Writing classic “Snake” in Java

Snake was a game made famous by the Nokia mobile phones, back in the 1990s. The object of the game is simple: to last as long as possible by controlling the snake to eat as many apples as possible before crashing in the edges of the game area or colliding with the snake’s tail. Each apple makes the snake a little longer.

In Java, we’re going to write this game with two classes, one large, one small.

# Snake

The first class, Snake, is the entry point for the game. This class creates a JFrame object, which is the window on the screen. The logic to run the game is in the runGame() method.

**package** elycc;

**import** java.awt.EventQueue;

**import** java.nio.file.Path;

**import** java.nio.file.Paths;

**import** javax.swing.JFrame;

/\*\*

\* Main game launcher.

\*/

**public** **class** Snake {

/\*\*

\* Add the board to this frame, set the title and configure default

\* close behaviour.

\*/

**public** **void** runGame() {

JFrame frame = **new** JFrame("Snake");

frame.add(**new** Board());

frame.setResizable(**false**);

frame.pack();

frame.setTitle("Snake");

frame.setLocationRelativeTo(**null**);

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setVisible(**true**);

}

/\*\*

\* Entry point for the game.

\* **@param** args command line parameters (unused).

\*/

**public** **static** **void** main(**final** String[] args) {

Path currentRelativePath = Paths.*get*("");

String s = currentRelativePath.toAbsolutePath().toString();

System.***out***.println("Current relative path is: " + s);

EventQueue.*invokeLater*(**new** Runnable() {

@Override

**public** **void** run() {

Snake snake = **new** Snake();

snake.runGame();

}

});

}

}

Note that the frame.add(new Board()) line won’t compile yet, because Java doesn’t have a definition for the Board class, so it can’t create an object that is an instance of the Board class.

So, create the Board class as shown below.

**package** elycc;

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Graphics;

**import** java.awt.Image;

**import** java.awt.Toolkit;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.ImageIcon;

**import** javax.swing.JPanel;

**import** javax.swing.Timer;

/\*\*

\* The main logic of the game.

\*/

**public** **class** Board **extends** JPanel **implements** ActionListener {

**private** **final** **int** cellSize = 50;

**private** **final** **int** boardWidth = 30;

**private** **final** **int** boardHeight = 30;

**private** **final** **int** boardWidthPixels = boardWidth \* cellSize;

**private** **final** **int** boardHeightPixels = boardWidth \* cellSize;

**private** **final** **int** snakeSpeed = 140;

**private** **final** **int** totalBoardCells = boardWidth \* boardHeight;

**private** **final** **int**[] snakeX = **new** **int**[totalBoardCells];

**private** **final** **int**[] snakeY = **new** **int**[totalBoardCells];

**private** Timer timer;

**private** **int** snakeLength;

**private** **int** appleX;

**private** **int** appleY;

**private** Image ball;

**private** Image apple;

**private** Image head;

/\*\*

\* Sets up everything that the game needs to run.

\*/

**public** Board() {

setBackground(Color.***black***);

setFocusable(**true**);

setPreferredSize(**new** Dimension(boardWidthPixels, boardHeightPixels));

loadImages();

initGame();

}

/\*\*

\* Loads an image from disk and scales it to the right size for the game.

\*

\* **@param** name

\* the name of the image to load.

\* **@return** a square image of the {@link #cellSize}.

\*/

**private** Image loadImage(**final** String name) {

ImageIcon iid = **new** ImageIcon(name);

Image image = iid.getImage();

**return** image.getScaledInstance(cellSize, cellSize, Image.***SCALE\_DEFAULT***);

}

/\*\*

\* Load all of the images required by the game.

\*/

**private** **void** loadImages() {

ball = loadImage("dot.png");

apple = loadImage("apple.png");

head = loadImage("head.png");

}

/\*\*

\* Initialise the game by resetting the length and position of the snake,

\* adding an apple at a random position and kick off the timer that runs the

\* main game loop.

\*/

**private** **void** initGame() {

// Starting length of the snake

snakeLength = 3;

// Set the X & Y coordinates of the snakes 3 segment body

snakeX[0] = 5;

snakeY[0] = 5;

snakeX[1] = 4;

snakeY[1] = 5;

snakeX[2] = 3;

snakeY[2] = 5;

placeApple();

// Set a timer to call

timer = **new** Timer(snakeSpeed, **this**);

timer.start();

}

/\*\*

\* Called when Java decides that the window should be repainted.

\*/

@Override

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

drawGraphics(g);

}

/\*\*

\* Called from {@link Board#paintComponent(Graphics)} to draw the game or

\* game over screen.

\*

\* **@param** g

\* the graphic context to draw the game or game over screen on.

\*/

**private** **void** drawGraphics(Graphics g) {

// Draw the apple

g.drawImage(apple, appleX \* cellSize, appleY \* cellSize, **this**);

// Draw the head of the snake

g.drawImage(head, snakeX[0] \* cellSize, snakeY[0] \* cellSize, **this**);

// Draw the body of the snake

**for** (**int** z = 1; z < snakeLength; z++) {

g.drawImage(ball, snakeX[z] \* cellSize, snakeY[z] \* cellSize, **this**);

}

Toolkit.*getDefaultToolkit*().sync();

}

/\*\*

\* Places an apple on the board at a random position.

\*/

**private** **void** placeApple() {

**int** r = (**int**) (Math.*random*() \* boardWidth);

appleX = r;

r = (**int**) (Math.*random*() \* boardHeight);

appleY = r;

}

@Override

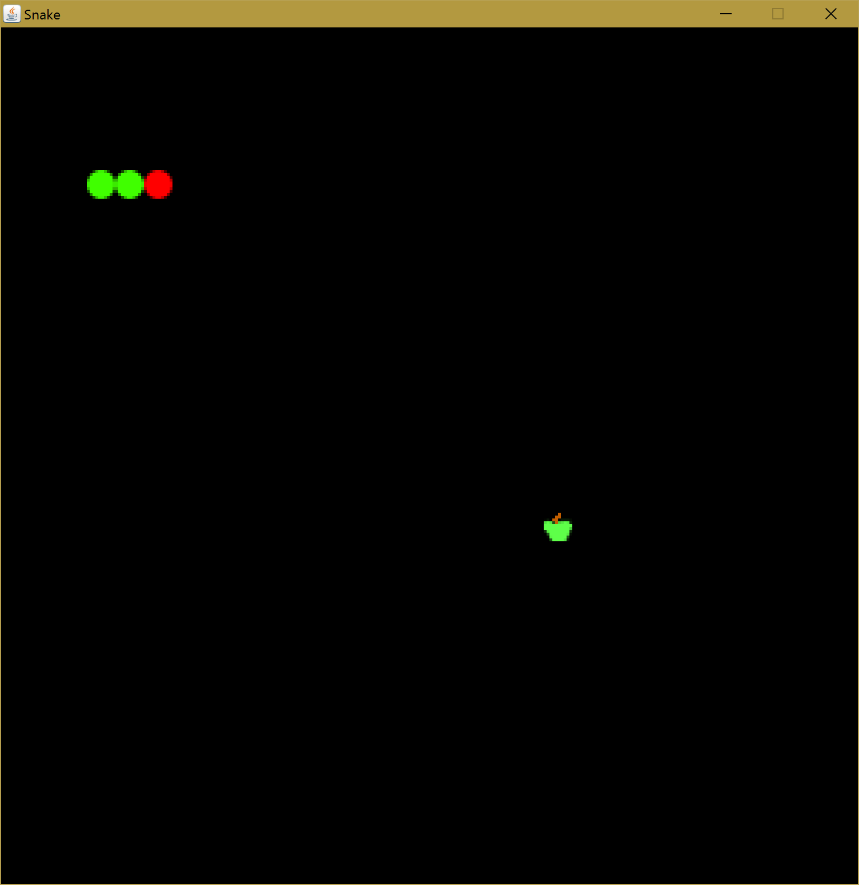
**public** **void** actionPerformed(**final** ActionEvent e) {

repaint();

}

}

When run, this code should render the starting position of the snake and an apple, like this:



The next step is to make the snake move, respond to keypresses to change direction and, of course, check collisions with the apple and the side of the grid.

# Making the Snake Move

To make the snake move we need to know what direction the snake is moving in. Define the following constants and something to store the current direction in.

/\* The different directions that the snake can go \*/

**private** **static** **int** *UP* = 0;

**private** **static** **int** *RIGHT* = 1;

**private** **static** **int** *DOWN* = 2;

**private** **static** **int** *LEFT* = 3;

/\* The current direction \*/

**private** **int** direction = *RIGHT*;

Also, add a flag to record that we’re currently in the game (this will turn false at the end of the game):

/\*\* Are we in the middle of a game or not? \*\*/

**private** **boolean** inGame = **true**;

Add a new method to move the snake.

/\*\*

\* Move the snake 1 cell in the current direction.

\*/

**private** **void** move() {

**for** (**int** z = snakeLength; z > 0; z--) {

snakeX[z] = snakeX[(z - 1)];

snakeY[z] = snakeY[(z - 1)];

}

**if** (direction==*LEFT*) {

snakeX[0] -= 1;

}

**if** (direction==*RIGHT*) {

snakeX[0] += 1;

}

**if** (direction==*UP*) {

snakeY[0] -= 1;

}

**if** (direction==*DOWN*) {

snakeY[0] += 1;

}

}

Then add a call to move() inside the timer action method defined before:

@Override

**public** **void** actionPerformed(**final** ActionEvent e) {

**if** (inGame) {

move();

}

repaint();

}

Run the code and the snake should move across the screen.

# Collision Detection

We need to know when the snake has hit the edge of the screen, or itself.

/\*\*

\* Checks to see if the snake has hit its own tail or the edges of the

\* board.

\*/

**private** **void** checkCollision() {

// Check to see if the snake has hit its own tail

**for** (**int** z = snakeLength; z > 0; z--) {

**if** ((z > 4) && (snakeX[0] == snakeX[z]) && (snakeY[0] == snakeY[z])) {

inGame = **false**;

}

}

// Make sure the snake hasn't hit the edges of the board

**if** ((snakeY[0] >= boardHeight) || (snakeY[0] < 0) ||

(snakeX[0] >= boardWidth) || (snakeX[0] < 0)) {

inGame = **false**;

}

**if** (!inGame) {

timer.stop();

}

}

Call this in the main loop, alongside move().

checkCollision();

# Controlling the Snake

We need to change the direction of the snake when we the cursor keys on the keyboard.

/\*\*

\* Event handler for key presses.

\*/

**private** **class** TAdapter **extends** KeyAdapter {

@Override

**public** **void** keyPressed(**final** KeyEvent e) {

**int** key = e.getKeyCode();

**if** (key == KeyEvent.***VK\_LEFT***) {

direction = *LEFT*;

}

**if** (key == KeyEvent.***VK\_RIGHT***) {

direction = *RIGHT*;

}

**if** (key == KeyEvent.***VK\_UP***) {

direction = *UP*;

}

**if** (key == KeyEvent.***VK\_DOWN***) {

direction = *DOWN*;

}

}

}

And add it to the constructor (the method named Board()) at the top:

addKeyListener(**new** TAdapter());

You can now control the snake, move it around the play area, however if you go over the apple, it doesn’t eat it. That’s because we haven’t written the collision detection routine for the head of the snake occupying the same co-ordinates as the apple.

/\*\*

\* Check to see if the head of the snake is on top of the apple, if so then

\* make the snake longer, increment the score and put a new apple in a

\* random location.

\*/

**private** **void** checkApple() {

**if** ((snakeX[0] == appleX) && (snakeY[0] == appleY)) {

snakeLength++;

placeApple();

}

}

And call this in the main loop, alongside checkCollision.

checkApple();

You’ve now got a fully working snake game!

# Keeping Score

To keep score (how many apples the snake ate), add a new variable (an int) called score, then increment score each time an apple is eaten. You can work out when to do this yourself.

Finally, add a new method to show the score at the end:

/\*\*

\* When the game is over, print out a score and a game over screen.

\*

\* **@param** g

\* the graphic context to draw information on.

\*/

**private** **void** gameOver(Graphics g) {

String msg = "Game Over! Score: " + score;

Font small = **new** Font("Helvetica", Font.***BOLD***, 14);

FontMetrics metr = getFontMetrics(small);

g.setColor(Color.***white***);

g.setFont(small);

g.drawString(msg, (boardWidthPixels - metr.stringWidth(msg)) / 2, boardHeightPixels / 2);

}

And change drawGraphics(…) to call this new method:

**private** **void** drawGraphics(Graphics g) {

**if (inGame) {**

g.drawImage(apple, appleX \* cellSize, appleY \* cellSize, **this**);

g.drawImage(head, snakeX[0] \* cellSize, snakeY[0] \* cellSize, **this**);

**for** (**int** z = 1; z < snakeLength; z++) {

g.drawImage(ball, snakeX[z] \* cellSize, snakeY[z] \* cellSize, **this**);

}

Toolkit.*getDefaultToolkit*().sync();

**} else {**

**gameOver(g);**

**}**

}

The changes are shown in bold.

# Further Enhancement Ideas

Ideas for further enhancements:

1. Be able to restart the game by pressing space after game oven, rather than having to restart the program.
2. Show the score all the time during gameplay.
3. Add another type of fruit to the board that gets adds 2 points each time you get it.
4. Make the game run a little faster each time you collect an apple.