Java GAME Programming

Game Engine & FrameWork

Instead of writing games in an ad-hoc manner (which I did but found it grossly inefficient), I decided to put up a framework for programming games in Java. This is because many of the tasks involved in game programming are similar in many types of games. A framework not only helps in improving the productivity but also ensures high quality codes. Before presenting the entire framework, let's look at the individual pieces involved in game programming.

Custom Drawing

Game programming requires custom drawing. For a Swing application, we extend a `JPanel` (called `GameCanvas`) and override the `paintComponent(Graphics)` method to do custom drawing. (Older Java AWT appplications perform custom drawing on `java.awt.Canvas`, which is discouraged for Swing applications.) The `paintComponent()` method is not meant to be called directly, but called-back via the `repaint()` method. The `Graphics` context supports rendering of text (`drawString()`), primitive shapes (`drawXxx()`, `fillXxx()`), and bitmap images (`drawImage()`). For higher-quality graphics, we could use `Graphics2D` (of the Java 2D API) instead of the legacy `Graphics`. The custom drawing panel is usually designed as an inner class to the main game class, in order to directly access the private variables of the outer class - in particular, the game objects. The `GameCanvas` is a key-event source, as well as listener (via `addKeyListener(this)`). As source, it triggers `KeyEvent` upon key-pressed, key-released and key-typed. As a listener, it implements `KeyListener` interface and provides event handlers for key-pressed, key-released and key-typed.

The main game class is derived from a `JFrame`, as a typical Swing application. An instance of `GameCanvas` called `canvas` is instantiated and set as the content pane for the `JFrame`.

The game logic is supported in these methods:

- `gameInit()`: initialization codes. Run only once.
- `gameStart()`: to start or re-start the game.
- `gameShutdown()`: clean-up codes (e.g., write the high-score). Run once before the program terminates.
- `gameUpdate()`: to be called in the game loop for updating the position and state of the game objects, detecting collision and providing proper responses.
- `gameDraw(Graphics2D g2d)` or `gameDraw(Graphics g)`: render the graphics, to be called inside the `paintComponent()` of the drawing `JPanel`.
- `gameKeyPressed(int keyCode)`, `gameKeyReleased(int keyCode)`, and `gameKeyTyped(int keyCode)`: `KeyEvent` handler for key-pressed, key-released and key-typed.

The structure (i.e., template) of the graphics part of the a Java game is as follows:

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

public class GameMain extends JFrame {     // main class for the game - a swing application

   // Define constants for the game
   static final int CANVAS_WIDTH = 800;    // width and height of the drawing canvas
   static final int CANVAS_HEIGHT = 600;
   // ......

   // Define instance variables for the game objects
   // ......
   // ......

   // Handle for the custom drawing panel
   private GameCanvas canvas;

   // Constructor to initialize the UI components and game objects
   public GameMain() {
      // Initialize the game objects
      gameInit();

      // UI components
      canvas = new GameCanvas();
      canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
      this.setContentPane(canvas);

      // Other UI components such as button, score board, if any.
```

```
      // ......

      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.pack();
      this.setTitle("MY GAME");
      this.setVisible(true);
   }

   // ------ All the game related codes here ------

   // Initialize all the game objects, run only once.
   public void gameInit() { ...... }

   // Start and re-start the game.
   public void gameStart() { ...... }

   // Shutdown the game, clean up code that runs only once.
   public void gameShutdown() { ...... }

   // One step of the game.
   public void gameUpdate() { ...... }

   // Refresh the display after each step.
   // Use (Graphics g) as argument if you are not using Java 2D.
   public void gameDraw(Graphics2D g2d) { ...... }

   // Process a key-pressed event.
   public void gameKeyPressed(int keyCode) {
      switch (keyCode) {
         case KeyEvent.VK_UP:
            // ......
            break;
         case KeyEvent.VK_DOWN:
            // ......
            break;
         case KeyEvent.VK_LEFT:
            // ......
            break;
         case KeyEvent.VK_RIGHT:
            // ......
            break;
      }
   }

   // Process a key-released event.
   public void gameKeyReleased(int keyCode) { ...... }

   // Process a key-typed event.
   public void gameKeyTyped(char keyChar) { ...... }

   // Other methods
   // ......

   // Custom drawing panel, written as an inner class.
   class GameCanvas extends JPanel implements KeyListener {
      // Constructor
      public GameCanvas() {
         setFocusable(true);  // so that can receive key-events
         requestFocus();
         addKeyListener(this);
      }

      // Override paintComponent to do custom drawing.
      // Called back by repaint().
      @Override
      public void paintComponent(Graphics g) {
         Graphics2D g2d = (Graphics2D)g;  // if using Java 2D
         super.paintComponent(g2d);       // paint background
         setBackground(Color.BLACK);      // may use an image for background

         // Draw the game objects
         gameDraw(g2d);
      }

      // KeyEvent handlers
      @Override
      public void keyPressed(KeyEvent e) {
         gameKeyPressed(e.getKeyCode());
      }

      @Override
      public void keyReleased(KeyEvent e) {
         gameKeyReleased(e.getKeyCode());
```

```
      }

      @Override
      public void keyTyped(KeyEvent e) {
         gameKeyTyped(e.getKeyChar());
      }
   }

   // main
   public static void main(String[] args) {
      // Use the event dispatch thread to build the UI for thread-safety.
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            new GameMain();
         }
      });
   }
}
```

The purpose of the template (and framework) is to guide you on where to place your programming codes. In other words, do not modify the "fixed" portion of the template, but concentrate on implementing your game logic.

The above template implements only key listener. Some games may require mouse listener or mouse-motion listener. They can be added similar to key listener.

Init and Shutdown - `gameInit()` & `gameShutdown()`

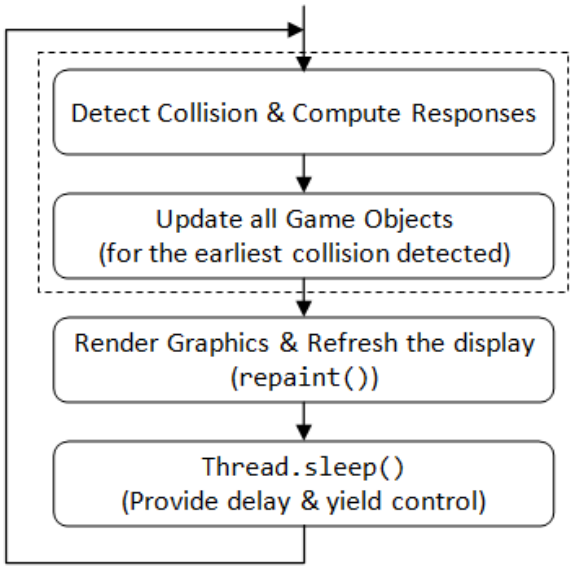
The `gameInit()` method is meant to be run once, to instantiate all the gaming objects, pre-load images and sound effects, among others. In this framework, `gameInit()` is called once in the constructor of the main application `GameMain`.

The `gameShutdown()` is also meant to be run once, for clean-up operations such as writing out the high-score.

Starting the Game Play - `gameStart()`

Once all the game objects are in place (via `gameInit()`), we can start the game by running a *game loop* to repeat the game steps.

In a typical single-player game, the game loop executes at a fixed interval. It calculates the new position of all the game objects and move them into the new position. It then detects collision among the game objects and provides responses, renders a new frame and pushes it to the screen.



We shall use the method `gameStart()` to start the play, or re-start after the the previous play. The `gameStart()` runs the game-loop as follows:

```
// Start (and re-start) the game.
public void gameStart() {
   // Regenerate the game objects for a new game
   ......
   // Game loop
   while (true) {
      // Update the state and position of all the game objects,
      // detect collisions and provide responses.
      gameUpdate();
      // Refresh the display.
      repaint();
      // Delay timer to provide the necessary delay to meet the target rate.
```

```
      ......
   }
}
```

Within the game loop, we invoke the `gameUpdate()` method to calculate the position of the game objects, update their states, detect collision and provide responses.

The game loop is written as an *infinite* loop. We need to keep track of the *state* of the game via `boolean` variables such as `gameOver` and `gamePaused`. (I will change it to an enumeration later.) We can then use these `boolean` flags to control the loop. For example,

```
// State of the game
boolean gameOver = false;
boolean gamePaused = false;
......
......

public void gameStart() {
   // Regenerate the game objects for a new game
   ......
   // Game loop
   while (true) {
      if (gameOver) break;  // break the loop to finish the current play
      if (!gamePaused) {
         // Update the state and position of all the game objects,
         // detect collisions and provide responses.
         gameUpdate();
      }
      // Refresh the display
      repaint();
      // Delay timer to provide the necessary delay to meet the target rate.
      ......
   }
}
```

Controlling the Refresh (Update) Rate of the Game

The monitor refreshes at 50-100 Hz. There is pointless for your game to update your game faster than the monitor. For some action games, you may wish to intercept the video buffer and update the game at the same rate as the monitor. For others like tetris and snake games, you may wish to update at the same rate as the player's response, says 1-5 moves per second.

Suppose that we wish to refresh the game at 5 moves per second. Each move takes 1000/5 = 200 milliseconds. The delay timer must provide (200 - time taken to run the earlier processes). This can be achieved as follows:

```
static final int UPDATE_RATE = 4;  // number of game update per second
static final long UPDATE_PERIOD = 1000000000L / UPDATE_RATE;  // nanoseconds

// State of the game
boolean gameOver = false;
boolean gamePaused = false;
......
......

public void gameStart() {
   long beginTime, timeTaken, timeLeft;
   // Regenerate the game objects for a new game
   ......
   // Game loop
   while (true) {
      beginTime = System.nanoTime();
      if (gameOver) break;  // break the loop to finish the current play
      if (!gamePaused) {
         // Update the state and position of all the game objects,
         // detect collisions and provide responses.
         gameUpdate();
      }
      // Refresh the display
      repaint();
      // Delay timer to provide the necessary delay to meet the target rate
      timeTaken = System.nanoTime() - beginTime;
      timeLeft = (UPDATE_PERIOD - timeTaken) / 1000000;  // in milliseconds
      if (timeLeft < 10) timeLeft = 10;   // set a minimum
      try {
         // Provides the necessary delay and also yields control so that other thread can do work.
         Thread.sleep(timeLeft);
      } catch (InterruptedException ex) { }
   }
}
```

JDK 1.5 provides a new timer called `System.nanoTime()` for measuring the elapsed time, which is reportedly more precise than the legacy `System.currentTimeMillis()`.

The static method `Thread.sleep()` suspends the current thread, and wait for the specified milliseconds before attempting to resume. This process serves two purposes. Firstly, it provides the necessary time delay needed to meet the target rate. Secondly, by suspending itself, another thread can resume and perform its task. In particular, the so-called *event dispatch thread*, which is responsible for processing input events (such as mouse-clicked, key-pressed) and refreshing the display, cannot be *starved*. Otherwise, the infamous *unresponsive user-interface* resulted, that is, the display is frozen and the system does not response to any input event.

Game Thread

Next, we need to run the game loop in its own thread - let's call it the game thread. We use a dedicated thread to run our game loop to ensure responsive user-interface (as mentioned above).

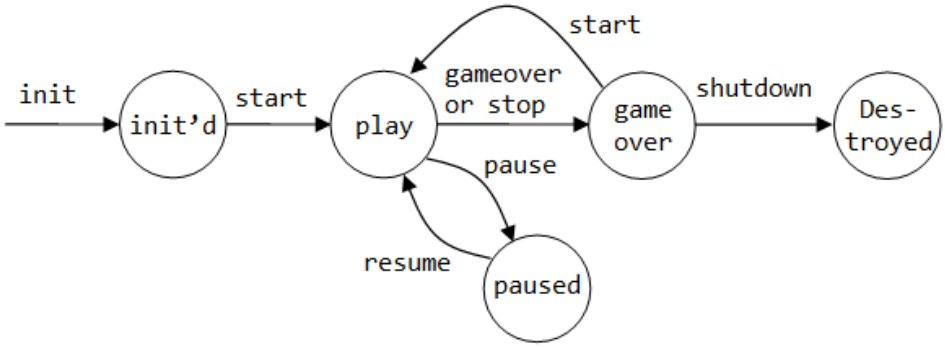
The game thread (called `GameThread`) is derived from the class `Thread`. It is written as an inner class. We override the `run()` method to program the running behavior of the thread. To start the play, we need to create a new instance of the `GameThread` and invoke the `start()` method, which will call-back the `run()` method to run the programmed behavior.

There is, however, an issue here. We would like to start the play via the `gameStart()` method. But we need to program the running behavior in the overridden `run()`. To resolve this program, I break the game starting method into two parts: `gameStart()` which simply create and run a new `GameThread`; and `gameLoop()`, which is called by the `run()` method to run the game loop, as follows:

```
public class GameMain extends JFrame {  // main game class
   static final int UPDATE_RATE = 4;    // number of game update per second
   static final long UPDATE_PERIOD = 1000000000L / UPDATE_RATE;  // nanoseconds

   // State of the game
   boolean gameOver = false;
   boolean gamePaused = false;
   ......
   ......

   // To start and re-start the game.
   public void gameStart() {
      // Create a new thread
      Thread gameThread =  new Thread() {
         // Override run() to provide the running behavior of this thread.
         @Override
         public void run() {
            gameLoop();
         }
      };
      // Start the thread. start() calls run(), which in turn calls gameLoop().
      gameThread.start();
   }

   // Run the game loop here.
   private void gameLoop() {
      // Regenerate the game objects for a new game
      ......

      // Game loop
      long beginTime, timeTaken, timeLeft;
      while (true) {
         beginTime = System.nanoTime();
         if (gameOver) break;  // break the loop to finish the current play
         if (!gamePaused) {
            // Update the state and position of all the game objects,
            // detect collisions and provide responses.
            gameUpdate();
         }
         // Refresh the display
         repaint();
         // Delay timer to provide the necessary delay to meet the target rate
         timeTaken = System.nanoTime() - beginTime;
         timeLeft = (UPDATE_PERIOD - timeTaken) / 1000000;  // in milliseconds
         if (timeLeft < 10) timeLeft = 10;   // set a minimum
         try {
            // Provides the necessary delay and also yields control so that other thread can do work.
            Thread.sleep(timeLeft);
         } catch (InterruptedException ex) { }
      }
   }
}
```

Game States

Let's try to handle the game state in a more systematic way, instead of using boolean flags (such as `gameOver` and `gamePaused`). The state diagram for a typical game is as illustrated below:



We shall define a nested static enumeration to represent the game states in the `GameMain` class as follow:

```
public class GameMain extends JFrame
   // Enumeration for the states of the game.
   static enum State {
      INITIALIZED, PLAYING, PAUSED, GAMEOVER, DESTROYED
   }

   static State state;   // current state of the game

   ......
}
```

The enumeration `State` and the instance variable `state` are declared as static, which can be accessed via the classname directly. For example, you can manipulate the `state`:

```
// in GameMain class
state = State.GAMEOVER;
......
switch (state) {
   case INITIALIZED:
      ......
      break;
   case PLAYING:
      ......
      break;
   case PAUSED:
      ......
      break;
   case GAMEOVER:
      ......
      break;
   case DESTROYED:
      ......
      break;
}
```

The Complete Java Game Framework

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

public class GameMain extends JFrame {     // main class for the game as a Swing application

   // Define constants for the game
   static final int CANVAS_WIDTH = 800;    // width and height of the game screen
   static final int CANVAS_HEIGHT = 600;
   static final int UPDATE_RATE = 4;    // number of game update per second
   static final long UPDATE_PERIOD = 1000000000L / UPDATE_RATE;  // nanoseconds
   // ......

   // Enumeration for the states of the game.
   static enum State {
      INITIALIZED, PLAYING, PAUSED, GAMEOVER, DESTROYED
   }
   static State state;   // current state of the game

   // Define instance variables for the game objects
   // ......
   // ......

   // Handle for the custom drawing panel
   private GameCanvas canvas;

   // Constructor to initialize the UI components and game objects
```

```
   public GameMain() {
      // Initialize the game objects
      gameInit();

      // UI components
      canvas = new GameCanvas();
      canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
      this.setContentPane(canvas);

      // Other UI components such as button, score board, if any.
      // ......

      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.pack();
      this.setTitle("MY GAME");
      this.setVisible(true);
   }

   // All the game related codes here

   // Initialize all the game objects, run only once in the constructor of the main class.
   public void gameInit() {
      // ......
      state = State.INITIALIZED;
   }

   // Shutdown the game, clean up code that runs only once.
   public void gameShutdown() {
      // ......
   }

   // To start and re-start the game.
   public void gameStart() {
      // Create a new thread
      Thread gameThread =  new Thread() {
         // Override run() to provide the running behavior of this thread.
         @Override
         public void run() {
            gameLoop();
         }
      };
      // Start the thread. start() calls run(), which in turn calls gameLoop().
      gameThread.start();
   }

   // Run the game loop here.
   private void gameLoop() {
      // Regenerate the game objects for a new game
      // ......
      state = State.PLAYING;

      // Game loop
      long beginTime, timeTaken, timeLeft;
      while (true) {
         beginTime = System.nanoTime();
         if (state == State.GAMEOVER) break;  // break the loop to finish the current play
         if (state == State.PLAYING) {
            // Update the state and position of all the game objects,
            // detect collisions and provide responses.
            gameUpdate();
         }
         // Refresh the display
         repaint();
         // Delay timer to provide the necessary delay to meet the target rate
         timeTaken = System.nanoTime() - beginTime;
         timeLeft = (UPDATE_PERIOD - timeTaken) / 1000000L;  // in milliseconds
         if (timeLeft < 10) timeLeft = 10;   // set a minimum
         try {
            // Provides the necessary delay and also yields control so that other thread can do work.
            Thread.sleep(timeLeft);
         } catch (InterruptedException ex) { }
      }
   }

   // Update the state and position of all the game objects,
   // detect collisions and provide responses.
   public void gameUpdate() { ..... }

   // Refresh the display. Called back via rapaint(), which invoke the paintComponent().
   private void gameDraw(Graphics2D g2d) {
      switch (state) {
         case INITIALIZED:
            // ......
```

```
            break;
         case PLAYING:
            // ......
            break;
         case PAUSED:
            // ......
            break;
         case GAMEOVER:
            // ......
            break;
      }
      // ......
   }

   // Process a key-pressed event. Update the current state.
   public void gameKeyPressed(int keyCode) {
      switch (keyCode) {
         case KeyEvent.VK_UP:
            // ......
            break;
         case KeyEvent.VK_DOWN:
            // ......
            break;
         case KeyEvent.VK_LEFT:
            // ......
            break;
         case KeyEvent.VK_RIGHT:
            // ......
            break;
      }
   }

   // Process a key-released event.
   public void gameKeyReleased(int keyCode) {  }

   // Process a key-typed event.
   public void gameKeyTyped(char keyChar) {  }

   // Other methods
   // ......

   // Custom drawing panel, written as an inner class.
   class GameCanvas extends JPanel implements KeyListener {
      // Constructor
      public GameCanvas() {
         setFocusable(true);  // so that can receive key-events
         requestFocus();
         addKeyListener(this);
      }

      // Override paintComponent to do custom drawing.
      // Called back by repaint().
      @Override
      public void paintComponent(Graphics g) {
         Graphics2D g2d = (Graphics2D)g;
         super.paintComponent(g2d);   // paint background
         setBackground(Color.BLACK);  // may use an image for background

         // Draw the game objects
         gameDraw(g2d);
      }

      // KeyEvent handlers
      @Override
      public void keyPressed(KeyEvent e) {
         gameKeyPressed(e.getKeyCode());
      }

      @Override
      public void keyReleased(KeyEvent e) {
         gameKeyReleased(e.getKeyCode());
      }

      @Override
      public void keyTyped(KeyEvent e) {
         gameKeyTyped(e.getKeyChar());
      }
   }

   // main
   public static void main(String[] args) {
      // Use the event dispatch thread to build the UI for thread-safety.
      SwingUtilities.invokeLater(new Runnable() {
```

```
            @Override
            public void run() {
               new GameMain();
            }
         });
      }
   }
```

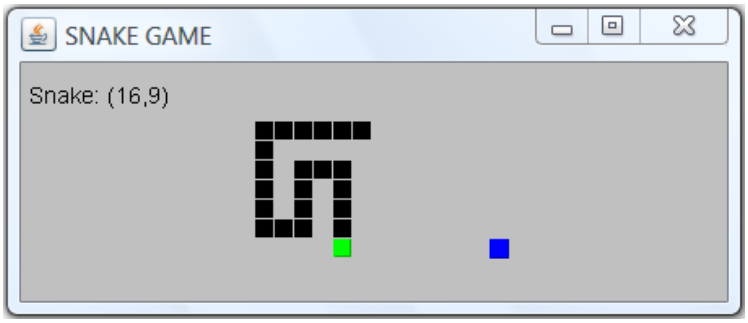
Summary

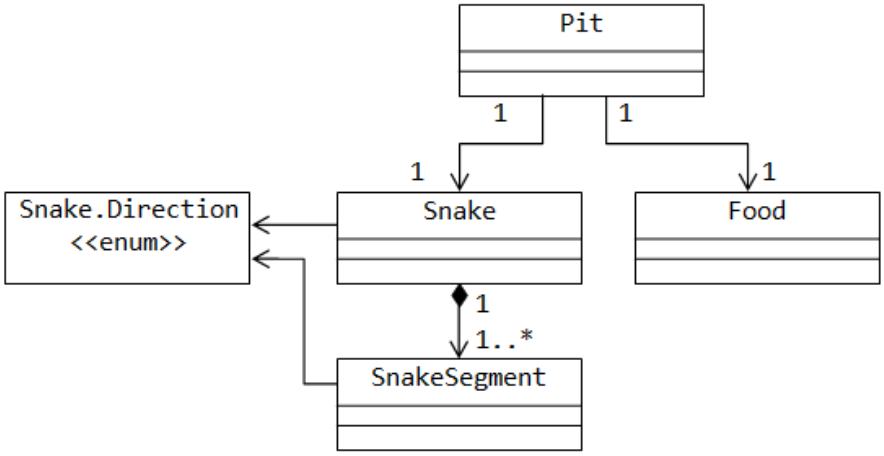
You can use the above template by:

- Declare all the instance variables for game objects.
- Instantiate all the game objects in the method `gameInit()`.
- Program the `gameUpdate()`, for calculating the new position of the game objects, updating the states, detecting the collision and providing the responses.
- Program the `gameDraw()`, for display game objects and information after each game step.
- Handle the input event in `gameKeyPressed()`. Update the state of the game.

Case Study 1: The Snake Game (Part I)

Let's design a simple snake game based on this framework. The snake move around the pit in search of food. It grows by one cell-length whenever it eats a piece of food. The snake dies when it try to eat itself, or moves outside the pit.





A `Snake` is made up of one or more horizontal or vertical `SnakeSegment`s. A `Pit` contains one `Snake` and one piece of `Food`. An enumeration called `Direction` is defined as a static nested class of `Snake` for the four moving directions: `UP`, `DOWN`, `LEFT` and `RIGHT`.

Game Actor Design

In general, a "moving" game actor should have these properties:

```
protected boolean alive;          // alive or dead
protected double x, y;            // (x, y) location of its center
protected double speed;           // speed in pixels per game-step
protected double direction;       // movement direction in degrees
protected double rotationSpeed;   // rotational speed in degrees per game-step (optional)
protected double width, height;   // width and height of this actor
```

It should possess these operations:

```
// Draw itself
public void draw(Graphics2D g2d) { ..... }
// Initialize (or re-initialize) this actor at the start of the game.
public void regenerate() { ..... }
// For collision detection
```

```
public boolean intersects(Shape shape) { ..... }  // intersects with the given shape
public boolean contains(Shape shape) { ..... }    // completely encloses the given shape
public Shape getBounds2D() { ..... }              // returns a bounding box
```

For the snake game, we can simplify the design as the snake can move left/right/up/down, one cell at each update. The pit consists of cells (of `ROWS` by `COLUMNS`). The coordinates are referenced to the cells' row and column numbers.

Enum `Snake.Direction`

We first decided to have an enum called `Direction`, to represent the movement direction of the snake. We decided to place this enum as a static nested enum inside the `Snake` class as follows:

```
public class Snake {
   public static enum Direction {
      UP, DOWN, LEFT, RIGHT
   }
   ......
   ......
}
```

Class `SnakeSegment`

Next, we decided that a snake shall make up of horizontal and vertical segments. Each segment has its head at (`headX`, `headY`), a length, and a movement direction. Segment can grow (at the head) and shrink (at the tail) by one cell. It can draw itself. It has a method called `contains(int x, int y)` for collision detection.

```
import java.awt.Graphics;
/**
 * SnakeSegment represents one horizontal or vertical segment of a snake. The "head" of
 * this segment is at (headX, headY). The segment is drawn starting from the "head"
 * and proceeding "length" cells in "direction", until it reaches the "tail".
 */
public class SnakeSegment {

   private int headX, headY;   // The position of the head of this segment
   private int length;         // length of this segment
   private Snake.Direction direction;   // direction of this segment

   // Construct a new snake segment at given (headX, headY), length and direction.
   public SnakeSegment(int headX, int headY, int length, Snake.Direction direction) {
      this.headX = headX;
      this.headY = headY;
      this.direction = direction;
      this.length = length;
   }

   // Grow by adding one cell to the head of this segment.
   public void grow() {
      ++length;
      // need to adjust the headX and headY
      switch (direction) {
         case LEFT:  --headX; break;
         case RIGHT: ++headX; break;
         case UP:    --headY; break;
         case DOWN:  ++headY; break;
      }
   }

   // Shrink by removing one cell from the tail of this segment.
   public void shrink() {
      length--;  // no change in headX and headY needed
   }

   // Get the length, in cells, of this segment.
   public int getLength() { return length; }

   // Get the X coordinate of the cell that contains the head of this snake segment.
   public int getHeadX() { return headX; }

   // Get the Y coordinate of the cell that contains the head of this snake segment.
   public int getHeadY() { return headY; }

   // Get the X coordinate of the cell that contains the tail of this snake segment.
   private int getTailX() {
      if (direction == Snake.Direction.LEFT) {
         return headX + length - 1;
      } else if (direction == Snake.Direction.RIGHT) {
         return headX - length + 1;
      } else {
         return headX;
```

```
      }
   }

   // Get the Y coordinate of the cell that contains the tail of this snake segment.
   private int getTailY() {
      if (direction == Snake.Direction.DOWN) {
         return headY - length + 1;
      } else if (direction == Snake.Direction.UP) {
         return headY + length - 1;
      } else {
         return headY;
      }
   }

   // Returns true if the snake segment contains the given cell. Used for collision detection.
   public boolean contains(int x, int y) {
      switch (direction) {
         case LEFT:  return ((y == this.headY) && ((x >= this.headX) && (x <= getTailX())));
         case RIGHT: return ((y == this.headY) && ((x <= this.headX) && (x >= getTailX())));
         case UP:    return ((x == this.headX) && ((y >= this.headY) && (y <= getTailY())));
         case DOWN:  return ((x == this.headX) && ((y <= this.headY) && (y >= getTailY())));
      }
      return false;
   }

   // Draw this segment.
   public void draw(Graphics g) {
      int x = headX;
      int y = headY;

      switch (direction) {
         case LEFT:
            for (int i = 0; i < length; ++i) {
               g.fill3DRect(x * GameMain.CELL_SIZE, y * GameMain.CELL_SIZE,
                     GameMain.CELL_SIZE - 1, GameMain.CELL_SIZE - 1, true);
               ++x;
            }
            break;
         case RIGHT:
            for (int i = 0; i < length; ++i) {
               g.fill3DRect(x * GameMain.CELL_SIZE, y * GameMain.CELL_SIZE,
                     GameMain.CELL_SIZE - 1, GameMain.CELL_SIZE - 1, true);
               --x;
            }
            break;
         case UP:
            for (int i = 0; i < length; ++i) {
               g.fill3DRect(x * GameMain.CELL_SIZE, y * GameMain.CELL_SIZE,
                     GameMain.CELL_SIZE - 1, GameMain.CELL_SIZE - 1, true);
               ++y;
            }
            break;
         case DOWN:
            for (int i = 0; i < length; ++i) {
               g.fill3DRect(x * GameMain.CELL_SIZE, y * GameMain.CELL_SIZE,
                     GameMain.CELL_SIZE - 1, GameMain.CELL_SIZE - 1, true);
               --y;
            }
            break;
      }
   }

   // For debugging.
   @Override
   public String toString() {
      return "Head at (" + headX + "," + headY + ")" + " to (" + getTailX() + ","
            + getTailY() + ")" + ", length is " + getLength() + ", dir is " + direction;
   }
}
```

Class `Snake`

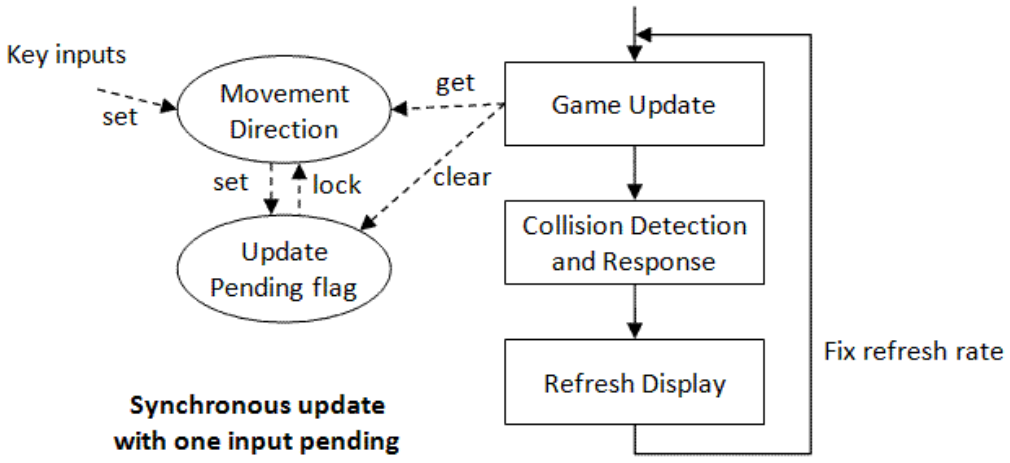
Next, the `Snake` class is designed to maintain a list of `SnakeSegments`.

```
import java.awt.*;
import java.util.*;
/**
 * A Snake is made up of one or more SnakeSegment. The first SnakeSegment is the
 * "head" of the snake. The last SnakeSegment is the "tail" of the snake. As the
 * snake moves, it adds one cell to the head and then removes one from the tail. If
 * the snake eats a piece of food, the head adds one cell but the tail will not
 * shrink.
```

```
 */
public class Snake {

   public static enum Direction {
      UP, DOWN, LEFT, RIGHT
   }

   private Color color = Color.BLACK;     // color for this snake body
   private Color colorHead = Color.GREEN; // color for the "head"
   private Snake.Direction direction;     // the current direction of the snake's head

   // The snake segments that forms the snake
   private java.util.List<SnakeSegment> snakeSegments = new ArrayList<SnakeSegment>();

   private boolean dirUpdatePending;   // Pending update for a direction change?
   private Random random = new Random();   // for randomly regenerating a snake

   // Regenerate the snake.
   public void regenerate() {
      snakeSegments.clear();
      // Randomly generate a snake inside the pit.
      int length = 20;
      int headX = random.nextInt(GameMain.COLUMNS - length * 2) + length;
      int headY = random.nextInt(GameMain.ROWS - length * 2) + length;
      direction = Snake.Direction.values()[random.nextInt(Snake.Direction.values().length)];
      snakeSegments.add(new SnakeSegment(headX, headY, length, direction));
      dirUpdatePending = false;
   }

   // Change the direction of the snake, but no 180 degree turn allowed.
   public void setDirection(Snake.Direction newDir) {
      // Ignore if there is a direction change pending and no 180 degree turn
      if (!dirUpdatePending && (newDir != direction)
            && ((newDir == Snake.Direction.UP && direction != Snake.Direction.DOWN)
             || (newDir == Snake.Direction.DOWN && direction != Snake.Direction.UP)
             || (newDir == Snake.Direction.LEFT && direction != Snake.Direction.RIGHT)
             || (newDir == Snake.Direction.RIGHT && direction != Snake.Direction.LEFT))) {
         SnakeSegment headSegment = snakeSegments.get(0);  // get the head segment
         int x = headSegment.getHeadX();
         int y = headSegment.getHeadY();
         // add a new segment with zero length as the new head segment
         snakeSegments.add(0, new SnakeSegment(x, y, 0, newDir));
         direction = newDir;
         dirUpdatePending = true; // will be cleared after updated
      }
   }

   // Move the snake by one step. The snake "head" segment grows by one cell. The rest of the
   // segments remain unchanged. The "tail" segment will later be shrink if collision detected.
   public void update() {
      SnakeSegment headSegment;
      headSegment = snakeSegments.get(0);   // "head" segment
      headSegment.grow();
      dirUpdatePending = false;   // can process the key input again
   }

   // Not eaten a food item. Shrink the tail by one cell.
   public void shrink() {
      SnakeSegment tailSegment;
      tailSegment = snakeSegments.get(snakeSegments.size() - 1);
      tailSegment.shrink();
      if (tailSegment.getLength() == 0) {
         snakeSegments.remove(tailSegment);
      }
   }

   // Get the X coordinate of the cell that contains the head of this snake segment.
   public int getHeadX() {
      return snakeSegments.get(0).getHeadX();
   }

   // Get the Y coordinate of the cell that contains the head of this snake segment.
   public int getHeadY() {
      return snakeSegments.get(0).getHeadY();
   }

   // Returns the length of this snake by adding up all the segments.
   public int getLength() {
      int length = 0;
      for (SnakeSegment segment : snakeSegments) {
         length += segment.getLength();
      }
      return length;
```

```
    }

    // Returns true if the snake contains the given (x, y) cell. Used in collision detection
    public boolean contains(int x, int y) {
       for (int i = 0; i < snakeSegments.size(); ++i) {
          SnakeSegment segment = snakeSegments.get(i);
          if (segment.contains(x, y)) {
             return true;
          }
       }
       return false;
    }

    // Returns true if the snake eats itself
    public boolean eatItself() {
       int headX = getHeadX();
       int headY = getHeadY();
       // eat itself if the (headX, headY) hits its body segment (4th onwards)
       for (int i = 3; i < snakeSegments.size(); ++i) {
          SnakeSegment segment = snakeSegments.get(i);
          if (segment.contains(headX, headY)) {
             return true;
          }
       }
       return false;
    }

    // Draw itself.
    public void draw(Graphics g) {
       g.setColor(color);
       for (int i = 0; i < snakeSegments.size(); ++i) {
          snakeSegments.get(i).draw(g);   // draw all the segments
       }
       if (snakeSegments.size() > 0) {
          g.setColor(colorHead);
          g.fill3DRect(getHeadX() * GameMain.CELL_SIZE, getHeadY()
                * GameMain.CELL_SIZE, GameMain.CELL_SIZE - 1,
                GameMain.CELL_SIZE - 1, true);
       }
    }

    // For debugging.
    @Override
    public String toString() {
       StringBuffer sb = new StringBuffer();
       sb.append("**Snake** Direction is " + direction + "\n");
       int count = 1;
       for (SnakeSegment segment : snakeSegments) {
          sb.append("  Segment " + count + ": ");
          ++count;
          sb.append(segment);
          sb.append('\n');
       }
       return sb.toString();
    }
 }
```

In order to process only the first key input, and ignore the rest until the first key is processed, a flag called `dirUpdatePending` is used, which will be set whenever a change of direction input is received. This stops further changes, until the snake is updated.



[PENDING] more explanation

Class `Food`

Next, the Food class, which is rather straight forward.

```
import java.awt.*;
import java.util.*;
/**
 * Food is a food item that the snake can eat. It is placed randomly in the pit.
 */
public class Food {

   private int x, y;   // current food location (x, y) in cells
   private Color color = Color.BLUE;   // color for display
   private Random rand = new Random(); // For randomly placing the food

   // Default constructor.
   public Food() {
      // place outside the pit, so that it will not be "displayed".
      x = -1;
      y = -1;
   }

   // Regenerate a food item. Randomly place inside the pit (slightly off the edge).
   public void regenerate() {
      x = rand.nextInt(GameMain.COLUMNS - 4) + 2;
      y = rand.nextInt(GameMain.ROWS - 4) + 2;
   }

   // Returns the x coordinate of the cell that contains this food item.
   public int getX() { return x; }

   // Returns the y coordinate of the cell that contains this food item.
   public int getY() { return y; }

   // Draw itself.
   public void draw(Graphics g) {
      g.setColor(color);
      g.fill3DRect(x * GameMain.CELL_SIZE, y * GameMain.CELL_SIZE,
            GameMain.CELL_SIZE, GameMain.CELL_SIZE, true);
   }
}
```

Class `GameMain`

Finally, it is the main class `GameMain`, based on the framework.

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

public class GameMain extends JFrame {   // main class for the game as a Swing application

   // Define constants for the game
   static final int ROWS = 50;      // number of rows (in cells)
   static final int COLUMNS = 50;   // number of columns (in cells)
   static final int CELL_SIZE = 12; // Size of a cell (in pixels)
   static final int CANVAS_WIDTH  = COLUMNS * CELL_SIZE;  // width and height of the game screen
   static final int CANVAS_HEIGHT = ROWS * CELL_SIZE;
   static final int UPDATE_RATE = 3;    // number of game update per second
   static final long UPDATE_PERIOD = 1000000000L / UPDATE_RATE;  // nanoseconds
   private final Color COLOR_PIT = Color.LIGHT_GRAY;

   // Enumeration for the states of the game.
   static enum State {
      INITIALIZED, PLAYING, PAUSED, GAMEOVER, DESTROYED
   }
   static State state;   // current state of the game

   // Define instance variables for the game objects
   private Food food;
   private Snake snake;

   // Handle for the custom drawing panel
   private GameCanvas pit;

   // Constructor to initialize the UI components and game objects
   public GameMain() {
      // Initialize the game objects
      gameInit();
```

```
      // UI components
      pit = new GameCanvas();
      pit.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
      this.setContentPane(pit);
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.pack();
      this.setTitle("SNAKE GAME");
      this.setVisible(true);

      // Start the game.
      gameStart();
   }

   // ------ All the game related codes here ------

   // Initialize all the game objects, run only once in the constructor of the main class.
   public void gameInit() {
      // Allocate a new snake and a food item, do not regenerate.
      snake = new Snake();
      food = new Food();
      state = State.INITIALIZED;
   }

   // Shutdown the game, clean up code that runs only once.
   public void gameShutdown() { }

   // To start and re-start the game.
   public void gameStart() {
      // Create a new thread
      Thread gameThread =  new Thread() {
         // Override run() to provide the running behavior of this thread.
         @Override
         public void run() {
            gameLoop();
         }
      };
      // Start the thread. start() calls run(), which in turn calls gameLoop().
      gameThread.start();
   }

   // Run the game loop here.
   private void gameLoop() {
      // Regenerate and reset the game objects for a new game
      if (state == State.INITIALIZED || state == State.GAMEOVER) {
         // Generate a new snake and a food item
         snake.regenerate();
         int x, y;
         do {
            food.regenerate();
            x = food.getX();
            y = food.getY();
         } while (snake.contains(x, y)); // regenerate if food placed under the snake
         state = State.PLAYING;
      }

      // Game loop
      long beginTime, timeTaken, timeLeft;
      while (true) {
         beginTime = System.nanoTime();
         if (state == State.GAMEOVER) break;  // break the loop to finish the current play
         if (state == State.PLAYING) {
            // Update the state and position of all the game objects,
            // detect collisions and provide responses.
            gameUpdate();
         }
         // Refresh the display
         repaint();
         // Delay timer to provide the necessary delay to meet the target rate
         timeTaken = System.nanoTime() - beginTime;
         timeLeft = (UPDATE_PERIOD - timeTaken) / 1000000;  // in milliseconds
         if (timeLeft < 10) timeLeft = 10;  // set a minimum
         try {
            // Provides the necessary delay and also yields control so that other thread can do work.
            Thread.sleep(timeLeft);
         } catch (InterruptedException ex) { }
      }
   }

   // Update the state and position of all the game objects,
   // detect collisions and provide responses.
   public void gameUpdate() {
      snake.update();
      processCollision();
```

```
   }

   // Collision detection and response
   public void processCollision() {
      // check if this snake eats the food item
      int headX = snake.getHeadX();
      int headY = snake.getHeadY();

      if (headX == food.getX() && headY == food.getY()) {
         // food eaten, regenerate one
         int x, y;
         do {
            food.regenerate();
            x = food.getX();
            y = food.getY();
         } while (snake.contains(x, y));
      } else {
         // not eaten, shrink the tail
         snake.shrink();
      }

      // Check if the snake moves out of bounds
      if (!pit.contains(headX, headY)) {
         state = State.GAMEOVER;
         return;
      }

      // Check if the snake eats itself
      if (snake.eatItself()) {
         state = State.GAMEOVER;
         return;
      }
   }

   // Refresh the display. Called back via rapaint(), which invoke the paintComponent().
   private void gameDraw(Graphics g) {
      switch (state) {
         case PLAYING:
            // draw game objects
            snake.draw(g);
            food.draw(g);
            // game info
            g.setFont(new Font("Dialog", Font.PLAIN, 14));
            g.setColor(Color.BLACK);
            g.drawString("Snake: (" + snake.getHeadX() + "," + snake.getHeadY() + ")", 5, 25);
            break;
         case GAMEOVER:
            g.setFont(new Font("Verdana", Font.BOLD, 30));
            g.setColor(Color.RED);
            g.drawString("GAME OVER!", 200, CANVAS_HEIGHT / 2);
            break;
      }
   }

   // Process a key-pressed event. Update the current state.
   public void gameKeyPressed(int keyCode) {
      switch (keyCode) {
         case KeyEvent.VK_UP:
            snake.setDirection(Snake.Direction.UP);
            break;
         case KeyEvent.VK_DOWN:
            snake.setDirection(Snake.Direction.DOWN);
            break;
         case KeyEvent.VK_LEFT:
            snake.setDirection(Snake.Direction.LEFT);
            break;
         case KeyEvent.VK_RIGHT:
            snake.setDirection(Snake.Direction.RIGHT);
            break;
      }
   }

   // Custom drawing panel, written as an inner class.
   class GameCanvas extends JPanel implements KeyListener {
      // Constructor
      public GameCanvas() {
         setFocusable(true);  // so that can receive key-events
         requestFocus();
         addKeyListener(this);
      }

      // Override paintComponent to do custom drawing.
      // Called back by repaint().
```

```
      @Override
      public void paintComponent(Graphics g) {
         super.paintComponent(g);   // paint background
         setBackground(COLOR_PIT);  // may use an image for background

         // Draw the game objects
         gameDraw(g);
      }

      // KeyEvent handlers
      @Override
      public void keyPressed(KeyEvent e) {
         gameKeyPressed(e.getKeyCode());
      }

      @Override
      public void keyReleased(KeyEvent e) { }

      @Override
      public void keyTyped(KeyEvent e) { }

      // Check if this pit contains the given (x, y), for collision detection
      public boolean contains(int x, int y) {
         if ((x < 0) || (x >= ROWS)) {
            return false;
         }
         if ((y < 0) || (y >= COLUMNS)) {
            return false;
         }
         return true;
      }
   }

   // main
   public static void main(String[] args) {
      // Use the event dispatch thread to build the UI for thread-safety.
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            new GameMain();
         }
      });
   }
}
```

- An instance of `GameCanvas` called `pit` is used. A method `contains(x, y)` is added in the `GameCanvas` for collision detection.
- [PENDING]

Collision Detection & Response

The hardest thing to decide is where (which class) to place the collision detection and responses codes. Should they be separated? How to add in these codes and yet the actors are still well encapsulated? If other words, one actor should not know the existence of the other actors.

I recommend that:

- Separate the detection and response codes. The game loop shall carry out these steps:
  - All sprites move to the next position (i.e., move one step).
  - Check for collision in these new positions, and update the state of the sprites.
  - Based on the new state, perform collision response.
  - Repeat the loop.
- For simple game, you may combine detection (in step 2) and response (in step 3) together. But take note that all the actors should move into the next position before carry out the collision detection.
- The collision detection and responses code should be kept in the main game class, instead of within the actor. To support this, the main class must be able to retrieve certain information from the actor. Hence, it is proposed that each of the actor should have the following methods. The main class can use these methods of the two sprites to detect collision between them. (What to return? Shape and Rectangle2D?)

```
public boolean intersects(Shape shape) { ..... }  // intersects with the given shape
public boolean contains(Shape shape) { ..... }    // completely encloses the given shape
public Shape getBounds2D() { ..... }              // returns a bounding box
```

```
for each pair of actors, actor1.checkCollision(actor2).
for each actor, responseUponCollision().
```

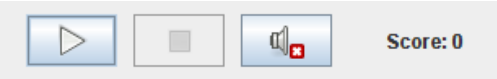
- Actor shall maintain its own state, via an associated enumeration called `State`.

[TODO] Shall I move, check collision, and unmove if not collided; or check move, move (no unmove)?

Snake Game - Part II

Let zest up the snake game by adding a control panel, score board, and sound effects.

Control Panel




```
public class GameMain extends JFrame {

   ......
   ......
   // Handle for the custom drawing panel and UI components.
   private GameCanvas pit;
   private ControlPanel control;
   private JLabel lblScore;

   // Constructor to initialize the UI components and game objects
   public GameMain() {
      // Initialize the game objects
      gameInit();

      // UI components
      Container cp = this.getContentPane();
      cp.setLayout(new BorderLayout());
      pit = new GameCanvas();               // drawing panel
      pit.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
      cp.add(pit);
      cp.add(pit, BorderLayout.CENTER);
      control = new ControlPanel();   // control panel
      cp.add(control, BorderLayout.SOUTH);

      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.pack();
      this.setTitle("SNAKE GAME");
      this.setVisible(true);
   }

   // Game Control Panel with Start, Stop, Pause and Mute buttons, designed as an inner class.
   class ControlPanel extends JPanel {
      private JButton btnStartPause;
      private JButton btnStop;
      private JButton btnMute;

      private ImageIcon iconStart = new ImageIcon(getClass().getResource("media-playback-start.png"), "START");
      private ImageIcon iconPause = new ImageIcon(getClass().getResource("media-playback-pause.png"), "PAUSE");
      private ImageIcon iconStop = new ImageIcon(getClass().getResource("media-playback-stop.png"), "STOP");
      private ImageIcon iconSound = new ImageIcon(getClass().getResource("audio-volume-high.png"), "SOUND ON");
      private ImageIcon iconMuted = new ImageIcon(getClass().getResource("audio-volume-muted.png"), "MUTED");

      public ControlPanel() {
         this.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));
         btnStartPause = new JButton(iconStart);
         btnStartPause.setEnabled(true);
         add(btnStartPause);
         btnStop = new JButton(iconStop);
         btnStop.setEnabled(false);
         add(btnStop);
         btnMute = new JButton(iconMuted);
         btnMute.setEnabled(true);
         add(btnMute);
         lblScore = new JLabel("        Score: " + score);
         add(lblScore);

         btnStartPause.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               switch (state) {
                  case INITIALIZED:
                  case GAMEOVER:
                     btnStartPause.setIcon(iconPause);
                     gameStart();
                     break;
                  case PLAYING:
                     state = State.PAUSED;
                     btnStartPause.setIcon(iconStart);
                     break;
                  case PAUSED:
                     state = State.PLAYING;
                     btnStartPause.setIcon(iconPause);
                     break;
```

```
                }
                btnStop.setEnabled(true);
                pit.requestFocus();
             }
          });
          btnStop.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
                state = State.GAMEOVER;
                btnStartPause.setIcon(iconStart);
                btnStartPause.setEnabled(true);
                btnStop.setEnabled(false);
             }
          });
          btnMute.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
                if (SoundEffect.volume == SoundEffect.Volume.MUTE) {
                   SoundEffect.volume = SoundEffect.Volume.LOW;
                   btnMute.setIcon(iconSound);
                   pit.requestFocus();
                } else {
                   SoundEffect.volume = SoundEffect.Volume.MUTE;
                   btnMute.setIcon(iconMuted);
                   pit.requestFocus();
                }
             }
          });
       }

       // Reset control for a new game
       public void reset() {
          btnStartPause.setIcon(iconStart);
          btnStartPause.setEnabled(true);
          btnStop.setEnabled(false);
       }
    }
}
```

- The control panel is derived from `JPanel` and is designed as an inner class, just like the drawing panel.
- There are four `JButton` inside the panel. Instead of text-label buttons, image-icons are used. The icons are downloaded from the open-source "Tango desktop project @ http://tango.freedesktop.org". For the "mute" button, two image-icons are required (mute and volume-on).
- You need to enable/disable the button accordingly, and transfer the focus to the drawing panel, after a button is pushed.

Playing Sound Effect

Include the `SoundEffect` enum, descried in the earlier chapter. Define the names of the sound effects and their associated wave file. You can then play the sound effect in the game logic.

```
public enum SoundEffect {
   DIE("die.wav"),     // game over
   EAT("eatfood.wav"); // eat an food item

   ......
   ......
}
```

Two Snakes

Try:

- Each snake shall maintain its own set of keys for UP, DOWN, LEFT and RIGHT turn, and other attributes such as color.
- The snake cannot climb over another. If one snake's movement is blocked by another, it halts, until its direction is changed via a proper key input.
- Each snake has a "heart" (3rd cells from its head). It will be killed if it is hit at its heart.

REFERENCES & RESOURCES

- Source codes of the "Sun Java Wireless Toolkit (WTK) 2.5 Demo: Worm Game".
- Jonathan S. Harbour, "Beginning Java 5 Game Programming": Good introductory book with sufficient technical details to get you started. Nice coverage on 2D vector graphics and bitmap including sprite animation. Easy to read and can be finished in a few days. However, the games are written in applet. No coverage of 3D technologies such as JOGL, JOAL and Java 3D. Also does not take full advantage of Java 5, in particular enumeration for keeping game and sprite states. No coverage on full-screen API, and performance for advanced game.
- Dustin Clingman, Shawn Kendall and Syrus Mesdaghi, "Practical Java Game Programming": filled with more advanced technical details.

Topics such as IO operations and performance are applicable even for non-gaming applications. Much harder to read. Cover 2D and 3D (JOGL, JOAL, Java 3D).

Latest version tested: JDK 1.6
Last modified: September 4, 2008