetris extends Frame {

        // application starting point
    public static void main (String [] args)
    {
        Tetris world = new Tetris();
        world.show();
    }

        // data fields
    public static final int FrameWidth = 300;
    public static final int FrameHeight = 400;

    private Image buffer;
    private PieceMover player;
    private int score = 0;
    private int currentCommand = Piece.Down;

        // constructor
    public Tetris ()
    {       // set game attributes
        resize (FrameWidth, FrameHeight);
        setTitle("Tetris");
            // create player thread
        player = new PieceMover(this);
        player.start();
    }

        // interface point between two threads

    public synchronized int getCommand (int nextCommand) { ... }

    public synchronized int addScore (int amount) { ... }

    public boolean keyDown (Event e, int key) { ... }

    public void update (Graphics g) { ... }

    public void paint (Graphics g) { ... }
}
```

Figure 6.1: The Controller class for the Tetris Game

```
    ...
}
```

The procedures are declared as synchronized. This means that only one of the two threads can be executing one of these procedures at any one time. If the other process attempts to execute one of the procedures, it will be held up until the first process completes. (In truth, the synchronization here is largely unnecessary, since the only penalty for simultaneous access will be that one process may received an out of date value. In Chapter *thread* we will examine situations where synchronization is much more critical. Nevertheless, the use of the synchronized designation also helps to document the interface between the two routines).

Note that these two routines are similar in use, but slightly different. The addScore routine adds a value to the current score, and returns the updated result. The getCommand routine updates a command buffer to a new command, but returns the old command. The controlling program will be placing commands into the buffer, while the piece mover will be removing them, both using this same routine. The potential exists that the controller can place a second command into the buffer before the player has had a chance to remove the first. While this potential is there, the relative speeds of humans versus computers makes this unlikely, and the immediate feedback in the graphical interface mitagates this problem.

6.1.2 Handing Keyboard Events

Reading keyboard events is somewhat more tricky than most other events (see Chapter *events*), as the details are somewhat platform specific. On most platforms there is some notion of the current "focus", and keyboard events are sent only to that focus. This differs from, say, mouse events, where the location of the cursor indicates the window most likely interested in the event. The window that is the focus for keyboard events need not be the window currently under the mouse. How one goes about setting the focus for the keyboard differs from one platform to another. Often clicking the mouse on a window associated with a process will implicitly set the focus to be that process.

Once the focus has been set, keyboard events are most commonly handled by a procedure named keyDown, which takes as argument an event record, and the character representing the key pressed. Like all event handing routines, this procedure returns a boolean value which should be set to true if the event has been properly handled, and false if the key should be passed on to some other procedure for handling. In our Tetris game, we will use the keyboard letters j, k and l for the key actions of move left, rotate, and move right. The letter g will start a new game, while the letter q will halt the program. All other letters will be ignored.

```
class Tetris extends Frame {
    ...
```

```
    public boolean keyDown (Event e, int key)
    {
        switch ((char) key) {
            case 'g': player.newGame (); break;
            case 'j': getCommand (Piece.Left); break;
            case 'k': getCommand (Piece.Rotate); break;
            case 'l': getCommand (Piece.Right); break;
            case 'q': System.exit (0);
            default: return false;
        }
        return true;
    }
    ...
}
```

6.1.3 Double Buffering

As the game is being played, the screen is constantly being updated. This update process consists largely of drawing each individual square of the display. If this is done as separate graphical commands, the result can be an annoying flicker. This flicker can be avoided by drawing not onto the screen canvas, but first drawing onto a temporary, behind the scenes canvas. This secondary canvas is then, in one operation, copied into the window canvas, reducing the large number of individual operations to a single operation. This technique is termed *double buffering*.

To perform double buffering, the programmer overrrides not only the paint method, but the update method as well. The update method creates a secondary buffer, if one has not already been created. As this buffer can be used over and over again, it is useful to store this in a class data field, and only create the buffer the first time the procedure is called.

```
class Tetris extends Frame {
    ...
    public void update (Graphics g)
    {
        if (buffer == null)
            buffer = createImage(FrameWidth, FrameHeight);
        Graphics secondBuffer = buffer.getGraphics();
        player.paint (secondBuffer);
        g.drawImage (buffer, 0, 0, this);
    }

    public void paint (Graphics g)
    {
```

```
        player.paint(g);
        String scoreString = "Score " + addScore(0);
        g.setColor(Color.blue);
        g.drawString(scoreString, 200, 200);
    }
}
```

The graphics context for the secondary buffer is then generated, and the paint method invoked using the graphics context for the buffer, instead of the graphics context for the current window.

The graphics operations are exactly the same as if double buffering was not being used. In this case, the piece mover class is used to paint display the location of all current pieces, and the current score is printed to the left of the playing area. Once the painting is finished, the entire buffer image is then drawn on to the window graphics context, overwriting the original values.

One often confusing feature to note is that although the paint procedure invokes the paint procedure in the class of the player object (namely, the class PieceMover) the resulting actions are not performed by the piece mover thread. Updating the image is performed by the first thread, and is independent of the piece mover thread, which continues to run on its own.

6.2 The PieceMover Thread

The class PieceMover is shown in Figure 6.2. The fact that the class extends class Thread, and the presence of the method named run, indicate that this class can potentially be executed as an independent thread. (This is one of two ways that a class can be made into a thread, in the cannon ball game we illustrated the alternative, making a class that implements the Runnable interface. See Chapter *thread* for more details.)

The constructor for the class will be executed when the instance of PieceMover is created, as in Figure 6.1. The constructor sets the values of local variables, including creating an array of colors for the playing area. The playing area will be 15 elements wide by 30 elements deep. Because we think of the playing area as a graphical object, it will be easier to describe this by placing the horizontal movement first and the vertical movement second.

The thread starting point is a procedure named run. This method will be executed when the thread is started. Note that the method the programmer writes is called run, while the method used to initiate the thread is called start (the call on start in this case appears in Figure 6.1). The start method performs some other actions, such as setting up a schedule for thread execution, before invoking run. In our simple example, the run method will simply repeatedly create and drop playing pieces.

As the playing pieces move, they will indicate the change by altering the colors stored in the color table. Thus, the paint method simply cycles through the table, and displays each

```
class PieceMover extends Thread {

        // private data fields
    private Tetris controller;
    private Color table[][];
    private Piece currentPiece;

        // constructor
    public PieceMover (Tetris t)
    {
        controller = t;
        table = new Color[15][30];
        currentPiece = null;
        newGame();
    }

        // thread starting point
    public void run ()
    {
        while (dropPiece()) { }
    }

        // other methods
    public void newGame ()
    {
        for (int i = 0; i  < 30; i++)
            for (int j = 0; j < 15; j++)
                table[j][i] = Color.white;
    }

    public void paint (Graphics g) { ... }

    public boolean dropPiece () { ... }

    private void moveDown (int start) { ... }

    private void checkScore()  { ... }
}
```

Figure 6.2: The PieceMover Class

color in a small rectangle.

```
class PieceMover extends Thread {
    ...
    public void paint (Graphics g)
    {
        for (int i = 0; i < 30; i++)
            for (int j = 0; j < 15; j++) {
                g.setColor(table[j][i]);
                g.fillRect(20+10*j, 350-10*i, 10, 10);
                }
    }
}
```

The heart of the piece movement thread is the function dropPiece. This procedure creates a new piece, selecting the type of piece randomly from among the seven possibilities. If the piece cannot be moved downward once initially created at the top of the scene, the game is over, and the procedure returns false. Otherwise, a loop then reads the commands placed in the buffer shared with the controller, and acts on them, passing the command value to the individual piece for action. After every move the controller is asked to repaint, and control of the processor is yielded, allowing the other process time to execute. When control is returned, the piece movement process still sleeps for 100 microseconds, before reading the next command and moving once more.

```
class PieceMover extends Thread {
    ...
    private boolean dropPiece ()
    {
        int piecetype = 1 + (int) (7 * Math.random ());
        currentPiece = new Piece (piecetype);
        if (! currentPiece.move (Piece.Down, table))
            return false;
        int command = controller.getCommand (Piece.Down);
        while (currentPiece.move (command, table)) {
            controller.repaint ();
            yield ();
            try {
                sleep (100);
                } catch (InterruptedException e) { }
            command = controller.getCommand (Piece.Down);
            }
        // piece cannot move, check score
```

```
        checkScore();
        return true;
    }
}
```

The while loop terminates when the piece can no longer move downward. At this point the procedure checkScore is called to determine if any points have been scored. Points are scored by completely filling a row. If such a row is found, the controller is notified that a score has been made, and the entire image is moved downward.

```
class PieceMover extends Thread {
    ...
    private void checkScore()
    {
        for (int i = 0; i < 30; i++) {
            boolean scored = true;
            for (int j = 0; j < 15; j++)
                if (table[j][i] == Color.white)
                    scored = false;
            if (scored) {
                controller.addScore(10);
                moveDown(i);
                i = i - 1; // check row again
                }
            }
    }

    private void moveDown (int start)
    {
        for (int i = start; i < 30; i++)
            for (int j = 0; j < 15; j++)
                if (i < 29)
                    table[j][i] = table[j][i+1];
                else
                    table[j][i] = Color.white;
    }
}
```

The procedure moveDown copies the colors from the row above into the current row, copying white values into the topmost row.

6.3 The Game Piece class

An individual game piece is composed of a number of blocks. Each block is identified only by an x and y coordinate, maintained in a pair of parallel arrays. A piece also has its own color. The constructor for the class Piece (Figure 6.3) takes an integer that represents the piece type, and initializes the locations approrpriately. Only the initialization of the "S" shaped piece is shown, the others are all similar.

Other than the constructor, the only publicly accessible procedure for the game piece is the function named move. This function tries to move the piece according to the command given. The function returns false if the command is a move down and the piece cannot move, which indicates that no further actions are possible.

```
class Piece {
    ...
    public boolean move (int command, Color [][] table)
    {
        erase(table);
        boolean canDoIt = false;
        switch (command) {
            case Down: canDoIt = testDown(table); break;
            case Right: canDoIt = testMoveRight(table); break;
            case Left: canDoIt = testMoveLeft(table); break;
            case Rotate: canDoIt = testRotate(table); break;
            }
        if (canDoIt)
            switch (command) {
                case Down: moveDown(); break;
                case Right: moveRight(); break;
                case Left: moveLeft(); break;
                case Rotate: moveRotate(); break;
                }
        draw(table);
        if (command == Down)
            return canDoIt;
        return true;
    }
}
```

Erasing and redrawing the piece are performed by transferring colors to the color table:

```
class Piece {
    ...
```

```
class Piece {
        // data fields
    private int x[];
    private int y[];
    private Color color;
        // moves
    public static final int Down = 1;
    public static final int Left = 2;
    public static final int Rotate = 3;
    public static final int Right = 4;

        // constructor
    public Piece (int type)
    {
        switch(type) {
            case 1:    // s shaped piece
                x = new int[4];
                y = new int[4];
                x[0] = 7;
                y[0] = 29;
                x[1] = 7;
                y[1] = 28;
                x[2] = 8;
                y[2] = 28;
                x[3] = 8;
                y[3] = 27;
                color = Color.green;
                break;
            ...
            }
    }
    ...
}
```

Figure 6.3: The class Piece

```
    private void erase (Color [][] table)
    {
        for (int i = 0; i < x.length; i++)
            table[x[i]][y[i]] = Color.white;
    }

    private void draw (Color [][] table)
    {
        for (int i = 0; i < x.length; i++)
            table[x[i]][y[i]] = color;
    }
}
```

Before any command is performed, a test is conducted to see if the spaces the blocks will occupy subsequent to the command are open. The following demonstrates one of these routines, the others are similar. If the target locations are open, the corresponding move routine performs the actual transformation.

```
class Piece {
    ...
    private boolean testPosition (int x, int y, Color[][] table)
    {
        if ((x < 0) || (x > 14))
            return false;
        if ((y < 0) || (y > 29))
            return false;
        if (table[x][y] != Color.white)
            return false;
        return true;
    }

    private boolean testDown (Color [][] table)
    {
        for (int i = 0; i < x.length; i++)
            if (! testPosition(x[i], y[i]-1, table))
                return false;
        return true;
    }

    private void moveDown ()
    {
        for (int i = 0; i < x.length; i++)
```

```
            y[i] = y[i] - 1;
    }
}
```

The only tricky procedure is the rotation. One square, the square at position 1, is designated the square around which rotation will be performed. The differences in coordinate from this square are computed, and the updated coordinates determined.

```
class Piece {
    ...
    private boolean testRotate (Color [][] table)
    {
        for (int i = 0; i < x.length; i++) {
            int dx = x[i] - x[1];
            int dy = y[i] - y[1];
            int nx = x[1] + dy;
            int ny = y[1] - dx;
            if (! testPosition(nx, ny, table))
                return false;
            }
        return true;
    }
}
```

Exercises

1. The game can be made to "speed up" by reducing the amount of time the piece mover sleeps between commands. Change the code so that this time is reduced by 5 milliseconds for every 100 points the user scores.

2. Change the program so that instead of scoring 10 points for each filled row, the player gets additional points for multiple rows scored with the name piece. For example, a second row will get 20 points, a third row 30 points, and a fourth row 40 points.

3. Change the program to score 50 extra points if an entirely filled row is all one color.