//SnakeGame. java

```
import java.awt.BorderLayout;
import java.awt.Point;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.LinkedList;
import java.util.Random;

import javax.swing.JFrame;

public class SnakeGame extends JFrame
{
private static final long FRAME_TIME = 1000L / 50L;

/**
 * The minimum length of the snake. This allows the snake to grow
 * right when the game starts, so that we're not just a head moving
 * around on the board.
 */
private static final int MIN_SNAKE_LENGTH = 5;
/**

* The maximum number of directions that we can have polled in the
* direction list.
*/
private static final int MAX_DIRECTIONS = 3;
private BoardPanel board;
private SidePanel side;
/**
 * The random number generator (used for spawning fruits).
 */
private Random random;
/**
* The Clock instance for handling the game logic.
```

```
 */
private Clock logicTimer;
private booleanisNewGame;
private booleanisGameOver;
private booleanisPaused;
private LinkedList<Point> snake;
private LinkedList<Direction> directions;
private int score;
private intfruitsEaten;
private intnextFruitScore;
/**

 * Creates a new SnakeGame instance. Creates a new window,
 * and sets up the controller input.
 */
private SnakeGame()
 {
super("Snake Game");
setLayout(new BorderLayout());
setDefaultCloseOperation(EXIT_ON_CLOSE);
setResizable(false);
/*
 * Initialize the game's panels and add them to the window.
 */
this.board = new BoardPanel(this);
this.side = new SidePanel(this);
add(board, BorderLayout.CENTER);
add(side, BorderLayout.EAST);
/*
 * Adds a new key listener to the frame to process input.
 */
addKeyListener(new KeyAdapter()
 {
@Override
```

```
public void keyPressed(KeyEvent e)

{
switch(e.getKeyCode())
{
/*
 * If the game is not paused, and the game is not over...
  * Ensure that the direction list is not full, and that the most
 * recent direction is adjacent to North before adding the
* direction to the list.
 */
case KeyEvent.VK_W:
case KeyEvent.VK_UP:
if(!isPaused&& !isGameOver)
{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.South&& last != Direction.North)
 {
directions.addLast(Direction.North);
}
}
}
break;

case KeyEvent.VK_S:
case KeyEvent.VK_DOWN:
if(!isPaused&& !isGameOver)
{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.North&& last != Direction.South)
```

```
{
directions.addLast(Direction.South);
}
}
}
break;

case KeyEvent.VK_A:
case KeyEvent.VK_LEFT:
if(!isPaused&& !isGameOver)
{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.East&& last != Direction.West)

{
directions.addLast(Direction.West);
}
}
}
break;

case KeyEvent.VK_D:
case KeyEvent.VK_RIGHT:
if(!isPaused&& !isGameOver)
{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.West&& last != Direction.East)
{
directions.addLast(Direction.East);
}
```

```
}
}
break;
/*
* If the game is not over, toggle the paused flag and update

* the logicTimer's pause flag accordingly.
 */
case KeyEvent.VK_P:
if(!isGameOver)
{
isPaused = !isPaused;
logicTimer.setPaused(isPaused);
}
break;
/*
* Reset the game if one is not currently in progress.
*/
case KeyEvent.VK_ENTER:
if(isNewGame || isGameOver)
{
resetGame();
}
break;
}
}
});
/*
* Resize the window to the appropriate size, center it on the

* screen and display it.
*/
pack();
setLocationRelativeTo(null);
```

```
setVisible(true);
}
/**
 * Starts the game running.
 */
private void startGame()
{
/*
* Initialize everything we're going to be using.
 */
this.random = new Random();
this.snake = new LinkedList<>();
this.directions = new LinkedList<>();
this.logicTimer = new Clock(9.0f);
this.isNewGame = true;
//Set the timer to paused initially.
logicTimer.setPaused(true);
/*
* This is the game loop. It will update and render the game and will

* continue to run until the game window is closed.
*/
while(true)
{
//Get the current frame's start time.
long start = System.nanoTime();
//Update the logic timer.
logicTimer.update();
if(logicTimer.hasElapsedCycle())
{
updateGame();
}
//Repaint the board and side panel with the new content.
board.repaint();
```

```
side.repaint();
/*
* Calculate the delta time between since the start of the frame
 * and sleep for the excess time to cap the frame rate. While not
* incredibly accurate, it is sufficient for our purposes.
 */
long delta = (System.nanoTime() - start) / 1000000L;
if(delta < FRAME_TIME)
{

try
{
Thread.sleep(FRAME_TIME - delta);
}
 catch(Exception e)
{
e.printStackTrace();
}
}
}
}
/**
 * Updates the game's logic.
 */
private void updateGame()
{
/*
 * Gets the type of tile that the head of the snake collided with. If
* the snake hit a wall, SnakeBody will be returned, as both conditions
* are handled identically.
*/
TileType collision = updateSnake();
```

```
if(collision == TileType.Fruit)
{
fruitsEaten++;
score += nextFruitScore;
spawnFruit();
}
 else if(collision == TileType.SnakeBody)
{
isGameOver = true;
logicTimer.setPaused(true);
}
else if(nextFruitScore> 10)
{
nextFruitScore--;
}
}
/**
 * Updates the snake's position and size.
 * @return Tile tile that the head moved into.
 */
private TileTypeupdateSnake()
{
Direction direction = directions.peekFirst();

/*
 * Here we calculate the new point that the snake's head will be at
* after the update.
*/
Point head = new Point(snake.peekFirst());
switch(direction)
{
case North:
head.y--;
break;
```

```
case South:
head.y++;
break;
case West:
head.x--;
break;
case East:
head.x++;
break;
}
if(head.x< 0 || head.x>= BoardPanel.COL_COUNT || head.y< 0 || head.y>=
BoardPanel.ROW_COUNT)
{
return TileType.SnakeBody; //Pretend we collided with our body.

}
TileType old = board.getTile(head.x, head.y);
if(old != TileType.Fruit&&snake.size() > MIN_SNAKE_LENGTH)
{
Point tail = snake.removeLast();
board.setTile(tail, null);
old = board.getTile(head.x, head.y);
}
if(old != TileType.SnakeBody)
{
board.setTile(snake.peekFirst(), TileType.SnakeBody);
snake.push(head);
board.setTile(head, TileType.SnakeHead);
if(directions.size() > 1)
{
directions.poll();
}
}
return old;
```

```
}
/**
* Resets the game's variables to their default states and starts a new game. */
private void resetGame()

{

this.score = 0;
this.fruitsEaten = 0;
this.isNewGame = false;
this.isGameOver = false;
Point head = new Point(BoardPanel.COL_COUNT / 2, BoardPanel.ROW_COUNT / 2);
snake.clear();
snake.add(head);
board.clearBoard();
board.setTile(head, TileType.SnakeHead);
directions.clear();
directions.add(Direction.North);
logicTimer.reset();
spawnFruit();
}
/**
 * Gets the flag that indicates whether or not we're playing a new game.
 * @return The new game flag.
*/
public booleanisNewGame()
{
return isNewGame;

}
public booleanisGameOver()
{
return isGameOver;
}
```

```
/**
 * Gets the flag that indicates whether or not the game is paused.
 * @return The paused flag.
 */
public booleanisPaused()
{
return isPaused;
}
/**
 * Spawns a new fruit onto the board.
 */
private void spawnFruit()
{
//Reset the score for this fruit to 100.
this.nextFruitScore = 100;
int index = random.nextInt(BoardPanel.COL_COUNT * BoardPanel.ROW_COUNT -
snake.size());
intfreeFound = -1;
for(int x = 0; x <BoardPanel.COL_COUNT; x++)

{
for(int y = 0; y <BoardPanel.ROW_COUNT; y++)
{
TileType type = board.getTile(x, y);
if(type == null || type == TileType.Fruit)
{
if(++freeFound == index)
{
board.setTile(x, y, TileType.Fruit);
break;
}
}
}
}
```

```
}
/**
* Gets the current score.
* @return The score.
*/
public intgetScore()
{
return score;
}

/**
 * Gets the number of fruits eaten.
 * @return The fruits eaten. */
public intgetFruitsEaten()
{
return fruitsEaten;
}
/**
 * Gets the next fruit score.
 * @return The next fruit score.
 */
public intgetNextFruitScore()
{
return nextFruitScore;
}
/**
 * Gets the current direction of the snake.
 * @return The current direction.
 */

public Direction getDirection()
{
return directions.peek();
```

```
}

public static void main(String[] args)
{
SnakeGame snake = new SnakeGame();
snake.startGame();
}
}
```

//BoardPanel.java

```
import java.awt.BorderLayout;
import java.awt.Point;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.LinkedList;
import java.util.Random;
import javax.swing.JFrame;
public class SnakeGame extends JFrame
{
private static final long serialVersionUID = 6678292058307426314L;
private static final long FRAME_TIME = 1000L / 50L;
private static final int MIN_SNAKE_LENGTH = 5;
/**
 * The maximum number of directions that we can have polled in the
 * direction list.
 */
private static final int MAX_DIRECTIONS = 3;
/**
 * The BoardPanel instance.
 */
private BoardPanel board;

/**
 * The SidePanel instance.
 */
private SidePanel side;
/**
 * The random number generator (used for spawning fruits).
 */
private Random random;
private Clock logicTimer;
private booleanisNewGame;
```

```
private booleanisGameOver;
private booleanisPaused;
private LinkedList<Point> snake;
private LinkedList<Direction> directions;
private int score;
private intfruitsEaten;
private intnextFruitScore;
/**
 * Creates a new SnakeGame instance. Creates a new window,
 * and sets up the controller input.
 */
private SnakeGame()
{

super("Snake Game");
setLayout(new BorderLayout());
setDefaultCloseOperation(EXIT_ON_CLOSE);
setResizable(false);
/*
 * Initialize the game's panels and add them to the window. */
this.board = new BoardPanel(this);
this.side = new SidePanel(this);
add(board, BorderLayout.CENTER);
add(side, BorderLayout.EAST);
/*
 * Adds a new key listener to the frame to process input.
 */
addKeyListener(new KeyAdapter()
{
@Override
public void keyPressed(KeyEvent e)
{
switch(e.getKeyCode())
{
```

```
case KeyEvent.VK_W:
case KeyEvent.VK_UP:
if(!isPaused&& !isGameOver)

{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.South&& last != Direction.North)
{
directions.addLast(Direction.North);
}
}
}
break;
case KeyEvent.VK_S:
case KeyEvent.VK_DOWN:
if(!isPaused&& !isGameOver)
{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.North&& last != Direction.South)
{
directions.addLast(Direction.South);
}
}

}
break;
case KeyEvent.VK_A:
case KeyEvent.VK_LEFT:
if(!isPaused&& !isGameOver)
{
```

```
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();
if(last != Direction.East&& last != Direction.West)
{
directions.addLast(Direction.West);
}
}
}
break;
case KeyEvent.VK_D:
case KeyEvent.VK_RIGHT:
if(!isPaused&& !isGameOver)
{
if(directions.size() < MAX_DIRECTIONS)
{
Direction last = directions.peekLast();

if(last != Direction.West&& last != Direction.East)
{
directions.addLast(Direction.East);
}
}
}
break;
case KeyEvent.VK_P:
if(!isGameOver)
{
isPaused = !isPaused;
logicTimer.setPaused(isPaused);
}
break;
/*
 * Reset the game if one is not currently in progress.
```

```
 */
case KeyEvent.VK_ENTER:
if(isNewGame || isGameOver)
{
resetGame();
}
break;

}
}
});
pack();
setLocationRelativeTo(null);
setVisible(true);
}
/**
* Starts the game running.
 */
private void startGame()
{
/*
* Initialize everything we're going to be using.
*/
this.random = new Random();
this.snake = new LinkedList<>();
this.directions = new LinkedList<>();
this.logicTimer = new Clock(9.0f);
this.isNewGame = true;
//Set the timer to paused initially.
logicTimer.setPaused(true);
while(true)

{
//Get the current frame's start time.
```

```
long start = System.nanoTime();
//Update the logic timer.
logicTimer.update();
if(logicTimer.hasElapsedCycle())
{
updateGame();
}
//Repaint the board and side panel with the new content.
board.repaint();
side.repaint();
long delta = (System.nanoTime() - start) / 1000000L;
if(delta < FRAME_TIME)
{
try
{
Thread.sleep(FRAME_TIME - delta);
}
 catch(Exception e)
{
e.printStackTrace();
}
}
}
}
/**
* Updates the game's logic.
*/
private void updateGame()
{
TileType collision = updateSnake();
if(collision == TileType.Fruit)
{
fruitsEaten++;
score += nextFruitScore;
```

```
spawnFruit();
}
else if(collision == TileType.SnakeBody)
{
isGameOver = true;
logicTimer.setPaused(true);
}
else if(nextFruitScore> 10)
{
nextFruitScore--;

}
}
private TileTypeupdateSnake()
{
Direction direction = directions.peekFirst();
Point head = new Point(snake.peekFirst());
switch(direction)
{
case North:
head.y--;
break;
case South:
head.y++;
break;
case West:
head.x--;
break;
case East:
head.x++;
break;
}
if(head.x< 0 || head.x>= BoardPanel.COL_COUNT || head.y< 0 || head.y>=
BoardPanel.ROW_COUNT) {
```

```
return TileType.SnakeBody; //Pretend we collided with our body.
}
TileType old = board.getTile(head.x, head.y);
if(old != TileType.Fruit&&snake.size() > MIN_SNAKE_LENGTH)
{
Point tail = snake.removeLast();
board.setTile(tail, null);
old = board.getTile(head.x, head.y);
}
if(old != TileType.SnakeBody)
{
board.setTile(snake.peekFirst(), TileType.SnakeBody);
snake.push(head);
board.setTile(head, TileType.SnakeHead);
if(directions.size() > 1)
{
directions.poll();
}
}
return old;
}
/**
* Resets the game's variables to their default states and starts a new game.
 */

private void resetGame()
{
this.score = 0;
this.fruitsEaten = 0;
this.isNewGame = false;
this.isGameOver = false;
/*
 * Create the head at the center of the board.
 */
```

```
Point head = new Point(BoardPanel.COL_COUNT / 2, BoardPanel.ROW_COUNT / 2);
/*
 * Clear the snake list and add the head.
 */
snake.clear();
snake.add(head);
/*
 * Clear the board and add the head.
 */
board.clearBoard();
board.setTile(head, TileType.SnakeHead);
/*
 * Clear the directions and add north as the
 * default direction.

 */
directions.clear();
directions.add(Direction.North);
/*
 * Reset the logic timer.
 */
logicTimer.reset();
spawnFruit();
}
public booleanisNewGame()
{
return isNewGame;
}
/**
* Gets the flag that indicates whether or not the game is over.
* @return The game over flag.
*/
public booleanisGameOver()
{
```

```
return isGameOver;
}
/**
* Gets the flag that indicates whether or not the game is paused.

* @return The paused flag.
*/
public booleanisPaused()
{
return isPaused;
}

/**
* Spawns a new fruit onto the board.
*/
private void spawnFruit()
{
//Reset the score for this fruit to 100.
this.nextFruitScore = 100;
/*
 * Get a random index based on the number of free spaces left on the board.
*/
int index = random.nextInt(BoardPanel.COL_COUNT * BoardPanel.ROW_COUNT -
snake.size());
intfreeFound = -1;
for(int x = 0; x <BoardPanel.COL_COUNT; x++)
{
for(int y = 0; y <BoardPanel.ROW_COUNT; y++)
{

TileType type = board.getTile(x, y);
if(type == null || type == TileType.Fruit)
{
if(++freeFound == index)
```

```
{
board.setTile(x, y, TileType.Fruit);
break;
}
}
}
}
}
/**
 * Gets the current score.
* @return The score.
 */
public intgetScore()
 {
return score;
}
/**
* Gets the number of fruits eaten.
* @return The fruits eaten.

*/
public intgetFruitsEaten()
{
return fruitsEaten;
}
/**
* Gets the next fruit score.
* @return The next fruit score.
*/
public intgetNextFruitScore()
 {
return nextFruitScore;
}
/**
```

```
* Gets the current direction of the snake.
* @return The current direction.
 */

public Direction getDirection()
 {
return directions.peek();
}


public static void main(String[] args)
{
SnakeGame snake = new SnakeGame();
snake.startGame();
}
}
```

//SidePanel.java

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import javax.swing.JPanel;
public class SidePanel extends JPanel
{
private static final long serialVersionUID = -40557434900946408L;
private static final Font LARGE_FONT = new Font("Tahoma", Font.BOLD, 20);
private static final Font MEDIUM_FONT = new Font("Tahoma", Font.BOLD, 16);
private static final Font SMALL_FONT = new Font("Tahoma", Font.BOLD, 12);
private SnakeGame game;

public SidePanel(SnakeGame game)
 {
this.game = game;
setPreferredSize(new Dimension(300, BoardPanel.ROW_COUNT *
BoardPanel.TILE_SIZE));
setBackground(Color.WHITE);
}

private static final int STATISTICS_OFFSET = 150;
private static final int CONTROLS_OFFSET = 320;
private static final int MESSAGE_STRIDE = 30;
private static final int SMALL_OFFSET = 30;
private static final int LARGE_OFFSET = 50;
@Override

public void paintComponent(Graphics g)
 {
super.paintComponent(g);
g.setColor(Color.BLACK);
```

```
g.setFont(LARGE_FONT);
g.drawString("Snake Game", getWidth() / 2 - g.getFontMetrics().stringWidth("Snake Game")
/ 2, 50);
g.setFont(MEDIUM_FONT);
g.drawString("Statistics and Score", SMALL_OFFSET, STATISTICS_OFFSET);
g.drawString("Controls", SMALL_OFFSET, CONTROLS_OFFSET);
g.setFont(SMALL_FONT);
//Draw the content for the statistics category.
intdrawY = STATISTICS_OFFSET;
g.drawString("Total Score: " + game.getScore(), LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);
g.drawString("Apples Eaten: " + game.getFruitsEaten(), LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);

g.drawString("Apple Score: " + game.getNextFruitScore(), LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);
//Draw the content for the controls category.
drawY = CONTROLS_OFFSET;
g.drawString("Move Up: W / Up Arrowkey", LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);
g.drawString("Move Down: S / Down Arrowkey", LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);
g.drawString("Move Left: A / Left Arrowkey", LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);
g.drawString("Move Right: D / Right Arrowkey", LARGE_OFFSET, drawY +=
MESSAGE_STRIDE);
g.drawString("Pause Game: P", LARGE_OFFSET, drawY += MESSAGE_STRIDE);
}
}
```

//Direction java

```
publicenum Direction
{

/**
 * Moving North (Up).
 */
North,

/**
 * Moving East (Right).
 */
East,

/**
 * Moving South (Down).
 */
South,

/**
 * Moving West (Left).
 */
West

}
```

//TitleType.java

```
publicenumTileType
 {
Fruit,
SnakeHead,
SnakeBody
}
```

//Clock.java

```
publicclass Clock
 {
/**
        * The number of milliseconds that make up one cycle.
        */
privatefloatmillisPerCycle;
/**
        * The last time that the clock was updated (used for calculating the
        * delta time).
        */
privatelonglastUpdate;
/**
        * The number of cycles that have elapsed and have not yet been polled.
        */
privateintelapsedCycles;
/**
        * The amount of excess time towards the next elapsed cycle.
        */
privatefloatexcessCycles;
/**
        * Whether or not the clock is paused.
        */
privatebooleanisPaused;
public Clock(floatcyclesPerSecond)
{
setCyclesPerSecond(cyclesPerSecond);
reset();
}
publicvoidsetCyclesPerSecond(floatcyclesPerSecond)

{
this.millisPerCycle = (1.0f / cyclesPerSecond) * 1000;
}
/**
        * Resets the clock stats. Elapsed cycles and cycle excess will be reset
        * to 0, the last update time will be reset to the current time, and the
```

```
        * paused flag will be set to false.
        */
publicvoid reset()
{
this.elapsedCycles = 0;
this.excessCycles = 0.0f;
this.lastUpdate = getCurrentTime();
this.isPaused = false;
}
/**
        * Updates the clock stats. The number of elapsed cycles, as well as the
        * cycle excess will be calculated only if the clock is not paused. This
        * method should be called every frame even when paused to prevent any
        * nasty surprises with the delta time.
        */
publicvoid update()
{
//Get the current time and calculate the delta time.
longcurrUpdate = getCurrentTime();
float delta = (float)(currUpdate - lastUpdate) + excessCycles;
//Update the number of elapsed and excess ticks if we're not paused.
if(!isPaused)
 {
this.elapsedCycles += (int)Math.floor(delta / millisPerCycle);
this.excessCycles = delta % millisPerCycle;
}

//Set the last update time for the next update cycle.
this.lastUpdate = currUpdate;
}

publicvoidsetPaused(boolean paused)
 {
this.isPaused = paused;
}
/**
```

```
        * Checks to see if the clock is currently paused.
        * @return Whether or not this clock is paused.
        */
publicbooleanisPaused()
{
returnisPaused;
}
/**
        * Checks to see if a cycle has elapsed for this clock yet. If so,
        * the number of elapsed cycles will be decremented by one.
        * @return Whether or not a cycle has elapsed.
        * @seepeekElapsedCycle
        */
publicbooleanhasElapsedCycle()
{
if(elapsedCycles> 0)
{
this.elapsedCycles--;
returntrue;
}
returnfalse;
}
/**
        * Checks to see if a cycle has elapsed for this clock yet. Unlike
        * {@code hasElapsedCycle}, the number of cycles will not be decremented
        * if the number of elapsed cycles is greater than 0.
        * @return Whether or not a cycle has elapsed.
        * @seehasElapsedCycle
        */
publicbooleanpeekElapsedCycle()
{
return (elapsedCycles> 0);


}
/**
        * Calculates the current time in milliseconds using the computer's high
        * resolution clock. This is much more reliable than
        * {@code System.getCurrentTimeMillis()}, and quicker than
        * {@code System.nanoTime()}.
        * @return The current time in milliseconds.
```

```
        */
privatestaticfinallonggetCurrentTime()
{
return (System.nanoTime() / 1000000L);
}

}
```