J2ME Games with MIDP 2

CAROL HAMER

Apress™

J2ME Games with MIDP 2
Copyright ©2004 by Carol Hamer

ISBN (pbk): 1-59059-382-0

Printed and bound in the United States of America 10987654321

Lead Editor: Steve Anglin

Technical Reviewer: William Swaney

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Kylie Johnston

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Katie Stence

Compositor: Kinetic Publishing Services, LLC

Proofreader: Elizabeth Berry

Indexers: Ron Strauss and Ann Rogers

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 3

Using the MIDP 2.0 Games API

Now it's time to look at the most important package you'll be dealing with when writing games with Mobile Internet Device Profile (MIDP) 2.0: the package `javax.microedition.lcdui.game.*`. In this chapter, I show you the main parts of a MIDP 2.0 game by explaining the code of an example game called Tumbleweed. The game involves a cowboy walking through a prairie jumping over tumbleweeds. It's kind of a silly game, but it illustrates most of the basics you'll need when writing more reasonable games.

As in the earlier chapters, I've included all of the code necessary to build the example, and you can download the code from the Downloads section of the Apress Web site (`http://www.apress.com`) with all its resources.

Starting with the MIDlet Class

As usual, the application starts with the `MIDlet` class. In this case, my `MIDlet` subclass is called `Jump`. This class is essentially the same as the `MIDlet` subclass from the previous chapter, so if you'd like a detailed explanation of what's going on in it, please see the "Using the MIDlet Class" section in Chapter 2. The only differences here are the use of a separate `GameThread` class and the fact that when the user presses a command button, I have the `MIDlet` change the command that's available on the screen. The command change is because the user can pause the game only when it's unpaused, can unpause the game only when it's paused, and can start over only when the game has ended.

Listing 3-1 shows the game's `MIDlet` subclass called `Jump.java`.

Listing 3-1. `Jump.java`

```
package net.frog_parrot.jump;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 * This is the main class of the Tumbleweed game.
 *
 * @author Carol Hamer
 */
public class Jump extends MIDlet implements CommandListener {

  //----------------------------------------------------------
  //   commands

  /**
   * the command to end the game.
   */
  private Command myExitCommand = new Command("Exit", Command.EXIT, 99);

  /**
   * the command to start moving when the game is paused.
   */
  private Command myGoCommand = new Command("Go", Command.SCREEN, 1);

  /**
   * the command to pause the game.
   */
  private Command myPauseCommand = new Command("Pause", Command.SCREEN, 1);

  /**
   * the command to start a new game.
   */
  private Command myNewCommand = new Command("Play Again", Command.SCREEN, 1);

  //----------------------------------------------------------
  //   game object fields

  /**
   * the canvas that all of the game will be drawn on.
   */
  private JumpCanvas myCanvas;

  /**
   * the thread that advances the cowboy.
   */
  private GameThread myGameThread;

  //------------------------------------------------------
  //    initialization and game state changes
```

```
/**
 * Initialize the canvas and the commands.
 */
public Jump() {
  try {
    myCanvas = new JumpCanvas(this);
    myCanvas.addCommand(myExitCommand);
    myCanvas.addCommand(myPauseCommand);
    myCanvas.setCommandListener(this);
  } catch(Exception e) {
    errorMsg(e);
  }
}

/**
 * Switch the command to the play again command.
 */
void setNewCommand () {
  myCanvas.removeCommand(myPauseCommand);
  myCanvas.removeCommand(myGoCommand);
  myCanvas.addCommand(myNewCommand);
}

/**
 * Switch the command to the go command.
 */
private void setGoCommand() {
  myCanvas.removeCommand(myPauseCommand);
  myCanvas.removeCommand(myNewCommand);
  myCanvas.addCommand(myGoCommand);
}

/**
 * Switch the command to the pause command.
 */
private void setPauseCommand () {
  myCanvas.removeCommand(myNewCommand);
  myCanvas.removeCommand(myGoCommand);
  myCanvas.addCommand(myPauseCommand);
}

//----------------------------------------------------------------
//  implementation of MIDlet
// these methods may be called by the application management
// software at any time, so you always check fields for null
// before calling methods on them.

/**
 * Start the application.
 */
public void startApp() throws MIDletStateChangeException {
  if(myCanvas != null) {
    if(myGameThread == null) {
      myGameThread = new GameThread(myCanvas);
      myCanvas.start();
      myGameThread.start();
    } else {
      myCanvas.removeCommand(myGoCommand);
      myCanvas.addCommand(myPauseCommand);
      myCanvas.flushKeys();
      myGameThread.resumeGame();
    }
  }
}

/**
 * stop and throw out the garbage.
 */
public void destroyApp(boolean unconditional)
    throws MIDletStateChangeException {
  if(myGameThread != null) {
    myGameThread.requestStop();
  }
  myGameThread = null;
  myCanvas = null;
  System.gc();
}

/**
 * request the thread to pause.
 */
public void pauseApp() {
  if(myCanvas != null) {
    setGoCommand();
  }
  if(myGameThread != null) {
    myGameThread.pauseGame();
  }
```

```
}

//------------------------------------------------------------
//  implementation of CommandListener

/*
 * Respond to a command issued on the Canvas.
 * (either reset or exit).
 */
public void commandAction(Command c, Displayable s) {
  if(c == myGoCommand) {
    myCanvas.removeCommand(myGoCommand);
    myCanvas.addCommand(myPauseCommand);
    myCanvas.flushKeys();
    myGameThread.resumeGame();
  } else if(c == myPauseCommand) {
    myCanvas.removeCommand(myPauseCommand);
    myCanvas.addCommand(myGoCommand);
    myGameThread.pauseGame();
  } else if(c == myNewCommand) {
    myCanvas.removeCommand(myNewCommand);
    myCanvas.addCommand(myPauseCommand);
    myCanvas.reset();
    myGameThread.resumeGame();
  } else if((c == myExitCommand) || (c == Alert.DISMISS_COMMAND)) {
    try {
      destroyApp(false);
      notifyDestroyed();
    } catch (MIDletStateChangeException ex) {
    }
  }
}

//----------------------------------------------------
//  error methods

/**
 * Converts an exception to a message and displays
 * the message..
 */
void errorMsg(Exception e) {
  if(e.getMessage() == null) {
    errorMsg(e.getClass().getName());
  } else {
      errorMsg(e.getClass().getName() + ":" + e.getMessage());
    }
  }

  /**
   * Displays an error message alert if something goes wrong.
   */
  void errorMsg(String msg) {
    Alert errorAlert = new Alert("error",
                                msg, null, AlertType.ERROR);
    errorAlert.setCommandListener(this);
    errorAlert.setTimeout(Alert.FOREVER);
    Display.getDisplay(this).setCurrent(errorAlert);
  }

}
```

Using the Thread Class

This game requires only the simplest use of the `Thread` class. Chapter 4 covers how to use threads. But even in this simple case, I'd like to mention a few points.

In this case, it really is necessary to spawn a new thread. The animation in this game is always moving, even when the user doesn't press a button, so I need to have a game loop that repeats constantly until the end of the game. I can't use the main thread for the game loop because the application management software may need to use the main thread while my game is running. While testing the game in the emulator, I found that if I use the main thread for my game's animation loop, the emulator is unable to respond to keystrokes. Of course, in general, it's good practice to spawn a new thread when you plan to go into a loop that's to be repeated throughout the duration of your program's active life cycle.

Here's how my `Thread` subclass (called `GameThread`) works: Once the thread starts, it goes into the main loop (inside the `while(true)` block). The first step is to check if the `Jump` class has called `requestStop()` since the last cycle. If so, you break out of the loop, and the `run()` method returns. Otherwise, if the user hasn't paused the game, you prompt the `GameCanvas` to respond to the user's keystrokes and advance the game animation. Then you do a short pause of one millisecond. This is partially to be sure that the freshly painted graphics stay on the screen for an instant before the next paint, but it's also useful to help the keystroke query work correctly. As mentioned, the information about the user's keystrokes is updated on another thread, so it's necessary to put a short wait inside your game loop to make sure that the other thread gets a turn and has the opportunity to update the key state's value in a timely fashion. This allows your game to respond immediately when the user presses a key. Even a millisecond will do the trick. (I earlier wrote a racecar game in which I neglected to put a wait in the main game loop, and I found that the car would go halfway around the track between the time I pressed the lane change key and the time the car actually changed lanes...).

Listing 3-2 shows the code for `GameThread.java`.

Listing 3-2. `GameThread.java`

```
package net.frog_parrot.jump;

/**
 * This class contains the loop that keeps the game running.
 *
 * @author Carol Hamer
 */
public class GameThread extends Thread {

  //---------------------------------------------------------
  //   fields

  /**
   * Whether the main thread would like this thread
   * to pause.
   */
  private boolean myShouldPause;

  /**
   * Whether the main thread would like this thread
   * to stop.
   */
  private boolean myShouldStop;

  /**
   * A handle back to the graphical components.
   */
  private JumpCanvas myJumpCanvas;

  //---------------------------------------------------------
  //   initialization

  /**
   * standard constructor.
   */
  GameThread(JumpCanvas canvas) {
    myJumpCanvas = canvas;
  }
```

```
//-------------------------------------------------------
//    actions

/**
 * pause the game.
 */
void pauseGame() {
  myShouldPause = true;
}

/**
 * restart the game after a pause.
 */
synchronized void resumeGame() {
  myShouldPause = false;
  notify();
}

/**
 * stops the game.
 */
synchronized void requestStop() {
  myShouldStop = true;
  notify();
}

/**
 * start the game..
 */
public void run() {
  // flush any keystrokes that occurred before the
  // game started:
  myJumpCanvas.flushKeys();
  myShouldStop = false;
  myShouldPause = false;
  while(true) {
    if(myShouldStop) {
      break;
    }
    synchronized(this) {
      while(myShouldPause) {
        try {
          wait();
        } catch(Exception e) {}
```

```
          }
        }
        myJumpCanvas.checkKeys();
        myJumpCanvas.advance();
        // you do a short pause to allow the other thread
        // to update the information about which keys are pressed:
        synchronized(this) {
          try {
            wait(1);
          } catch(Exception e) {}
        }
      }
    }

  }
```

Using the GameCanvas Class

Now you'll look at the class that allows you to paint customized game graphics to the screen.

How GameCanvas Differs from Canvas

The `GameCanvas` class represents the area of the screen that the device has allotted to your game. The `javax.microedition.lcdui.game.GameCanvas` class differs from its superclass `javax.microedition.lcdui.Canvas` in two important ways: graphics buffering and the ability to query key states. Both of these changes give the game developer enhanced control over precisely when the program deals with events such as keystrokes and screen repainting.

The graphics buffering allows all the graphical objects to be created behind the scenes and then flushed to the screen all at once when they're ready. This makes animation smoother. I've illustrated how to use it in the method `advance()` in Listing 3-3. (Recall that the method `advance()` is called from the main loop of my `GameThread` object.) Notice that to update and repaint the screen, all you need to do is call `paint(getGraphics())` and then call `flushGraphics()`. To make your program more efficient, there's even a version of the `flushGraphics()` method that allows you to repaint just a subset of the screen if you know that only part has changed. As an experiment I tried replacing the calls to `paint(getGraphics())` and `flushGraphics()` with calls to `repaint()` and then `serviceRepaints()` as you might if your class extended `Canvas` instead of `GameCanvas`. In my simple examples it didn't make much difference, but if your game has a lot of complicated graphics, the `GameCanvas` version will undoubtedly make a big difference.

The ability to query key states is helpful for the game's organization. When you extend the Canvas class directly, you must implement the keyPressed(int keyCode) if your game is interested in keystrokes. The application management software then calls this method when the user presses a button. But if your program is running on its own thread, this might happen at any point in your game's algorithm. If you're not careful about using synchronized blocks, this could potentially cause errors if one thread is updating data about the game's current state and the other is using that data to perform calculations. The program is simpler and easier to follow if you get the keystroke information when you want it by calling the GameCanvas method getKeyStates().

An additional advantage of the getKeyStates() method is that it can tell you if multiple keys are being pressed simultaneously. The keyCode that's passed to the keyPressed(int keyCode) method can tell you only about a single key and therefore will be called multiple times even if the user presses two keys at the same time. In a game, the precise timing of each keystroke is often important, so the Canvas method keyPressed() loses valuable information. Looking at the method checkKeys() in Listing 3-3, you can see that the value returned by getKeyStates() contains all the keystroke information. All you need to do is perform a bitwise "and" (&) between the getKeyStates() return value and a static field such as GameCanvas.LEFT_PRESSED to tell if a given key is currently being pressed.

This is a large classfile, but you can see the main idea of how it works by recalling that the main loop of the GameThread class first tells my GameCanvas subclass (called JumpCanvas) to query the key states (see the method JumpCanvas.checkKeys() in Listing 3-3 for details). Then once the key events have been dealt with, the main loop of the GameThread class calls JumpCanvas.advance(), which tells the LayerManager to make appropriate updates in the graphics (more on that in the next sections) and then paints the screen.

Listing 3-3 shows the code for JumpCanvas. java.

Listing 3-3. JumpCanvas.java

```
package net.frog_parrot.jump;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * This class is the display of the game.
 *
 * @author Carol Hamer
 */
public class JumpCanvas extends javax.microedition.lcdui.game.GameCanvas {
```

```
//---------------------------------------------------------
//    dimension fields
//   (constant after initialization)

/**
 * the height of the green region below the ground.
 */
static final int GROUND_HEIGHT = 32;

/**
 * a screen dimension.
 */
static final int CORNER_X = 0;

/**
 * a screen dimension.
 */
static final int CORNER_Y = 0;

/**
 * a screen dimension.
 */
static int DISP_WIDTH;

/**
 * a screen dimension.
 */
static int DISP_HEIGHT;

/**
 * a font dimension.
 */
static int FONT_HEIGHT;

/**
 * the default font.
 */
static Font FONT;

/**
 * a font dimension.
 */
static int SCORE_WIDTH;
```

```
/**
 * The width of the string that displays the time,
 * saved for placement of time display.
 */
static int TIME_WIDTH;

/**
 * color constant
 */
public static final int BLACK = 0;

/**
 * color constant
 */
public static final int WHITE = 0xffffff;

//---------------------------------------------------------
//   game object fields

/**
 * a handle to the display.
 */
private Display myDisplay;

/**
 * a handle to the MIDlet object (to keep track of buttons).
 */
private Jump myJump;

/**
 * the LayerManager that handles the game graphics.
 */
private JumpManager myManager;

/**
 * whether the game has ended.
 */
private boolean myGameOver;

/**
 * the player's score.
 */
private int myScore = 0;
```

```
/**
 * How many ticks you start with.
 */
private int myInitialGameTicks = 950;

/**
 * this is saved to determine if the time string needs
 * to be recomputed.
 */
private int myOldGameTicks = myInitialGameTicks;

/**
 * the number of game ticks that have passed.
 */
private int myGameTicks = myOldGameTicks;

/**
 * you save the time string to avoid recreating it
 * unnecessarily.
 */
private static String myInitialString = "1:00";

/**
 * you save the time string to avoid recreating it
 * unnecessarily.
 */
private String myTimeString = myInitialString;

//-----------------------------------------------------
//    gets/sets

/**
 * This is called when the game ends.
 */
void setGameOver() {
  myGameOver = true;
  myJump.pauseApp();
}

//-----------------------------------------------------
//    initialization and game state changes

/**
 * Constructor sets the data, performs dimension calculations,
   * and creates the graphical objects.
   */
  public JumpCanvas(Jump midlet) throws Exception {
    super(false);
    myDisplay = Display.getDisplay(midlet);
    myJump = midlet;
    // calculate the dimensions
    DISP_WIDTH = getWidth();
    DISP_HEIGHT = getHeight();
    Display disp = Display.getDisplay(myJump);
    if(disp.numColors() < 256) {
      throw(new Exception("game requires 256 shades"));
    }
    if((DISP_WIDTH < 150) || (DISP_HEIGHT < 170)) {
      throw(new Exception("Screen too small"));
    }
    if((DISP_WIDTH > 250) || (DISP_HEIGHT > 250)) {
      throw(new Exception("Screen too large"));
    }
    FONT = getGraphics().getFont();
    FONT_HEIGHT = FONT.getHeight();
    SCORE_WIDTH = FONT.stringWidth("Score: 000");
    TIME_WIDTH = FONT.stringWidth("Time: " + myInitialString);
    if(myManager == null) {
      myManager = new JumpManager(CORNER_X, CORNER_Y + FONT_HEIGHT*2,
           DISP_WIDTH, DISP_HEIGHT - FONT_HEIGHT*2 - GROUND_HEIGHT);
    }
  }

  /**
   * This is called as soon as the application begins.
   */
  void start() {
    myGameOver = false;
    myDisplay.setCurrent(this);
    repaint();
  }

  /**
   * sets all variables back to their initial positions.
   */
  void reset() {
    myManager.reset();
    myScore = 0;
    myGameOver = false;
    myGameTicks = myInitialGameTicks;
    myOldGameTicks = myInitialGameTicks;
    repaint();
  }

  /**
   * clears the key states.
   */
  void flushKeys() {
    getKeyStates();
  }

  /**
   * This version of the game does not deal with what happens
   * when the game is hidden, so I hope it won't be hidden...
   * see the version in the next chapter for how to implement
   * hideNotify and showNotify.
   */
  protected void hideNotify() {
  }

  /**
   * This version of the game does not deal with what happens
   * when the game is hidden, so I hope it won't be hidden...
   * see the version in the next chapter for how to implement
   * hideNotify and showNotify.
   */
  protected void showNotify() {
  }

  //-------------------------------------------------------
  //   graphics methods

  /**
   * paint the game graphic on the screen.
   */
  public void paint(Graphics g) {
    // clear the screen:
    g.setColor(WHITE);
    g.fillRect(CORNER_X, CORNER_Y, DISP_WIDTH, DISP_HEIGHT);
    // color the grass green
    g.setColor(0, 255, 0);
    g.fillRect(CORNER_X, CORNER_Y + DISP_HEIGHT - GROUND_HEIGHT,
               DISP_WIDTH, DISP_HEIGHT);
    // paint the layer manager:
    try {
      myManager.paint(g);
    } catch(Exception e) {
      myJump.errorMsg(e);
    }
    // draw the time and score
    g.setColor(BLACK);
    g.setFont(FONT);
    g.drawString("Score: " + myScore,
                 (DISP_WIDTH - SCORE_WIDTH)/2,
                 DISP_HEIGHT + 5 - GROUND_HEIGHT, g.TOP|g.LEFT);
    g.drawString("Time: " + formatTime(),
                 (DISP_WIDTH - TIME_WIDTH)/2,
                 CORNER_Y + FONT_HEIGHT, g.TOP|g.LEFT);
    // write game over if the game is over
    if(myGameOver) {
      myJump.setNewCommand();
      // clear the top region:
      g.setColor(WHITE);
      g.fillRect(CORNER_X, CORNER_Y, DISP_WIDTH, FONT_HEIGHT*2 + 1);
      int goWidth = FONT.stringWidth("Game Over");
      g.setColor(BLACK);
      g.setFont(FONT);
      g.drawString("Game Over", (DISP_WIDTH - goWidth)/2,
                   CORNER_Y + FONT_HEIGHT, g.TOP|g.LEFT);
    }
  }

  /**
   * a simple utility to make the number of ticks look like a time...
   */
  public String formatTime() {
    if((myGameTicks / 16) + 1 != myOldGameTicks) {
      myTimeString = "";
      myOldGameTicks = (myGameTicks / 16) + 1;
      int smallPart = myOldGameTicks % 60;
      int bigPart = myOldGameTicks / 60;
      myTimeString += bigPart + ":";
      if(smallPart / 10 < 1) {
        myTimeString += "0";
      }
      myTimeString += smallPart;
    }
    return(myTimeString);
  }

  //-------------------------------------------------------
  //   game movements

  /**
   * Tell the layer manager to advance the layers and then
   * update the display.
   */
  void advance() {
    myGameTicks--;
    myScore += myManager.advance(myGameTicks);
    if(myGameTicks == 0) {
      setGameOver();
    }
    // paint the display
    try {
      paint(getGraphics());
      flushGraphics();
    } catch(Exception e) {
      myJump.errorMsg(e);
    }
  }

  /**
   * Respond to keystrokes.
   */
  public void checkKeys() {
    if(! myGameOver) {
      int keyState = getKeyStates();
      if((keyState & LEFT_PRESSED) != 0) {
        myManager.setLeft(true);
      }
      if((keyState & RIGHT_PRESSED) != 0) {
        myManager.setLeft(false);
      }
      if((keyState & UP_PRESSED) != 0) {
        myManager.jump();
      }
    }
  }

}
```

Using the Graphics Class with a GameCanvas

Chapter 2 covered using the `Graphics` class. In this section, I just go over the main points of how the `Graphics` class is used in the example game.

In the Tumbleweed game I need to draw a cowboy walking through a prairie jumping over tumbleweeds. Figure 3-1 shows the game.

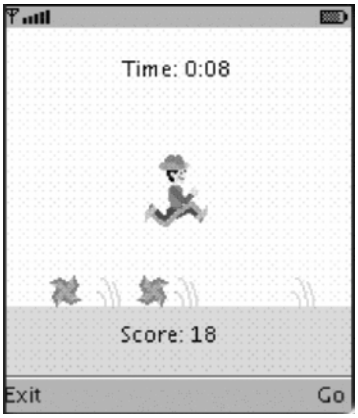

Figure 3-1. The Tumbleweed game

As you can see, I've put the score on the bottom and the time remaining on the top. (To simplify the game, I just have it end when the player runs out of time.) As the cowboy is walking along, I'd like his background to scroll to the right or to the left (otherwise he won't have very far to go on such a small screen...), but I'd like the time and the score to stay in place. To accomplish this, I have my `JumpCanvas` class take care of painting the stable strip on the top and the bottom of the screen, and I delegate the interesting graphics to the `LayerManager` (more details on that in the next section).

When the `JumpCanvas` is first created, you start by analyzing the screen with which you have to work. Some of the information about the screen's capacities comes from the `Graphics` object, some from the display object, and some directly from methods of the `GameCanvas`. This information calculates where the objects should be placed, including calculating the dimensions of the region that will be painted by the `LayerManager` subclass (`JumpManager`). If you're interested in maintaining Java's "write once, run anywhere" philosophy, it's obviously better to base the screen layout on the (dynamically determined) dimensions of the current screen rather than basing the dimensions on fixed constants. Of course, if massive changes to the game are needed when going from one target device to another, it may be better to save device memory by maintaining multiple versions of the game rather than putting all of the code together in a single version and using the display information to determine which version to run. In my example game, if the screen is too different from the screen I wrote the game for (the emulator that came with the toolkit in this case), I throw an `Exception` that the `Jump` class will catch and show it to the user as an `Alert`. When using this technique professionally, you'd of course make sure the `Alert` clearly states that the user needs to download a different version of the game for the current device.

At the risk of belaboring the obvious, I'll point out that once I know the appropriate sizes for the top and bottom regions, the `paint(Graphics g)` method paints the top one white and the bottom one green with `g.fillRect()`, and then the method `g.drawString()` adds the time and the score. Chapter 2 discussed the `drawString()` method. (Don't ask me why my prairie has both green grass and tumbleweeds; my only excuse is that I know more about Java than I know about the Wild West...).

Using the LayerManager Class

The interesting graphical objects in an MIDP game are usually represented by subclasses of the `javax.microedition.lcdui.game.Layer` class. The background layers could be instances of `javax.microedition.lcdui.game.TiledLayer`, and the player (and his enemies) would likely be instances of `javax.microedition.lcdui.game.Sprite`, both of which are subclasses of `Layer`. The `LayerManager` class helps you to organize all these graphical layers. The order in which you append your `Layers` to your `LayerManager` determines the order in which they'll be painted. (The first one appended is the last one painted.) The top layers will cover the lower layers, but you can allow parts of the lower layers to show through by creating image files that have transparent regions.

Probably the most useful aspect of the `LayerManager` class is that you can create a graphical painting that's much larger than the screen and then choose which section of it will appear on the screen. Imagine drawing a huge, elaborate drawing and then covering it with a piece of paper that has a small rectangular hole you can move. The whole drawing represents what you can stock into the `LayerManager`, and the hole is the window showing the part that appears on the screen at any given time. Allowing the possibility of a virtual screen that's much larger than the actual screen is extremely helpful for games on devices with small screens. It'll save you huge amounts of time and effort if, for example, your game involves a player exploring an elaborate dungeon (see Chapter 5 for just such an example). The confusing part is that this means you have to deal with two separate coordinate systems. The `Graphics` object of the `GameCanvas` has one coordinate system, but the various `Layers` need to be placed in the `LayerManager` according to the `LayerManager`'s coordinate system. So, keep in mind that the method `LayerManager.paint(Graphics g, int x, int y)` paints the layer on the screen according to the coordinates of the `GameCanvas`, and the method `LayerManager.setViewWindow(int x, int y, int width, int height)` sets the visible rectangle of the `LayerManager` in terms of the `LayerManager`'s coordinate system.

In my example I have a simple background (it's just a repeating series of patches of grass), but I'd like the cowboy to stay in the middle of the screen as he walks to the right and left, so I need to continuously change which part of the `LayerManager`'s graphical area is visible. I do this by calling the method `setViewWindow(int x, int y, int width, int height)` from the `paint(Graphics g)` method of my subclass of `LayerManager` (called `JumpManager`). More precisely, what happens is the following: The main loop in the `GameThread` calls `JumpCanvas.checkKeys()`, which queries the key states and tells the `JumpManager` class whether the cowboy should be walking to the right or to the left and whether he should be jumping. `JumpCanvas` passes this information along to `JumpManager` by calling the methods `setLeft(boolean left)or jump()`. If the message is to jump, the `JumpManager` calls `jump()` on the cowboy `Sprite`. If the message is that the cowboy is going to the left (or similarly to the right), then when the `GameThread` calls the `JumpCanvas` to tell the `JumpManager` to advance (in the next step of the loop), the `JumpManager` tells the cowboy `Sprite` to move one pixel to the left and compensates by moving the view window one pixel to the right to keep the cowboy in the center of the screen. You can accomplish these two actions by incrementing the field `myCurrentLeftX` (which is the X coordinate that's sent to the method `setViewWindow(int x, int y, int width, int height)`) and then calling `myCowboy .advance(gameTicks, myLeft)`. Of course, I could keep the cowboy centered by not moving him and not appending him to the `LayerManager` but, rather, painting him separately afterward, but it's easier to keep track of everything by putting all of the moving graphics on one set of layers and then keeping the view window focused on the cowboy `Sprite`. While telling the cowboy to advance his position, I also have the tumbleweed `Sprite`s advance their positions, and I have the grass `TiledLayer` advance its animation. Then I check if the cowboy has collided with any tumbleweeds (I'll go into more detail about those steps in the following sections). After moving the game pieces around, the `JumpManager` calls the method `wrap()` to see if the view window has reached the edge of the background and, if so, move all of the game objects so that the background appears to continue indefinitely in both directions. Then the `JumpCanvas` repaints everything, and the game loop begins again.

I'll just add a few words here about the method `wrap()`. The class `LayerManager` unfortunately doesn't have a built-in wrapping capability for the case in which you have a simple background you'd like to have repeat indefinitely. The `LayerManager`'s graphical area will appear to wrap when the coordinates sent to `setViewWindow (int x, int y, int width, int height)` exceed the value `Integer.MAX_VALUE`, but that's unlikely to help you. Thus, you have to write your own functions to prevent the player `Sprite` from leaving the region that contains background graphics. In my example, the background grass repeats after the number of pixels given by `Grass.TILE_WIDTH*Grass.CYCLE`. So, whenever the X coordinate of the view window (`myCurrentLeftX`) is an integer multiple of the length of the background, I move the view window back to the center and also move all of the `Sprite`s in the same direction, which seamlessly prevents the player from reaching the edge.

Listing 3-4 shows the code for `JumpManager.java`.

Listing 3-4. `JumpManager.java`

```
package net.frog_parrot.jump;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * This handles the graphics objects.
 *
 * @author Carol Hamer
 */
public class JumpManager extends javax.microedition.lcdui.game.LayerManager {

  //---------------------------------------------------------
  //    dimension fields
  //  (constant after initialization)

  /**
   * The X coordinate of the place on the game canvas where
   * the LayerManager window should appear, in terms of the
   * coordinates of the game canvas.
   */
  static int CANVAS_X;

  /**
   * The Y coordinate of the place on the game canvas where
   * the LayerManager window should appear, in terms of the
   * coordinates of the game canvas.
   */
  static int CANVAS_Y;

  /**
   * The width of the display window.
   */
  static int DISP_WIDTH;

  /**
   * The height of this object's graphical region. This is
   * the same as the height of the visible part because
   * in this game the layer manager's visible part scrolls
   * only left and right but not up and down.
   */
  static int DISP_HEIGHT;
```

```
//---------------------------------------------------------
//   game object fields

/**
 * the player's object.
 */
private Cowboy myCowboy;

/**
 * the tumbleweeds that enter from the left.
 */
private Tumbleweed[] myLeftTumbleweeds;

/**
 * the tumbleweeds that enter from the right.
 */
private Tumbleweed[] myRightTumbleweeds;

/**
 * the object representing the grass in the background..
 */
private Grass myGrass;

/**
 * Whether the player is currently going left.
 */
private boolean myLeft;

/**
 * The leftmost X coordinate that should be visible on the
 * screen in terms of this objects internal coordinates.
 */
private int myCurrentLeftX;

//-----------------------------------------------------
//    gets/sets

/**
 * This tells the player to turn left or right.
 * @param left whether the turn is toward the left.
 */
void setLeft(boolean left) {
  myLeft = left;
}
```

```
//---------------------------------------------------------
//    initialization and game state changes

/**
 * Constructor sets the data and constructs the graphical objects.
 * @param x The X coordinate of the place on the game canvas where
 * the LayerManager window should appear, in terms of the
 * coordinates of the game canvas.
 * @param y The Y coordinate of the place on the game canvas where
 * the LayerManager window should appear, in terms of the
 * coordinates of the game canvas.
 * @param width the width of the region that is to be
 * occupied by the LayoutManager.
 * @param height the height of the region that is to be
 * occupied by the LayoutManager.
 */
public JumpManager(int x, int y, int width, int height)
    throws Exception {
  CANVAS_X = x;
  CANVAS_Y = y;
  DISP_WIDTH = width;
  DISP_HEIGHT = height;
  myCurrentLeftX = Grass.CYCLE*Grass.TILE_WIDTH;
  setViewWindow(0, 0, DISP_WIDTH, DISP_HEIGHT);
  // create the player:
  if(myCowboy == null) {
    myCowboy = new Cowboy(myCurrentLeftX + DISP_WIDTH/2,
                          DISP_HEIGHT - Cowboy.HEIGHT - 2);
    append(myCowboy);
  }
  // create the tumbleweeds to jump over:
  if(myLeftTumbleweeds == null) {
    myLeftTumbleweeds = new Tumbleweed[2];
    for(int i = 0; i < myLeftTumbleweeds.length; i++) {
      myLeftTumbleweeds[i] = new Tumbleweed(true);
      append(myLeftTumbleweeds[i]);
    }
  }
  if(myRightTumbleweeds == null) {
    myRightTumbleweeds = new Tumbleweed[2];
    for(int i = 0; i < myRightTumbleweeds.length; i++) {
      myRightTumbleweeds[i] = new Tumbleweed(false);
      append(myRightTumbleweeds[i]);
    }
    }
    // create the background object:
    if(myGrass == null) {
      myGrass = new Grass();
      append(myGrass);
    }
  }

  /**
   * sets all variables back to their initial positions.
   */
  void reset() {
    if(myGrass != null) {
      myGrass.reset();
    }
    if(myCowboy != null) {
      myCowboy.reset();
    }
    if(myLeftTumbleweeds != null) {
      for(int i = 0; i < myLeftTumbleweeds.length; i++) {
        myLeftTumbleweeds[i].reset();
      }
    }
    if(myRightTumbleweeds != null) {
      for(int i = 0; i < myRightTumbleweeds.length; i++) {
        myRightTumbleweeds[i].reset();
      }
    }
    myLeft = false;
    myCurrentLeftX = Grass.CYCLE*Grass.TILE_WIDTH;
  }

  //-------------------------------------------------------
  //   graphics methods

  /**
   * paint the game graphic on the screen.
   */
  public void paint(Graphics g) {
    setViewWindow(myCurrentLeftX, 0, DISP_WIDTH, DISP_HEIGHT);
    paint(g, CANVAS_X, CANVAS_Y);
  }
```

```
/**
 * If the cowboy gets to the end of the graphical region,
 * move all of the pieces so that the screen appears to wrap.
 */
private void wrap() {
  if(myCurrentLeftX % (Grass.TILE_WIDTH*Grass.CYCLE) == 0) {
    if(myLeft) {
      myCowboy.move(Grass.TILE_WIDTH*Grass.CYCLE, 0);
      myCurrentLeftX += (Grass.TILE_WIDTH*Grass.CYCLE);
      for(int i = 0; i < myLeftTumbleweeds.length; i++) {
        myLeftTumbleweeds[i].move(Grass.TILE_WIDTH*Grass.CYCLE, 0);
      }
      for(int i = 0; i < myRightTumbleweeds.length; i++) {
        myRightTumbleweeds[i].move(Grass.TILE_WIDTH*Grass.CYCLE, 0);
      }
    } else {
      myCowboy.move(-(Grass.TILE_WIDTH*Grass.CYCLE), 0);
      myCurrentLeftX -= (Grass.TILE_WIDTH*Grass.CYCLE);
      for(int i = 0; i < myLeftTumbleweeds.length; i++) {
        myLeftTumbleweeds[i].move(-Grass.TILE_WIDTH*Grass.CYCLE, 0);
      }
      for(int i = 0; i < myRightTumbleweeds.length; i++) {
        myRightTumbleweeds[i].move(-Grass.TILE_WIDTH*Grass.CYCLE, 0);
      }
    }
  }
}

//-------------------------------------------------------
//  game movements

/**
 * Tell all of the moving components to advance.
 * @param gameTicks the remaining number of times that
 *        the main loop of the game will be executed
 *        before the game ends.
 * @return the change in the score after the pieces
 *         have advanced.
 */
int advance(int gameTicks) {
  int retVal = 0;
  // first you move the view window
  // (so you are showing a slightly different view of
    // the manager's graphical area.)
    if(myLeft) {
      myCurrentLeftX--;
    } else {
      myCurrentLeftX++;
    }
    // now you tell the game objects to move accordingly.
    myGrass.advance(gameTicks);
    myCowboy.advance(gameTicks, myLeft);
    for(int i = 0; i < myLeftTumbleweeds.length; i++) {
      retVal += myLeftTumbleweeds[i].advance(myCowboy, gameTicks,
                     myLeft, myCurrentLeftX, myCurrentLeftX + DISP_WIDTH);
      retVal -= myCowboy.checkCollision(myLeftTumbleweeds[i]);
    }
    for(int i = 0; i < myLeftTumbleweeds.length; i++) {
      retVal += myRightTumbleweeds[i].advance(myCowboy, gameTicks,
           myLeft, myCurrentLeftX, myCurrentLeftX + DISP_WIDTH);
      retVal -= myCowboy.checkCollision(myRightTumbleweeds[i]);
    }
    // now you check if you have reached an edge of the viewable
    // area, and if so, you move the view area and all of the
    // game objects so that the game appears to wrap.
    wrap();
    return(retVal);
  }

  /**
   * Tell the cowboy to jump..
   */
  void jump() {
    myCowboy.jump();
  }

}
```

Using the Sprite Class

A Sprite is a graphical object represented by one image (at a time). The fact that a Sprite is composed of only one image is the principal difference between a Sprite and a TiledLayer, which is a region that's covered with images that can be manipulated. (The Sprite class has a few extra features, but the fact that it uses one image rather than filling an area with images is the most obvious difference.) So, a `Sprite` is generally used for small, active game objects (such as your spaceship and the asteroids that are coming to crash into it), and a `TiledLayer` would be more likely to be used for an animated background. One cool feature of `Sprite` is that even though a `Sprite` is represented by only one image at a time, it can be easily represented by different images under different circumstances, including by a series of images that make up an animation. In my example game, the cowboy has three different images in which he's walking and one in which he's jumping. All of the images used for a given `Sprite` need to be stored together in a single image file. (To indicate where to find the image file to use, send the address of the image within the `jar` in the same format that's used to find resources in the method `Class.getResource()`; see Chapter 1 for more details.) Multiple frames are stored in a single `Image` object, which is a convenience that means you don't have to manipulate multiple `Image` objects to determine which face your `Sprite` is wearing at any given time. Figure 3-2 shows the image file for the cowboy `Sprite`.

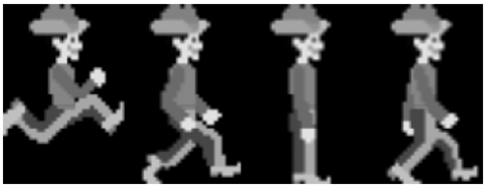

Figure 3-2. The image file to use for the cowboy `Sprite`

The tumbleweed image file consists of three frames that give a rolling animation when shown in sequence (see Figure 3-3).

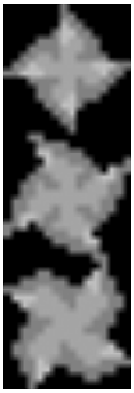

Figure 3-3. The tumbleweed image

The way to select which frame is shown at any given time is intuitive. First, if your image file comprises multiple images (as these two do), you should construct the `Sprite` with the constructor that specifies the width and height (in pixels) that you'd like your `Sprite` to be. The width and height of the `Image` should be integer multiples of the width and height you send to the constructor. In other words, the computer should be able to divide your image file evenly into rectangles of the size you specify. As you can see from the previous examples, you can arrange the subimages arranged horizontally or vertically. You can even arrange them in a grid with multiple rows and columns. Then, to identify the individual frames, they're numbered starting with zero at the top-left corner, continuing to the right and then continuing to the lower rows, in the same order in which you're reading the letters on this page. To select the frame that's currently displayed, use the method `setFrame(int sequenceIndex)`, sending the frame number as an argument.

The `Sprite` class has some added support for animation that allows you to define a frame sequence with the method `setFrameSequence(int[] sequence)`. As you can see in Listing 3-5, I've set a frame sequence of `{ 1, 2, 3, 2 }` for my cowboy and `{ 0, 1, 2 }` for my tumbleweed. (Note that for the tumbleweed, the frame sequence I'd like to use is just the default frame sequence, so I don't have to set the Tumbleweed's frame sequence in the code.) To advance your `Sprite`'s animation from one frame to the next, you use the method `nextFrame()` (or, if you prefer, `prevFrame()`). This is convenient in cases such as my tumbleweed where all the available frames are used in the animation. It's slightly less convenient in cases such as my cowboy that have an image or images that fall outside of the frame sequence of the animation. This is because once a frame sequence has been set, the argument to the method `setFrame(int sequenceIndex)` gives the index of an entry in the frame sequence instead of giving the index of the frame itself. What that means is that once I've set my cowboy's frame sequence to `{ 3, 2, 1, 2 }`, if I call `setFrame(0)`, it'll show frame number `1`, `setFrame(1)` will show frame number `2`, `setFrame(2)` will show frame number `3`, and `setFrame(3)` will show frame number `2`. But when the cowboy is jumping, I'd like it to show frame number `0`, which is no longer accessible. So, when my cowboy jumps, I have to set my frame sequence to `null` before calling `setFrame(0)`, and then I have to set my frame sequence back to the animation sequence `{ 1, 2, 3, 2 }` afterward. Listing 3-5 shows this in the methods `jump()` and `advance(int tickCount, boolean left)`.

In addition to changing your `Sprite`'s appearance by changing frames, you can change it by applying simple transforms such as rotations or mirror images. Both my cowboy and my tumbleweed in Listings 3-5 and 3-6 can be going either left or right, so of course I need to use the mirror image transform to change from one direction to the other. Once you start applying transforms, you need to keep track of the `Sprite`'s reference pixel. This is because when you transform your `Sprite`, the reference pixel is the pixel that doesn't move. You might expect that if your `Sprite`'s image is square, then after a transformation the `Sprite`'s image will continue to occupy the same square of area on the screen. This isn't the case. The best way to illustrate what happens is to imagine an example `Sprite` of a standing person facing left whose reference pixel has been defined to be the tip of his toe. Then after applying a 90-degree rotation, your person will be in the spot he'd be in if he had tripped and fallen forward. Clearly, this has its applications if your `Sprite` has a special pixel (such as the tip of an arrow) that should stay put after a transformation. But if you want your `Sprite` to continue to occupy the same space on the screen after a transformation, then you should first call `defineReferencePixel(int x, int y)` and set your `Sprite`'s reference pixel to the center of the `Sprite`, as I did in the `Cowboy` constructor in Listing 3-5. (Another trick to keep the `Sprite` from moving after a transformation is to use `getX()` and `getY()` to get the absolute coordinates of the `Sprite`'s upper-left corner before the transformation and then after the transformation, use `setPosition()` to set the upper-left corner back to the earlier location.) Be aware that the coordinates in `defineReferencePixel(int x, int y)` are relative to the top corner of the `Sprite` whereas the coordinates sent to `setRefPixelPosition(int x, int y)` tell where to place the Sprite's reference pixel on the screen in terms of the screen's coordinates. To be more precise, the coordinates sent to `setRefPixelPosition(int x, int y)` refer to the coordinate system of the `Canvas` if the `Sprite` is painted directly onto the `Canvas`, but if the `Sprite` is painted by a `LayerManager`, these coordinates should be given in terms of the `LayerManager`'s coordinate system. (I explained how these coordinate systems fit together in the earlier "Using the LayerManager Class" section.) The coordinates in the various methods to set and get the position of the reference pixel or to set and get the position of the pixel in the top-left corner refer to the coordinates of the appropriate `Canvas` or `LayerManager`. Also note that if you perform multiple transformations, the later transformations are applied to the original image and not to its current state. In other words, if I apply `setTransform(TRANS_MIRROR)` twice in a row, the second transform won't mirror my image back to its original position; it'll just repeat the action of setting the `Image` to being a mirror image of the original `Image`. If you want to set a transformed `Sprite` back to normal, use `setTransform(TRANS_NONE)`. This is illustrated in the top of the `Cowboy.advance(int tickCount, boolean left)` method.

Another great feature of the `Layer` class (including both `Sprites` and `TiledLayers`) is the support it gives you for placing your objects in relative terms instead of in absolute terms. If your `Sprite` needs to move over three pixels regardless of where it currently is, you can just call `move(int x, int y)`, sending it the `x` and `y` distances it should move from its current position, as opposed to calling `setRefPixelPosition(int x, int y)` with the absolute coordinates of the `Sprite`'s new location. Even more useful is the set of `collidesWith()` methods. This allows you to check if a `Sprite` is occupying the same space as another `Sprite` or `TiledLayer` or even an `Image`. It's easy to see that this saves you quite a number of comparisons, especially since when you send the `pixelLevel` argument as `true`, it will consider the two `Layers` as having collided only if their opaque pixels overlap.

In the Tumbleweed game, after advancing all of the `Sprites`, I check if the cowboy has collided with any tumbleweeds. (This happens in the

`Cowboy.checkCollision(Tumbleweed tumbleweed)` method that's called from `JumpManager.advance(int gameTicks)`.) I check the collisions between the cowboy and all of the tumbleweeds each time because it automatically returns `false` for any tumbleweeds that aren't currently visible anyway, so I'm not really being wasteful by checking the cowboy against tumbleweeds that aren't currently in use. In many cases, however, you can save some effort by checking only for collisions that you know are possible rather than checking all of the `Sprite`s against each other. Note that in my example I don't bother to check if the tumbleweeds collide with each other or if anything collides with the background grass because that's irrelevant. If you're checking for pixel-level collisions, you'll want to be sure your images have a transparent background. (This is also helpful in general so that your `Sprite` doesn't paint an ugly rectangle of background color over another `Sprite` or `Image`.) You can find some discussion of creating the image files correctly in the sidebar "Making Image Files" in Chapter 1.

Listing 3-5 shows the code for `Cowboy.java`.

Listing 3-5. `Cowboy.java`

```
package net.frog_parrot.jump;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * This class represents the player.
 *
 * @author Carol Hamer
 */
public class Cowboy extends Sprite {

  //---------------------------------------------------------
  //    dimension fields

  /**
   * The width of the cowboy's bounding rectangle.
   */
  static final int WIDTH = 32;

  /**
   * The height of the cowboy's bounding rectangle.
   */
  static final int HEIGHT = 48;
```

```
/**
 * This is the order that the frames should be displayed
 * for the animation.
 */
static final int[] FRAME_SEQUENCE = { 3, 2, 1, 2 };

//---------------------------------------------------------
//    instance fields

/**
 * the X coordinate of the cowboy where the cowboy starts
 * the game.
 */
private int myInitialX;

/**
 * the Y coordinate of the cowboy when not jumping.
 */
private int myInitialY;

/**
 * The jump index that indicates that no jump is
 * currently in progress..
 */
private int myNoJumpInt = -6;

/**
 * Where the cowboy is in the jump sequence.
 */
private int myIsJumping = myNoJumpInt;

/**
 * If the cowboy is currently jumping, this keeps track
 * of how many points have been scored so far during
 * the jump.  This helps the calculation of bonus points since
 * the points being scored depend on how many tumbleweeds
 * are jumped in a single jump.
 */
private int myScoreThisJump = 0;

//---------------------------------------------------------
//   initialization
```

```
/**
 * constructor initializes the image and animation.
 */
public Cowboy(int initialX, int initialY) throws Exception {
  super(Image.createImage("/images/cowboy.png"),
        WIDTH, HEIGHT);
  myInitialX = initialX;
  myInitialY = initialY;
  // you define the reference pixel to be in the middle
  // of the cowboy image so that when the cowboy turns
  // from right to left (and vice versa) he does not
  // appear to move to a different location.
  defineReferencePixel(WIDTH/2, 0);
  setRefPixelPosition(myInitialX, myInitialY);
  setFrameSequence(FRAME_SEQUENCE);
}

//---------------------------------------------------------
//   game methods

/**
 * If the cowboy has landed on a tumbleweed, you decrease
 * the score.
 */
int checkCollision(Tumbleweed tumbleweed) {
  int retVal = 0;
  if(collidesWith(tumbleweed, true)) {
    retVal = 1;
    // once the cowboy has collided with the tumbleweed,
    // that tumbleweed is done for now, so you call reset
    // which makes it invisible and ready to be reused.
    tumbleweed.reset();
  }
  return(retVal);
}

/**
 * set the cowboy back to its initial position.
 */
void reset() {
  myIsJumping = myNoJumpInt;
  setRefPixelPosition(myInitialX, myInitialY);
  setFrameSequence(FRAME_SEQUENCE);
```

```
    myScoreThisJump = 0;
    // at first the cowboy faces right:
    setTransform(TRANS_NONE);
  }

  //------------------------------------------------------
  //   graphics

  /**
   * alter the cowboy image appropriately for this frame..
   */
  void advance(int tickCount, boolean left) {
    if(left) {
      // use the mirror image of the cowboy graphic when
      // the cowboy is going toward the left.
      setTransform(TRANS_MIRROR);
      move(-1, 0);
    } else {
      // use the (normal, untransformed) image of the cowboy
      // graphic when the cowboy is going toward the right.
      setTransform(TRANS_NONE);
      move(1, 0);
    }
    // this section advances the animation:
    // every third time through the loop, the cowboy
    // image is changed to the next image in the walking
    // animation sequence:
    if(tickCount % 3 == 0) { // slow the animation down a little
      if(myIsJumping == myNoJumpInt) {
        // if he's not jumping, set the image to the next
        // frame in the walking animation:
        nextFrame();
      } else {
        // if he's jumping, advance the jump:
        // the jump continues for several passes through
        // the main game loop, and myIsJumping keeps track
        // of where you are in the jump:
        myIsJumping++;
        if(myIsJumping < 0) {
          // myIsJumping starts negative, and while it's
          // still negative, the cowboy is going up.
          // here you use a shift to make the cowboy go up a
          // lot in the beginning of the jump and ascend
          // more and more slowly as he reaches his highest
          // position:
          setRefPixelPosition(getRefPixelX(),
                              getRefPixelY() - (2<<(-myIsJumping)));
        } else {
          // once myIsJumping is negative, the cowboy starts
          // going back down until he reaches the end of the
          // jump sequence:
          if(myIsJumping != -myNoJumpInt - 1) {
            setRefPixelPosition(getRefPixelX(),
                                getRefPixelY() + (2<<myIsJumping));
          } else {
            // once the jump is done, you reset the cowboy to
            // his nonjumping position:
            myIsJumping = myNoJumpInt;
            setRefPixelPosition(getRefPixelX(), myInitialY);
            // you set the image back to being the walking
            // animation sequence rather than the jumping image:
            setFrameSequence(FRAME_SEQUENCE);
            // myScoreThisJump keeps track of how many points
            // were scored during the current jump (to keep
            // track of the bonus points earned for jumping
            // multiple tumbleweeds).  Once the current jump is done,
            // you set it back to zero.
            myScoreThisJump = 0;
          }
        }
      }
    }
  }

  /**
   * makes the cowboy jump.
   */
  void jump() {
    if(myIsJumping == myNoJumpInt) {
      myIsJumping++;
      // switch the cowboy to use the jumping image
      // rather than the walking animation images:
      setFrameSequence(null);
      setFrame(0);
    }
  }
  /**
   * This is called whenever the cowboy clears a tumbleweed
   * so that more points are scored when more tumbleweeds
   * are cleared in a single jump.
   */
  int increaseScoreThisJump() {
    if(myScoreThisJump == 0) {
      myScoreThisJump++;
    } else {
      myScoreThisJump *= 2;
    }
    return(myScoreThisJump);
  }

}
```

Listing 3-6 shows the code for `Tumbleweed.java`.

Listing 3-6. `Tumbleweed.java`

```
package net.frog_parrot.jump;

import java.util.Random;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * This class represents the tumbleweeds that the player
 * must jump over.
 *
 * @author Carol Hamer
 */
public class Tumbleweed extends Sprite {

  //---------------------------------------------------------
  //   dimension fields

  /**
   * The width of the tumbleweed's bounding square.
   */
  static final int WIDTH = 16;
```

```
//-------------------------------------------------------
//    instance fields

/**
 * Random number generator to randomly decide when to appear.
 */
private Random myRandom = new Random();

/**
 * whether this tumbleweed has been jumped over.
 * This is used to calculate the score.
 */
private boolean myJumpedOver;

/**
 * whether this tumbleweed enters from the left.
 */
private boolean myLeft;

/**
 * the Y coordinate of the tumbleweed.
 */
private int myY;

//-------------------------------------------------------
//   initialization

/**
 * constructor initializes the image and animation.
 * @param left whether this tumbleweed enters from the left.
 */
public Tumbleweed(boolean left) throws Exception {
  super(Image.createImage("/images/tumbleweed.png"),
        WIDTH, WIDTH);
  myY = JumpManager.DISP_HEIGHT - WIDTH - 2;
  myLeft = left;
  if(!myLeft) {
    setTransform(TRANS_MIRROR);
  }
  myJumpedOver = false;
  setVisible(false);
}
```

```
//-----------------------------------------------------------
//   graphics

/**
 * move the tumbleweed back to its initial (inactive) state.
 */
void reset() {
  setVisible(false);
  myJumpedOver = false;
}

/**
 * alter the tumbleweed image appropriately for this frame..
 * @param left whether the player is moving left
 * @return how much the score should change by after this
 *         advance.
 */
int advance(Cowboy cowboy, int tickCount, boolean left,
            int currentLeftBound, int currentRightBound) {
  int retVal = 0;
  // if the tumbleweed goes outside of the display
  // region, set it to invisible since it is
  // no longer in use.
  if((getRefPixelX() + WIDTH <= currentLeftBound) ||
     (getRefPixelX() - WIDTH >= currentRightBound)) {
    setVisible(false);
  }
  // If the tumbleweed is no longer in use (i.e. invisible)
  // it is given a 1 in 100 chance (per game loop)
  // of coming back into play:
  if(!isVisible()) {
    int rand = getRandomInt(100);
    if(rand == 3) {
      // when the tumbleweed comes back into play,
      // you reset the values to what they should
      // be in the active state:
      myJumpedOver = false;
      setVisible(true);
      // set the tumbleweed's position to the point
      // where it just barely appears on the screen
      // to that it can start approaching the cowboy:
      if(myLeft) {
        setRefPixelPosition(currentRightBound, myY);
        move(-1, 0);
```

```
          } else {
            setRefPixelPosition(currentLeftBound, myY);
            move(1, 0);
          }
        }
      } else {
        // when the tumbleweed is active, you advance the
        // rolling animation to the next frame and then
        // move the tumbleweed in the right direction across
        // the screen.
        if(tickCount % 2 == 0) { // slow the animation down a little
          nextFrame();
        }
        if(myLeft) {
          move(-3, 0);
          // if the cowboy just passed the tumbleweed
          // (without colliding with it) you increase the
          // cowboy's score and set myJumpedOver to true
          // so that no further points will be awarded
          // for this tumbleweed until it goes off the screen
          // and then is later reactivated:
          if((! myJumpedOver) &&
             (getRefPixelX() < cowboy.getRefPixelX())) {
            myJumpedOver = true;
            retVal = cowboy.increaseScoreThisJump();
          }
        } else {
          move(3, 0);
          if((! myJumpedOver) &&
             (getRefPixelX() > cowboy.getRefPixelX() + Cowboy.WIDTH)) {
            myJumpedOver = true;
            retVal = cowboy.increaseScoreThisJump();
          }
        }
      }
      return(retVal);
    }

    /**
     * Gets a random int between
     * zero and the param upper.
     */
    public int getRandomInt(int upper) {
      int retVal = myRandom.nextInt() % upper;
    if(retVal < 0) {
      retVal += upper;
    }
    return(retVal);
  }

}
```

Using the TiledLayer Class

As mentioned, the TiledLayer class is similar to the Sprite class except that a TiledLayer can comprise multiple cells, each of which is painted with an individually set image frame. The other differences between TiledLayer and Sprite are mostly related to functionality missing from TiledLayer; TiledLayer has no transforms, reference pixel, or frame sequence.

Of course, the mere fact that you're simultaneously managing multiple images complicates things a bit. I'll explain it by going over my subclass of TiledLayer, which I've called Grass. This class represents a row of grass in the background that waves back and forth as the game is being played (see Figure 3-4). To make it more interesting, some of the cells in my TiledLayer have animated grasses, and others have no tall grasses and hence just consist of a green line representing the ground at the bottom of the cell.

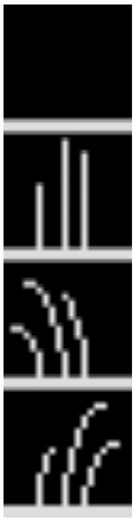

Figure 3-4. The image file that's used by the Grass TiledLayer

CAUTION *The tile index for* `Sprite` *starts with* `0`*, but the tile index for* `TiledLayer` *starts with* `1`*! This is a little confusing (it caused me to get an* `IndexOutOfBoundsException` *the first time I made a* `Sprite` *because I assumed that the* `Sprite` *images were numbered like the* `TiledLayer` *images). Yet the system is completely logical. In a* `TiledLayer`*, the tile index* `0` *indicates a blank tile (in other words, paint nothing in the cell if the cell's tile index is set to* `0`*). A* `Sprite`*, however, comprises only one cell, so if you want that cell to be blank, then you can just call* `setVisible(false)`*, meaning that* `Sprite` *doesn't need to reserve a special index to indicate a blank tile. This little confusion in the indices shouldn't pose a big problem, but it's something to keep in mind if you can't figure out why your animation appears to be displaying the wrong images. Aside from this point, the image file is divided into individual frames or tiles in* `TiledLayer` *just as in* `Sprite`*, explained previously.*

The first step in creating your `TiledLayer` is to decide how many rows and columns of cells you'll need. If you don't want your layer to be rectangular, it isn't a problem because any unused cells are by default set to being blank, which prevents them from getting in the way of other images. In my example, shown in Listing 3-7, I have only one row, and I calculate the number of columns based on the width of the screen.

Once you've set how many rows and columns you'll be using, you can fill each cell with a tile using the method `setCell(int col, int row, int tileIndex)`. The "Using the Sprite Class" section explained the `tileIndex` argument. If you'd like some of the cells to be filled with animated images, you need to create an animated tile by calling `createAnimatedTile(int staticTileIndex)`, which returns the tile index that has been allotted to your new animated tile. You can make as many animated tiles as you want, but remember that each animated tile can be used in multiple cells if you want the cells to display the same animation simultaneously. In my case, I create only one animated tile and reuse it because I want all of my animated grass to be waving in sync. The cells are set in the constructor of `Grass` in Listing 3-7. To advance the animation you don't get built-in frame-sequence functionality as in `Sprite`, so you have to set the frames with the method `setAnimatedTile(int animatedTileIndex, int staticTileIndex)`. This sets the current frame of the given animated tile. Thus, all the cells that have been set to contain the animated tile corresponding to `animatedTileIndex` will change to the image given by the argument `staticTileIndex`. To simplify the animation updates, it's easy to add your own frame-sequence functionality; see the method `Grass.advance(int tickCount)` for an idea of how to do it.

Listing 3-7 shows the code for the last class, `Grass.java`.

Listing 3-7. Grass.java

```
package net.frog_parrot.jump;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

/**
 * This class draws the background grass.
 *
 * @author Carol Hamer
 */
public class Grass extends TiledLayer {

  //---------------------------------------------------------
  //    dimension fields
  //  (constant after initialization)

  /**
   * The width of the square tiles that make up this layer..
   */
  static final int TILE_WIDTH = 20;

  /**
   * This is the order that the frames should be displayed
   * for the animation.
   */
  static final int[] FRAME_SEQUENCE = { 2, 3, 2, 4 };

  /**
   * This gives the number of squares of grass to put along
   * the bottom of the screen.
   */
  static int COLUMNS;

  /**
   * After how many tiles does the background repeat.
   */
  static final int CYCLE = 5;

  /**
   * the fixed Y coordinate of the strip of grass.
   */
  static int TOP_Y;
```

```
//---------------------------------------------------------
//    instance fields

/**
 * Which tile you are currently on in the frame sequence.
 */
private int mySequenceIndex = 0;

/**
 * The index to use in the static tiles array to get the
 * animated tile..
 */
private int myAnimatedTileIndex;

//---------------------------------------------------------
//   gets / sets

/**
 * Takes the width of the screen and sets my columns
 * to the correct corresponding number
 */
static int setColumns(int screenWidth) {
  COLUMNS = ((screenWidth / 20) + 1)*3;
  return(COLUMNS);
}

//---------------------------------------------------------
//   initialization

/**
 * constructor initializes the image and animation.
 */
public Grass() throws Exception {
  super(setColumns(JumpCanvas.DISP_WIDTH), 1,
        Image.createImage("/images/grass.png"),
        TILE_WIDTH, TILE_WIDTH);
  TOP_Y = JumpManager.DISP_HEIGHT - TILE_WIDTH;
  setPosition(0, TOP_Y);
  myAnimatedTileIndex = createAnimatedTile(2);
  for(int i = 0; i < COLUMNS; i++) {
    if((i % CYCLE == 0) || (i % CYCLE == 2)) {
      setCell(i, 0, myAnimatedTileIndex);
      } else {
        setCell(i, 0, 1);
      }
    }
  }

  //------------------------------------------------------
  //   graphics

  /**
   * sets the grass back to its initial position.
   */
  void reset() {
    setPosition(-(TILE_WIDTH*CYCLE), TOP_Y);
    mySequenceIndex = 0;
    setAnimatedTile(myAnimatedTileIndex, FRAME_SEQUENCE[mySequenceIndex]);
  }

  /**
   * alter the background image appropriately for this frame..
   * @param left whether the player is moving left
   */
  void advance(int tickCount) {
    if(tickCount % 2 == 0) { // slow the animation down a little
      mySequenceIndex++;
      mySequenceIndex %= 4;
      setAnimatedTile(myAnimatedTileIndex, FRAME_SEQUENCE[mySequenceIndex]);
    }
  }

}
```

So, now you've seen a basic game that illustrates how to use all of the classes of the `javax.microedition.lcdui.game` package, and the Tumbleweed game example especially shows how to take advantage of the graphics and animation features. In the next chapter, you'll take the same game and improve it by adding some more threads to play music and to optimize performance.