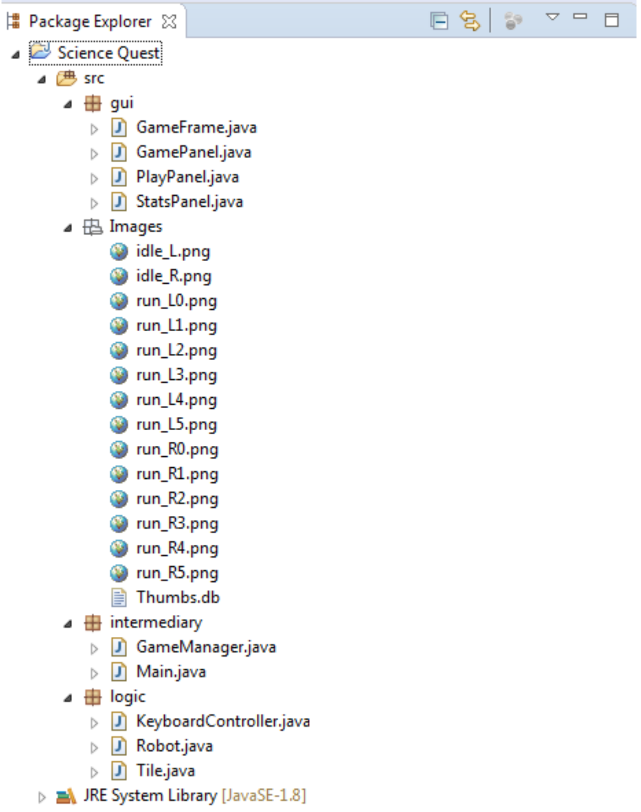
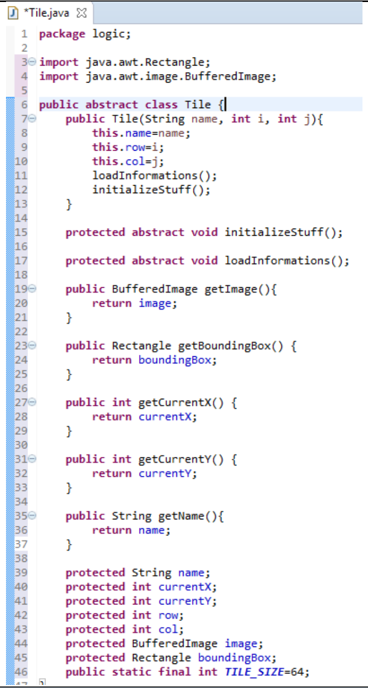
Developing the Coded Solution for Project *Version 3*

With the character completed, version 3 concerns itself with the creation of the environment that the game will take place in. This will create the world that will house each level and deal with the transitions in between; the tiles that will that the role of platforms and the logic that will begin to give the robot class limited interactions with its environment.



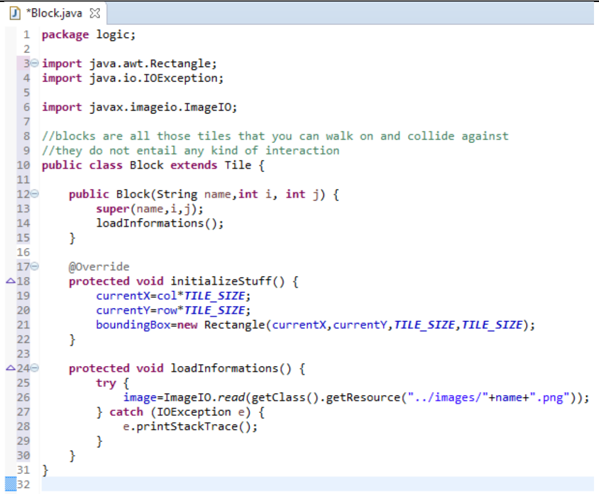
```
Package Explorer
Science Quest
  src
    gui
      GameFrame.java
      GamePanel.java
      PlayPanel.java
      StatsPanel.java
    Images
      idle_L.png
      idle_R.png
      run_L0.png
      run_L1.png
      run_L2.png
      run_L3.png
      run_L4.png
      run_L5.png
      run_R0.png
      run_R1.png
      run_R2.png
      run_R3.png
      run_R4.png
      run_R5.png
      Thumbs.db
    intermediary
      GameManager.java
      Main.java
    logic
      KeyboardController.java
      Robot.java
      Tile.java
  JRE System Library [JavaSE-1.8]
```

From version 2, we have the **logic** and **gui** packages that allow the robot to function. With the exception of the empty **Tile** class, all environmental aspects of the game will have to be created from scratch.



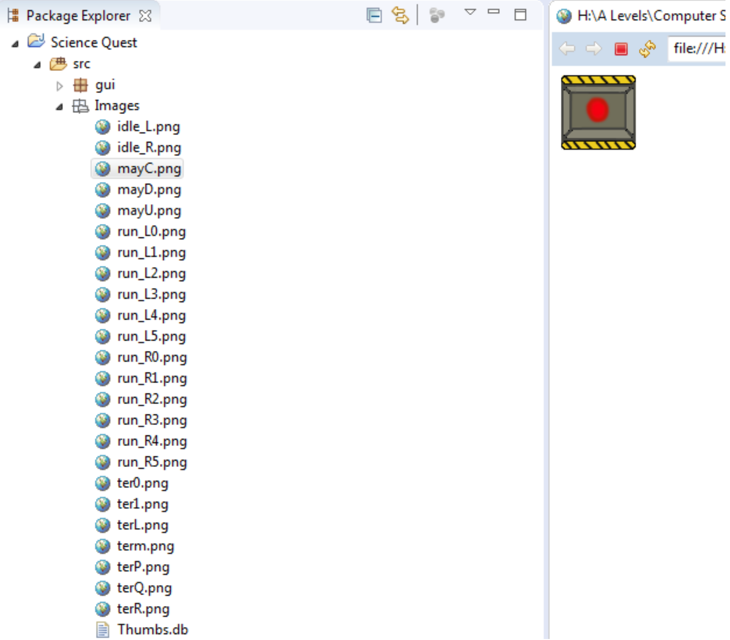
```
*Tile.java
package logic;

import java.awt.Rectangle;
import java.awt.image.BufferedImage;

public abstract class Tile {
    public Tile(String name, int i, int j){
        this.name=name;
        this.row=i;
        this.col=j;
        loadInformations();
        initializeStuff();
    }

    protected abstract void initializeStuff();

    protected abstract void loadInformations();

    public BufferedImage getImage(){
        return image;
    }

    public Rectangle getBoundingBox() {
        return boundingBox;
    }

    public int getCurrentX() {
        return currentX;
    }

    public int getCurrentY() {
        return currentY;
    }

    public String getName(){
        return name;
    }

    protected String name;
    protected int currentX;
    protected int currentY;
    protected int row;
    protected int col;
    protected BufferedImage image;
    protected Rectangle boundingBox;
    public static final int TILE_SIZE=64;
```

Finally, I can work on the **Tile** class. This is an abstract class as it is important to not want to instantiate a generic **Tile**, but rather specific types of tiles that behave in different ways. This will allow for the development of child classes, such as a tile that will make up sections of the platforms or even the collectable batteries in the later versions.



```
*Block.java
package logic;

import java.awt.Rectangle;
import java.io.IOException;

import javax.imageio.ImageIO;

//blocks are all those tiles that you can walk on and collide against
//they do not entail any kind of interaction
public class Block extends Tile {

    public Block(String name,int i, int j) {
        super(name,i,j);
        loadInformations();
    }

    @Override
    protected void initializeStuff() {
        currentX=col*TILE_SIZE;
        currentY=row*TILE_SIZE;
        boundingBox=new Rectangle(currentX,currentY,TILE_SIZE,TILE_SIZE);
    }

    protected void loadInformations() {
        try {
            image=ImageIO.read(getClass().getResource("../images/"+name+".png"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

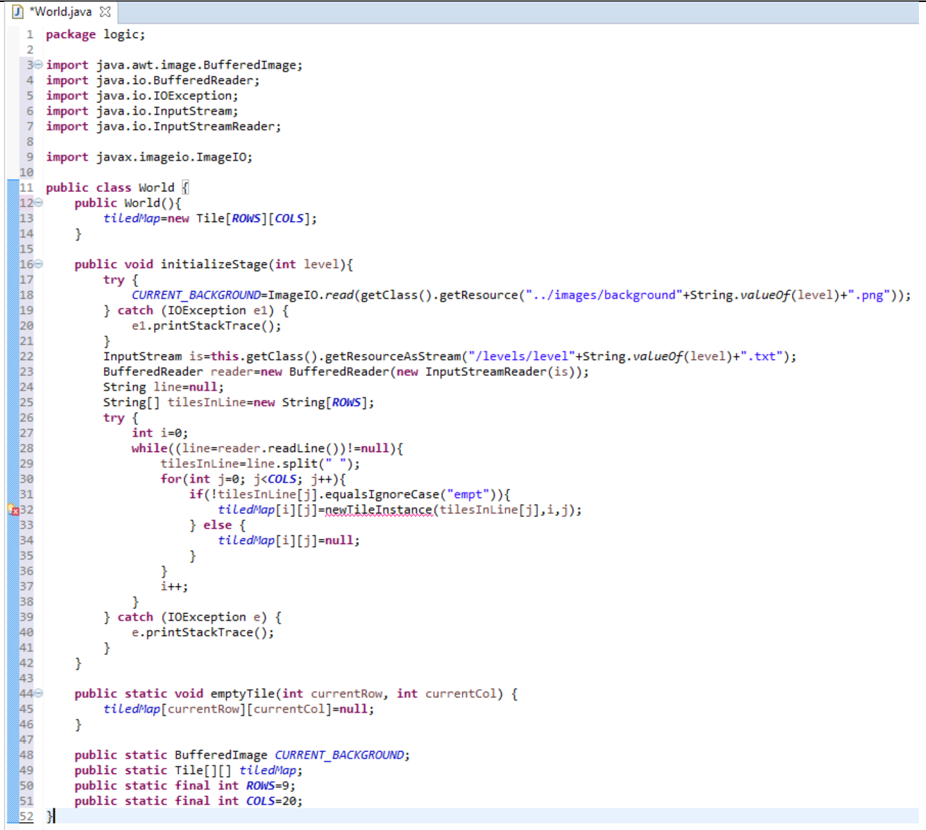
The **Block** class extends **Tile**: you can think of a block as any piece of material you can walk on and collide with. Its bounding box occupies the entire perimeter of a cell in the tiled map grid. Each block will perform as a segment of a platform, giving the game its important platform element.



```
Package Explorer
  Science Quest
    src
      gui
      Images
        idle_L.png
        idle_R.png
        mayC.png
        mayD.png
        mayU.png
        run_L0.png
        run_L1.png
        run_L2.png
        run_L3.png
        run_L4.png
        run_L5.png
        run_R0.png
        run_R1.png
        run_R2.png
        run_R3.png
        run_R4.png
        run_R5.png
        ter0.png
        ter1.png
        terL.png
        term.png
        terP.png
        terQ.png
        terR.png
        Thumbs.db
```

The next step was to make images for the tiles. For the block tiles, there are two different types of tiles aesthetically- platform tiles (saved by the name of may_, *an inside joke due to a draft of the tile was saved a maybe*) and terrain tiles (saved by the name of ter_).

While all the different images will be reskins of the same block, they exist in a variation to make the game an increase of aesthetic complexity. I plan on making more tile skins for each level in a later version.



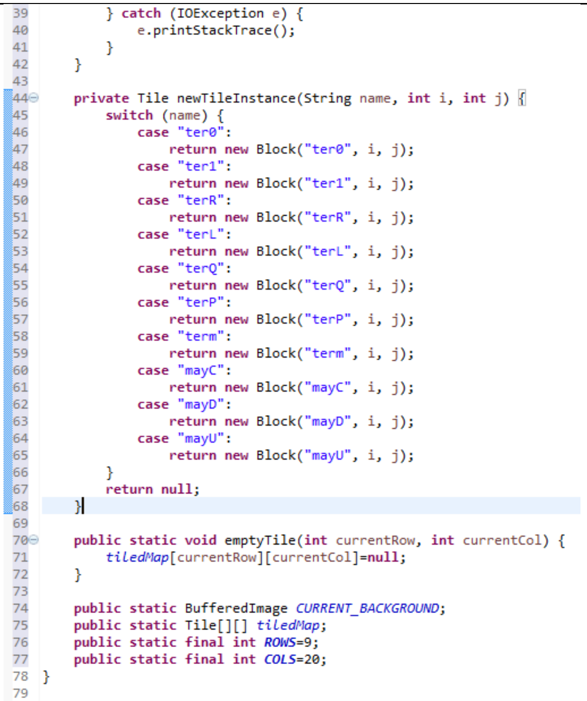
```java
 1 package logic;
 2 
 3 import java.awt.image.BufferedImage;
 4 import java.io.BufferedReader;
 5 import java.io.IOException;
 6 import java.io.InputStream;
 7 import java.io.InputStreamReader;
 8 
 9 import javax.imageio.ImageIO;
10 
11 public class World {
12     public World(){
13         tiledMap=new Tile[ROWS][COLS];
14     }
15 
16     public void initializeStage(int level){
17         try {
18             CURRENT_BACKGROUND=ImageIO.read(getClass().getResource("../images/background"+String.valueOf(level)+".png"));
19         } catch (IOException e1) {
20             e1.printStackTrace();
21         }
22         InputStream is=this.getClass().getResourceAsStream("/levels/level"+String.valueOf(level)+".txt");
23         BufferedReader reader=new BufferedReader(new InputStreamReader(is));
24         String line=null;
25         String[] tilesInLine=new String[ROWS];
26         try {
27             int i=0;
28             while((line=reader.readLine())!=null){
29                 tilesInLine=line.split(" ");
30                 for(int j=0; j<COLS; j++){
31                     if(!tilesInLine[j].equalsIgnoreCase("empt")){
32                         tiledMap[i][j]=newTileInstance(tilesInLine[j],i,j);
33                     } else {
34                         tiledMap[i][j]=null;
35                     }
36                 }
37                 i++;
38             }
39         } catch (IOException e) {
40             e.printStackTrace();
41         }
42     }
43 
44     public static void emptyTile(int currentRow, int currentCol) {
45         tiledMap[currentRow][currentCol]=null;
46     }
47 
48     public static BufferedImage CURRENT_BACKGROUND;
49     public static Tile[][] tiledMap;
50     public static final int ROWS=9;
51     public static final int COLS=20;
52 }
```

With the tiles created, I needed to create an environment for my game to take place. This environment is created by the **World** class.

Right now the **World** class only makes a *tiledMap.* The tiled map for this game has 9 rows and 20 columns for a total of 180 cells that will contain single static elements of the game (lines 50 and 51).

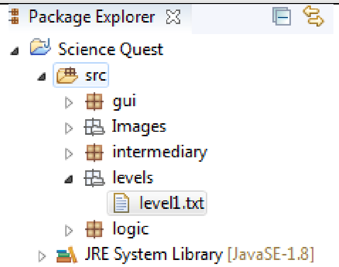
All the tiles of the current stage are stored in this two dimensional array. The tiledMap is public, that's because there's no need to hide any information about the world as you see it. This also opens up the ability to fetch the tiled Map from anywhere in the code, adding a layer of convenience.

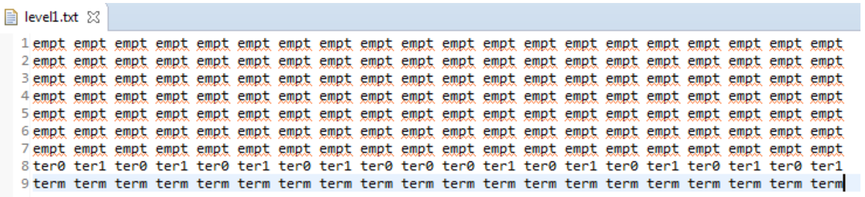
The *initializeStage* method fetches the layout of the tiles in a given level, and the background image. These two aspects are important for creating levels.



```
39          } catch (IOException e) {
40              e.printStackTrace();
41          }
42      }
43  
44      private Tile newTileInstance(String name, int i, int j) {
45          switch (name) {
46              case "ter0":
47                  return new Block("ter0", i, j);
48              case "ter1":
49                  return new Block("ter1", i, j);
50              case "terR":
51                  return new Block("terR", i, j);
52              case "terL":
53                  return new Block("terL", i, j);
54              case "terQ":
55                  return new Block("terQ", i, j);
56              case "terP":
57                  return new Block("terP", i, j);
58              case "term":
59                  return new Block("term", i, j);
60              case "mayC":
61                  return new Block("mayC", i, j);
62              case "mayD":
63                  return new Block("mayD", i, j);
64              case "mayU":
65                  return new Block("mayU", i, j);
66          }
67          return null;
68      }
69  
70      public static void emptyTile(int currentRow, int currentCol) {
71          tiledMap[currentRow][currentCol]=null;
72      }
73  
74      public static BufferedImage CURRENT_BACKGROUND;
75      public static Tile[][] tiledMap;
76      public static final int ROWS=9;
77      public static final int COLS=20;
78  }
79  
```

The final section allows me to enter different tiles to be recognized on the *tiledMap* array. This will allow me to develop many different tiles and implement them quickly, as the **World** class actually loads the current disposition of tiles from a simple text file.



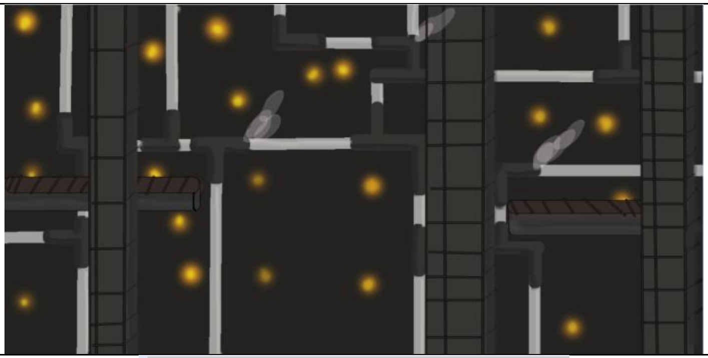


level1.txt

```
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt empt
ter0 ter1 ter0 ter1 ter0 ter1 ter0 ter1 ter0 ter0 ter0 ter1 ter0 ter1 ter0 ter1 ter0 ter1 ter0 ter1
term term term term term term term term term term term term term term term term term term term term
```

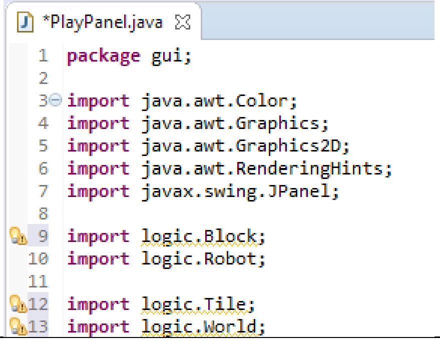
The simple text file in question is located in the newly created **levels** folder. Line 22-42 fetches the information and reads it, placing tiles in the recorded position. It will then increment number next to the level when fetching, allowing it to automatically go to level1 to level2 and so on.

The next file itself is full of "empt" tiles that haven't been included in the *newTileInstance.* This is because they represent a lack of a tile and are used as a placeholder to allow the code to work in the given format lines 31-32 in the **World** class shows the empt tile case.



Like the text documents that are fetched to get information on the tile layout of the world, the background is also fetched from the **images** folder. This is done in lines 17-21, but needs an *IOException* read the image (the same issue exists with the .txt files).

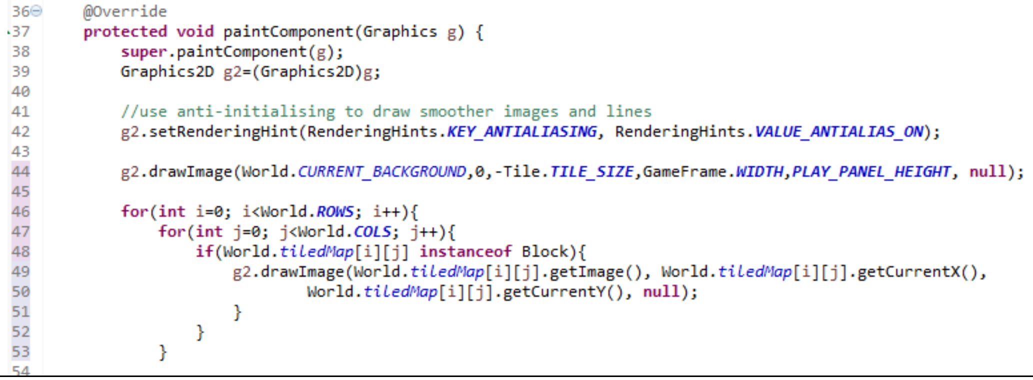
This background will be used to test the mechanics and may be used as an introduction level to get the user familiar with the controls.



*PlayPanel.java

```
1  package gui;
2
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.RenderingHints;
7  import javax.swing.JPanel;
8
9  import logic.Block;
10 import logic.Robot;
11
12 import logic.Tile;
13 import logic.World;
```

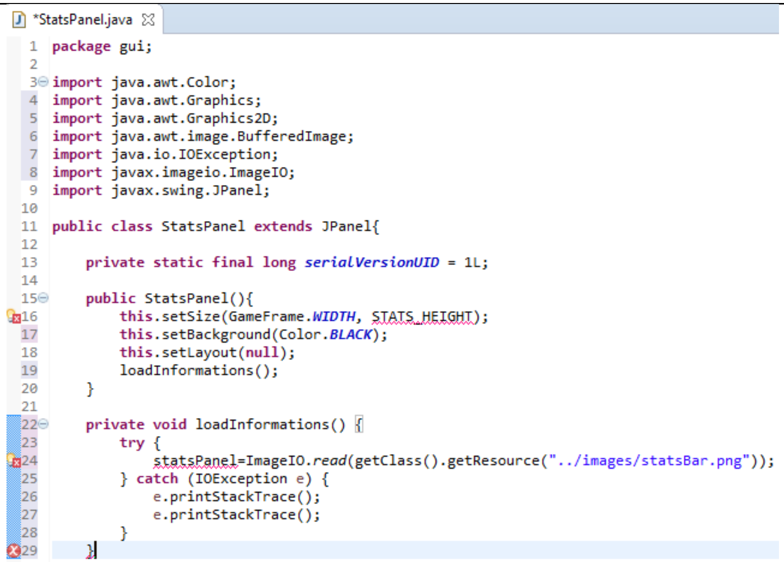
With the logic of the environmental classes made, it is now time to start implementing them with the graphical elements of the project. Starting at the PlayPanel, I need to show the tiles and the world to user.



```
36      @Override
37      protected void paintComponent(Graphics g) {
38          super.paintComponent(g);
39          Graphics2D g2=(Graphics2D)g;
40
41          //use anti-initialising to draw smoother images and lines
42          g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
43
44          g2.drawImage(World.CURRENT_BACKGROUND,0,-Tile.TILE_SIZE,GameFrame.WIDTH,PLAY_PANEL_HEIGHT, null);
45
46          for(int i=0; i<World.ROWS; i++){
47              for(int j=0; j<World.COLS; j++){
48                  if(World.tiledMap[i][j] instanceof Block){
49                      g2.drawImage(World.tiledMap[i][j].getImage(), World.tiledMap[i][j].getCurrentX(),
50                              World.tiledMap[i][j].getCurrentY(), null);
51                  }
52              }
53          }
54
```

The first step was to get rid of lines 40-44, as the lines used to judge the size of tiles and give a perspective of distance are redundant with the presence of real tiles.

Line 44 draws the fetched level background, followed by lines 46 to 54 fetching and placing the tiles following the text file.



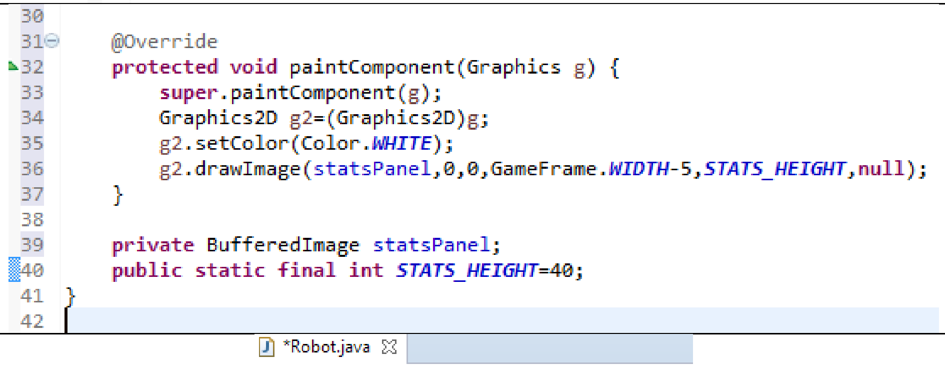
*StatsPanel.java

```
1  package gui;
2
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.image.BufferedImage;
7  import java.io.IOException;
8  import javax.imageio.ImageIO;
9  import javax.swing.JPanel;
10
11 public class StatsPanel extends JPanel{
12
13     private static final long serialVersionUID = 1L;
14
15     public StatsPanel(){
16         this.setSize(GameFrame.WIDTH, STATS_HEIGHT);
17         this.setBackground(Color.BLACK);
18         this.setLayout(null);
19         loadInformations();
20     }
21
22     private void loadInformations() {
23         try {
24             statsPanel=ImageIO.read(getClass().getResource("../images/statsBar.png"));
25         } catch (IOException e) {
26             e.printStackTrace();
27             e.printStackTrace();
28         }
29     }
```

While I'm updating the **PlayPanel**, I decided to update the **StatsPanel** too. It still doesn't serve any function as there is nothing to be recorded, but I have created an image inspired by a circuit board to become the visuals of the stats panel.

Firstly, I created a black rectangle for the space that my panel will take (lines 18 to 21), as my image is thicker than my placeholder, I will have to assign a new value to STATS_HEIGHT.

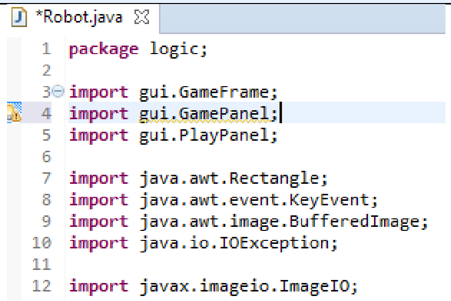
The next step is to fetch the visual for my statsBar (this has been added to the images folder). This image will be used in each level and will not be changed.



```
30
31     @Override
32     protected void paintComponent(Graphics g) {
33         super.paintComponent(g);
34         Graphics2D g2=(Graphics2D)g;
35         g2.setColor(Color.WHITE);
36         g2.drawImage(statsPanel,0,0,GameFrame.WIDTH-5,STATS_HEIGHT,null);
37     }
38
39     private BufferedImage statsPanel;
40     public static final int STATS_HEIGHT=40;
41 }
42
```

The next step is to draw the fetched image onto the stats panel. This will provide a platform for all recorded statistical information to be recorded and displayed to the user.

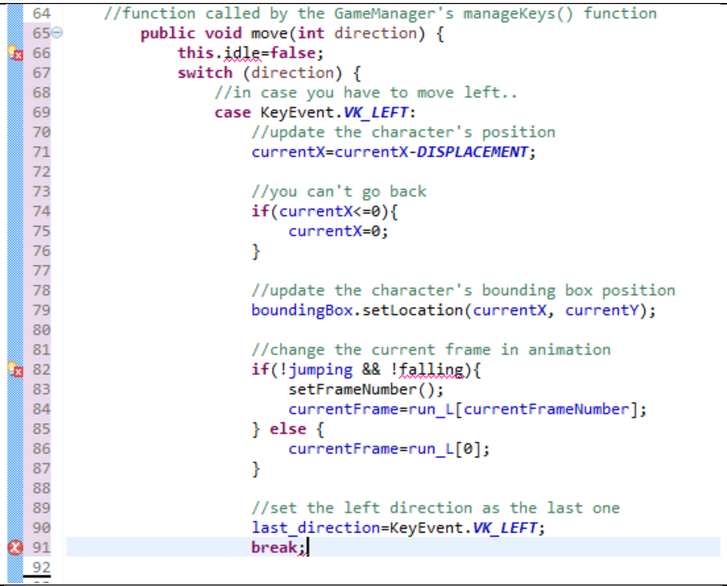
The last step is to now supply the dimensions of the stats panel so it can be drawn.



*Robot.java

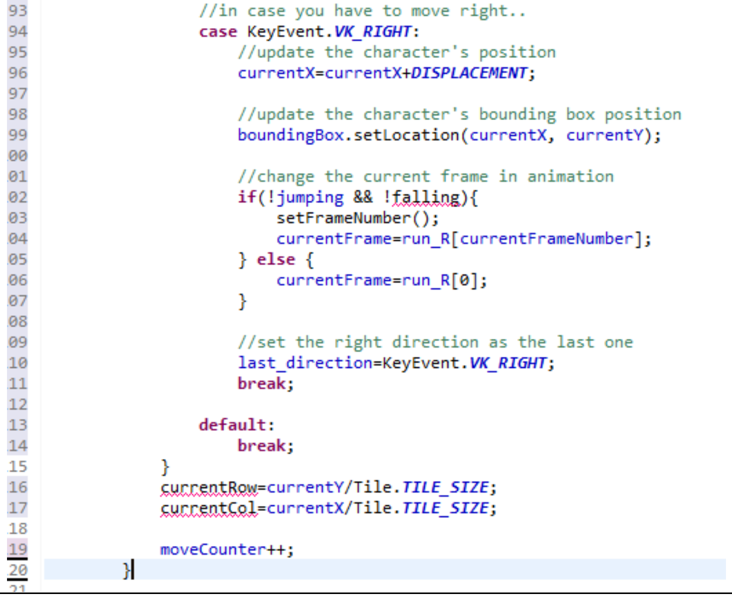
```
package logic;

import gui.GameFrame;
import gui.GamePanel;
import gui.PlayPanel;

import java.awt.Rectangle;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.IOException;

import javax.imageio.ImageIO;
```

The next step is to update the **Robot** class so that the character will interact with the world around it. The first step is to import the **GamePanel** class, as the robot will have to be added again to each new level.



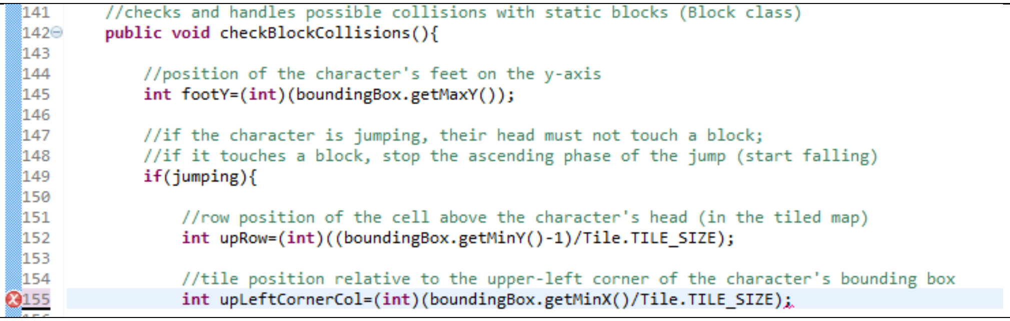
```
64  //function called by the GameManager's manageKeys() function
65      public void move(int direction) {
66          this.idle=false;
67          switch (direction) {
68              //in case you have to move left..
69              case KeyEvent.VK_LEFT:
70                  //update the character's position
71                  currentX=currentX-DISPLACEMENT;
72
73                  //you can't go back
74                  if(currentX<=0){
75                      currentX=0;
76                  }
77
78                  //update the character's bounding box position
79                  boundingBox.setLocation(currentX, currentY);
80
81                  //change the current frame in animation
82                  if(!jumping && !falling){
83                      setFrameNumber();
84                      currentFrame=run_L[currentFrameNumber];
85                  } else {
86                      currentFrame=run_L[0];
87                  }
88
89                  //set the left direction as the last one
90                  last_direction=KeyEvent.VK_LEFT;
91                  break;
92
```

For the blocks to have collisions, then I will have to first update the way the character moves to simulate the mechanic of something being solid without using a specialist gaming API. I have given the robot 3 new states, *idle, jumping* and *falling.* Each of these states will be called upon later to limit movement when the robot will collide with something.



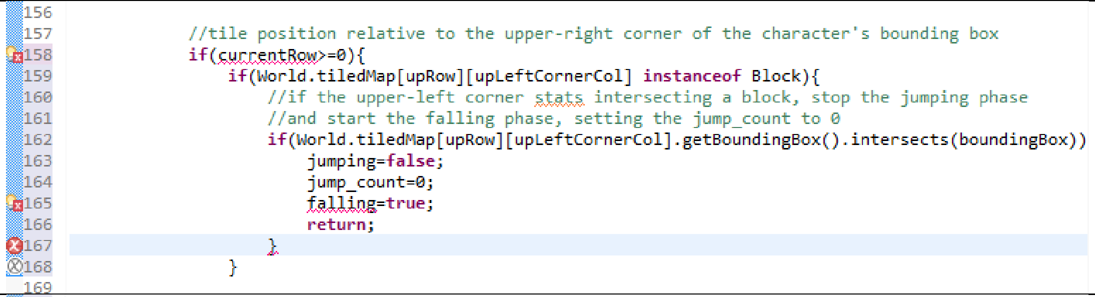
```
93              //in case you have to move right..
94              case KeyEvent.VK_RIGHT:
95                  //update the character's position
96                  currentX=currentX+DISPLACEMENT;
97
98                  //update the character's bounding box position
99                  boundingBox.setLocation(currentX, currentY);
00
01                  //change the current frame in animation
02                  if(!jumping && !falling){
03                      setFrameNumber();
04                      currentFrame=run_R[currentFrameNumber];
05                  } else {
06                      currentFrame=run_R[0];
07                  }
08
09                  //set the right direction as the last one
10                  last_direction=KeyEvent.VK_RIGHT;
11                  break;
12
13              default:
14                  break;
15          }
16          currentRow=currentY/Tile.TILE_SIZE;
17          currentCol=currentX/Tile.TILE_SIZE;
18
19          moveCounter++;
20      }
```

The same code is repeated and modified to fit the condition of movement to the right. References to the robot's X and Y position on the array have now been given in rows and columns in lines 116 and 117, allowing it to now be referenced with the *tiledMap*.



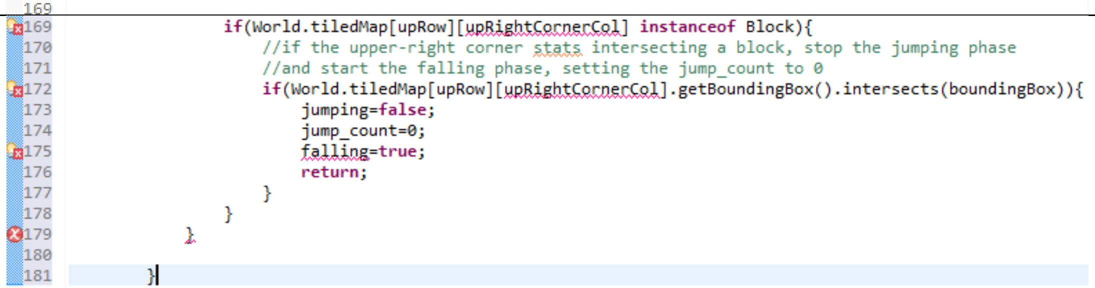
```
141     //checks and handles possible collisions with static blocks (Block class)
142     public void checkBlockCollisions(){
143 
144         //position of the character's feet on the y-axis
145         int footY=(int)(boundingBox.getMaxY());
146 
147         //if the character is jumping, their head must not touch a block;
148         //if it touches a block, stop the ascending phase of the jump (start falling)
149         if(jumping){
150 
151             //row position of the cell above the character's head (in the tiled map)
152             int upRow=(int)((boundingBox.getMinY()-1)/Tile.TILE_SIZE);
153 
154             //tile position relative to the upper-left corner of the character's bounding box
155             int upLeftCornerCol=(int)(boundingBox.getMinX()/Tile.TILE_SIZE);
```

This method *checkBlockCollisions(),* will set out the rules to create rules that will give the impression of solid objects without relying on a gaming API. The first step is to provide a way for the robot and the blocks into interact. These lines will allow the robot to reference the position of blocks based on their location on the tiled map, relative to the robots own coordinates.



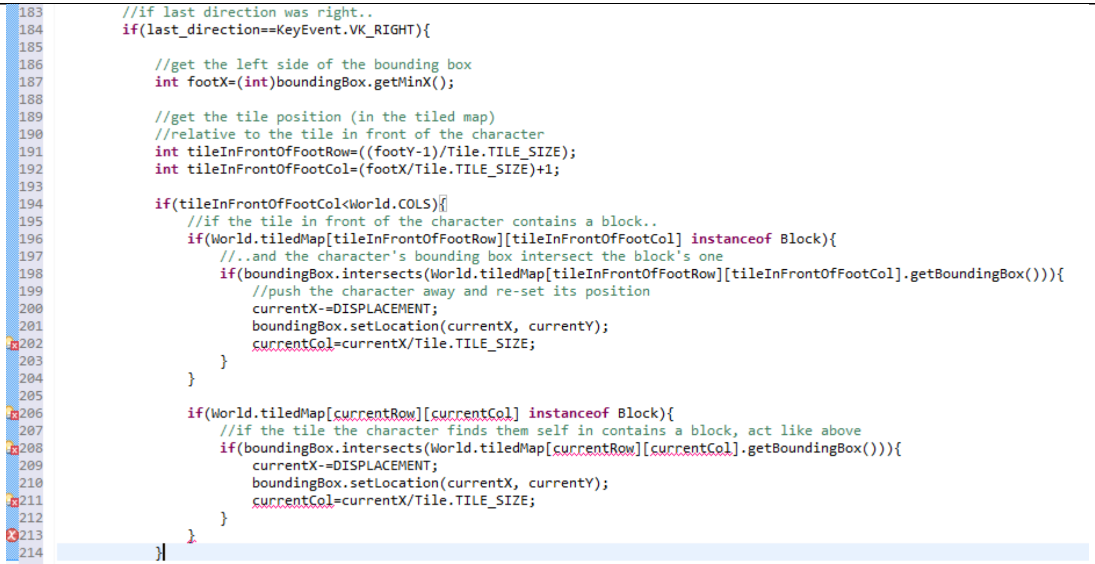
```
156 
157             //tile position relative to the upper-right corner of the character's bounding box
158             if(currentRow>=0){
159                 if(World.tiledMap[upRow][upLeftCornerCol] instanceof Block){
160                     //if the upper-left corner stats intersecting a block, stop the jumping phase
161                     //and start the falling phase, setting the jump_count to 0
162                     if(World.tiledMap[upRow][upLeftCornerCol].getBoundingBox().intersects(boundingBox))
163                         jumping=false;
164                         jump_count=0;
165                         falling=true;
166                         return;
167                     }
168                 }
169 
```

I briefly mentioned the introduction of the *jumping* and *falling* states. In lines 162 to 166 I have set up a condition to ensure that the robot cannot jump through platforms. By splitting the ascending phase and the descending phase of the jump. In particular, when the character is going up we have that the *jumping* boolean is set to true and the *falling* is set to false. When the robot is going down we have the opposite setting of those variables.



```
169                 if(World.tiledMap[upRow][upRightCornerCol] instanceof Block){
170                     //if the upper-right corner stats intersecting a block, stop the jumping phase
171                     //and start the falling phase, setting the jump_count to 0
172                     if(World.tiledMap[upRow][upRightCornerCol].getBoundingBox().intersects(boundingBox)){
173                         jumping=false;
174                         jump_count=0;
175                         falling=true;
176                         return;
177                     }
178                 }
179             }
180 
181     }
```

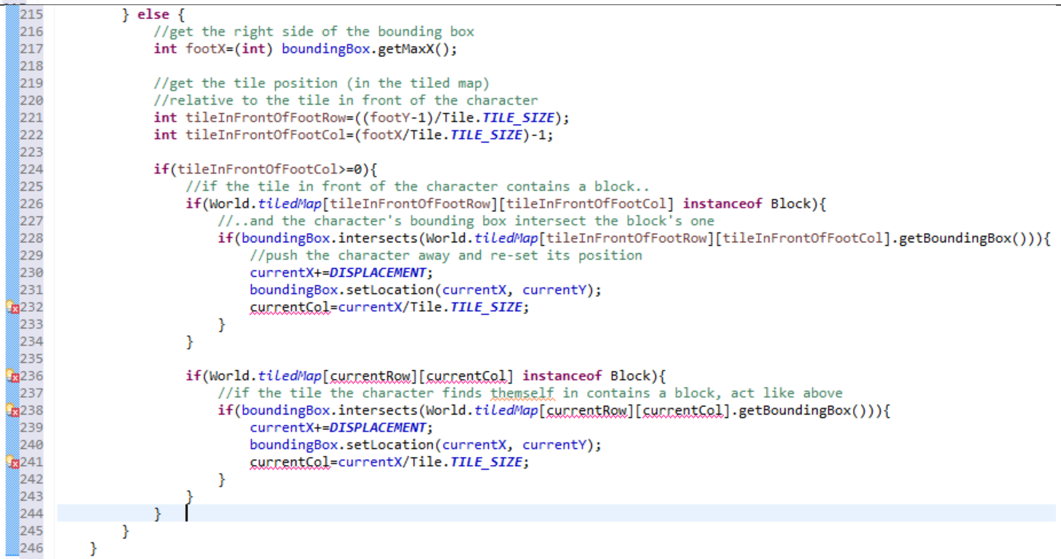
To have a specific *falling* state is really helpful even in other situations: for example I can set it to true when the character doesn’t have a Block object under his feet. This way I can make character fall (increment y position) while falling=true. But collision-wise, the character’s head can only collide if it’s in ascending phase.



```
        //if last direction was right..
        if(last_direction==KeyEvent.VK_RIGHT){

            //get the left side of the bounding box
            int footX=(int)boundingBox.getMinX();

            //get the tile position (in the tiled map)
            //relative to the tile in front of the character
            int tileInFrontOfFootRow=((footY-1)/Tile.TILE_SIZE);
            int tileInFrontOfFootCol=(footX/Tile.TILE_SIZE)+1;

            if(tileInFrontOfFootCol<World.COLS){
                //if the tile in front of the character contains a block..
                if(World.tiledMap[tileInFrontOfFootRow][tileInFrontOfFootCol] instanceof Block){
                    //..and the character's bounding box intersect the block's one
                    if(boundingBox.intersects(World.tiledMap[tileInFrontOfFootRow][tileInFrontOfFootCol].getBoundingBox())){
                        //push the character away and re-set its position
                        currentX-=DISPLACEMENT;
                        boundingBox.setLocation(currentX, currentY);
                        currentCol=currentX/Tile.TILE_SIZE;
                    }
                }

                if(World.tiledMap[currentRow][currentCol] instanceof Block){
                    //if the tile the character finds them self in contains a block, act like above
                    if(boundingBox.intersects(World.tiledMap[currentRow][currentCol].getBoundingBox())){
                        currentX-=DISPLACEMENT;
                        boundingBox.setLocation(currentX, currentY);
                        currentCol=currentX/Tile.TILE_SIZE;
                    }
                }
            }
```

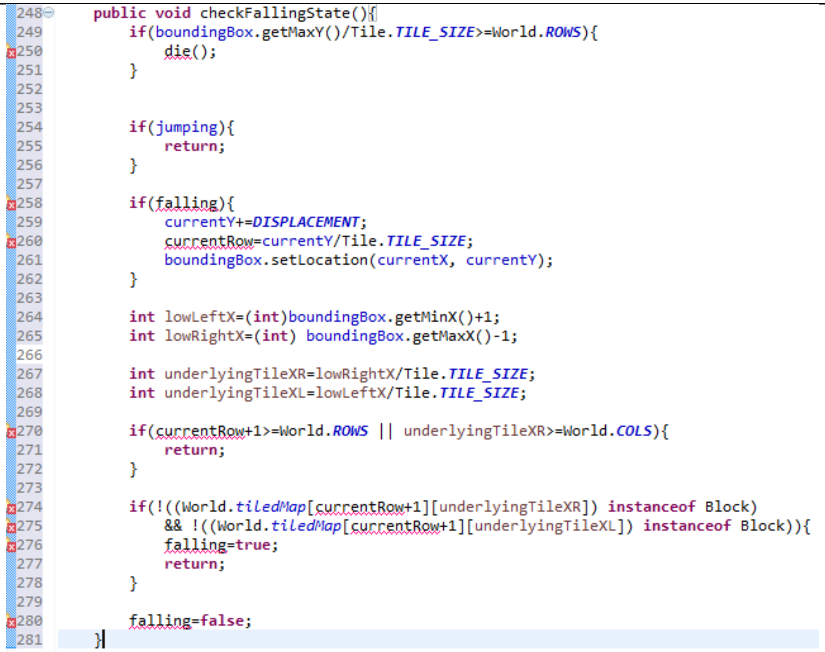
When the character runs right and its bounding box intersects a Block's one, the robot pushed away of an amount of pixels equals to the number of pixels he covers with a single step (the *DISPLACEMENT* value). This way the character's bounding box can never intersect the Block.

The resetting of the character's location when it intersects a bounding box happens within the refresh rate of the game, this means that the player will never see the robot phase into a tile and then get pushed back. This gives the illusion of a solid block that has collisions without relying on an outside collision API.



```
        } else {
            //get the right side of the bounding box
            int footX=(int) boundingBox.getMaxX();

            //get the tile position (in the tiled map)
            //relative to the tile in front of the character
            int tileInFrontOfFootRow=((footY-1)/Tile.TILE_SIZE);
            int tileInFrontOfFootCol=(footX/Tile.TILE_SIZE)-1;

            if(tileInFrontOfFootCol>=0){
                //if the tile in front of the character contains a block..
                if(World.tiledMap[tileInFrontOfFootRow][tileInFrontOfFootCol] instanceof Block){
                    //..and the character's bounding box intersect the block's one
                    if(boundingBox.intersects(World.tiledMap[tileInFrontOfFootRow][tileInFrontOfFootCol].getBoundingBox())){
                        //push the character away and re-set its position
                        currentX+=DISPLACEMENT;
                        boundingBox.setLocation(currentX, currentY);
                        currentCol=currentX/Tile.TILE_SIZE;
                    }
                }

                if(World.tiledMap[currentRow][currentCol] instanceof Block){
                    //if the tile the character finds themself in contains a block, act like above
                    if(boundingBox.intersects(World.tiledMap[currentRow][currentCol].getBoundingBox())){
                        currentX+=DISPLACEMENT;
                        boundingBox.setLocation(currentX, currentY);
                        currentCol=currentX/Tile.TILE_SIZE;
                    }
                }
            }
        }
    }
```

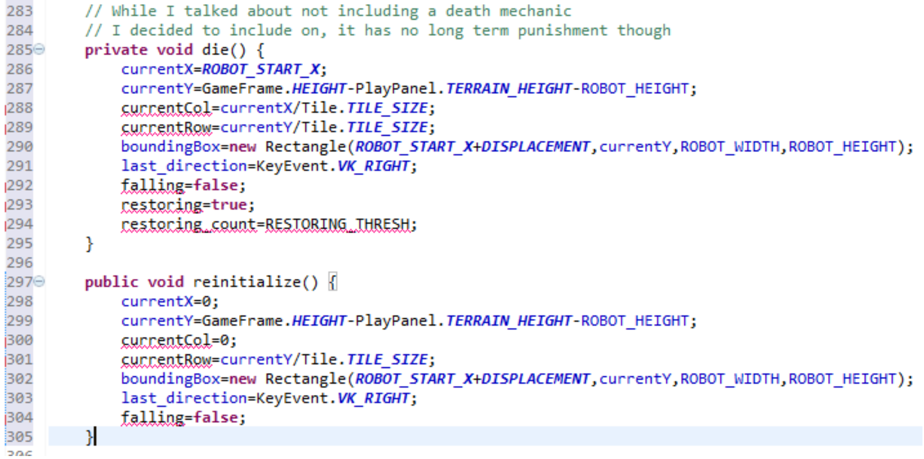
The condition is then repeated and modified for collisions when the character intersects a block when moving to the left.



```
248    public void checkFallingState(){
249        if(boundingBox.getMaxY()/Tile.TILE_SIZE>=World.ROWS){
250            die();
251        }
252
253
254        if(jumping){
255            return;
256        }
257
258        if(falling){
259            currentY+=DISPLACEMENT;
260            currentRow=currentY/Tile.TILE_SIZE;
261            boundingBox.setLocation(currentX, currentY);
262        }
263
264        int lowLeftX=(int)boundingBox.getMinX()+1;
265        int lowRightX=(int) boundingBox.getMaxX()-1;
266
267        int underlyingTileXR=lowRightX/Tile.TILE_SIZE;
268        int underlyingTileXL=lowLeftX/Tile.TILE_SIZE;
269
270        if(currentRow+1>=World.ROWS || underlyingTileXR>=World.COLS){
271            return;
272        }
273
274        if(!((World.tiledMap[currentRow+1][underlyingTileXR]) instanceof Block)
275            && !((World.tiledMap[currentRow+1][underlyingTileXL]) instanceof Block)){
276            falling=true;
277            return;
278        }
279
280        falling=false;
281    }
```

In this version, there are a few actions that need to be checked when the character is falling. *checkFallingState()* is the method that does these checks. Ignoring lines 248-252 (that will be explained later), the lines 258 to 262 ensure that the robot's boundingBox is updated even when in the air, this being even more important as collisions now exist in this version.

Lines 264 to 272 are used for tile referencing and the last lines are used to set the *falling* Boolean to false. This means that the robot is not falling when it is one space above the tile, giving the impression that it can stand on the blocks.

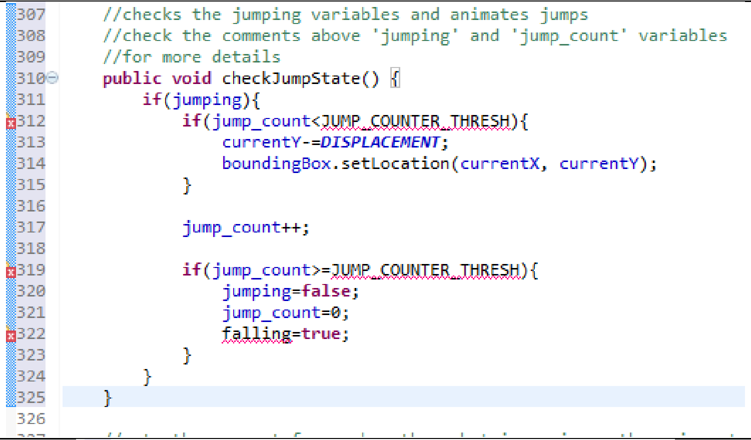


```
283    // While I talked about not including a death mechanic
284    // I decided to include on, it has no long term punishment though
285    private void die() {
286        currentX=ROBOT_START_X;
287        currentY=GameFrame.HEIGHT-PlayPanel.TERRAIN_HEIGHT-ROBOT_HEIGHT;
288        currentCol=currentX/Tile.TILE_SIZE;
289        currentRow=currentY/Tile.TILE_SIZE;
290        boundingBox=new Rectangle(ROBOT_START_X+DISPLACEMENT,currentY,ROBOT_WIDTH,ROBOT_HEIGHT);
291        last_direction=KeyEvent.VK_RIGHT;
292        falling=false;
293        restoring=true;
294        restoring_count=RESTORING_THRESH;
295    }
296
297    public void reinitialize() {
298        currentX=0;
299        currentY=GameFrame.HEIGHT-PlayPanel.TERRAIN_HEIGHT-ROBOT_HEIGHT;
300        currentCol=0;
301        currentRow=currentY/Tile.TILE_SIZE;
302        boundingBox=new Rectangle(ROBOT_START_X+DISPLACEMENT,currentY,ROBOT_WIDTH,ROBOT_HEIGHT);
303        last_direction=KeyEvent.VK_RIGHT;
304        falling=false;
305    }
```

Due to issues arising of the difficultly of my game, I was hard pressed to find ways to add complexity to the game that didn't draw too much attention away from the obstacles. My first response is to tune up the difficulty of the platforming.

When faced with the issue of platforming I had to choices, difficult level design to avoid the player getting trapped, or to add a respawn mechanic. This is when *die()* and *reinitialize()* come in.

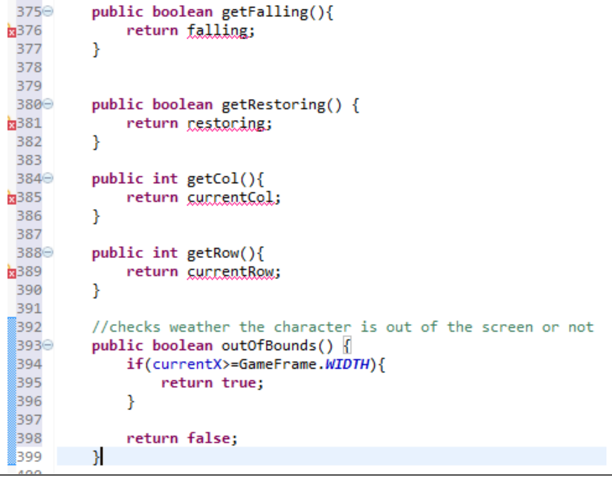
In this version, if the character touches the bottom of the level, it will stop falling and reinitialize in a pre-determined space, allowing the user an unlimited number of tries.



```
    //checks the jumping variables and animates jumps
    //check the comments above 'jumping' and 'jump_count' variables
    //for more details
    public void checkJumpState() {
        if(jumping){
            if(jump_count<JUMP_COUNTER_THRESH){
                currentY-=DISPLACEMENT;
                boundingBox.setLocation(currentX, currentY);
            }

            jump_count++;

            if(jump_count>=JUMP_COUNTER_THRESH){
                jumping=false;
                jump_count=0;
                falling=true;
            }
        }
    }
```

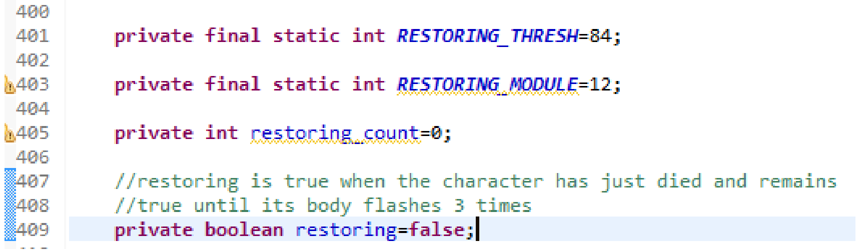
The JUMP_COUNTER_THRESH is an internal measurement of how long the robot will be in a jump state for. As an effect of this, the higher the count, the longer the jump duration, this leads to the robot jumping higher.

After testing with some values. Any smaller than 20 and the robot begins to fail to jump over blocks, any higher and the whole process starts to look ridiculous with the robot leaping bounds with no effort. As such I kept the value as 20, as it allows levels to be tricky.



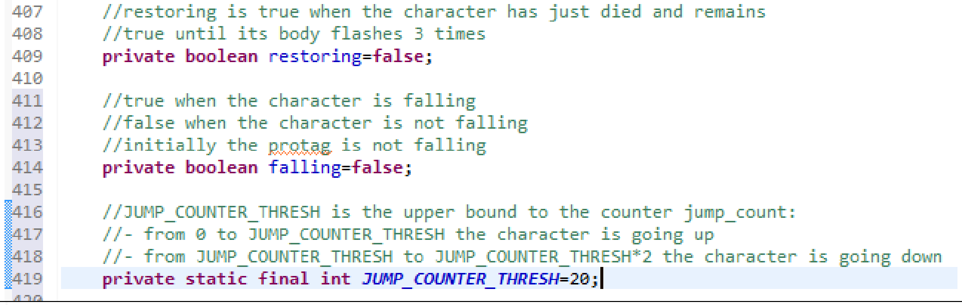
```
    public boolean getFalling(){
        return falling;
    }


    public boolean getRestoring() {
        return restoring;
    }

    public int getCol(){
        return currentCol;
    }

    public int getRow(){
        return currentRow;
    }

    //checks weather the character is out of the screen or not
    public boolean outOfBounds() {
        if(currentX>=GameFrame.WIDTH){
            return true;
        }

        return false;
    }
```

As the biggest Class, **Robot** ends with many variables being declared. Starting from the top: *falling* is used when the robot jumps and is a fundamental boolean used in vertical block collisions. *Restoring* is another boolean that is used by the respawn mechanic, ensuring that death isn't permanent. The *getCol and getRow* values are used with block collisions, allowing the **tile and robot** classes to reference each other. Finally, the *outOfBounds* Boolean is used to determine when the level needs to be changed to the next incremented level.



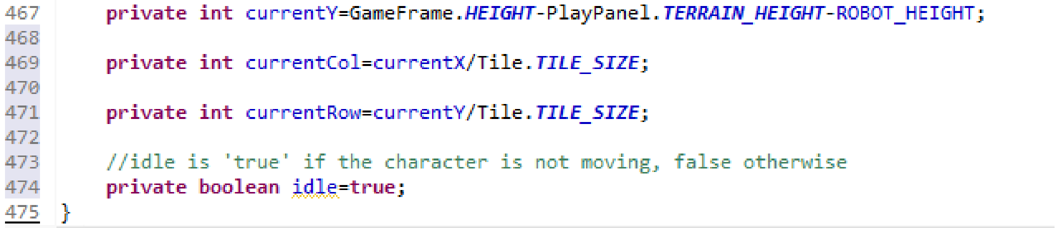
```

    private final static int RESTORING_THRESH=84;

    private final static int RESTORING_MODULE=12;

    private int restoring_count=0;

    //restoring is true when the character has just died and remains
    //true until its body flashes 3 times
    private boolean restoring=false;
```

These variables are used in the restoring method, it mimics the respawn and damage system used in other platforms (ones that do not feature instant death). It gives the robot an iconic three flashes after respawn.



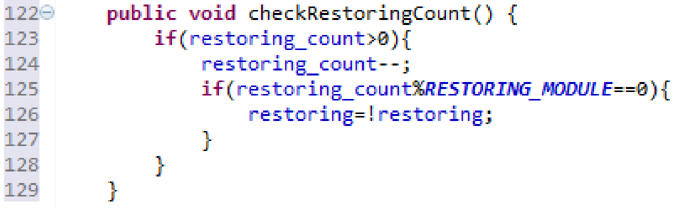
```
    //restoring is true when the character has just died and remains
    //true until its body flashes 3 times
    private boolean restoring=false;

    //true when the character is falling
    //false when the character is not falling
    //initially the protag is not falling
    private boolean falling=false;

    //JUMP_COUNTER_THRESH is the upper bound to the counter jump_count:
    //- from 0 to JUMP_COUNTER_THRESH the character is going up
    //- from JUMP_COUNTER_THRESH to JUMP_COUNTER_THRESH*2 the character is going down
    private static final int JUMP_COUNTER_THRESH=20;
```

These booleans determine if the robot is in a *falling* or *restoring* state. The logic for both states have already been defined. The value for the JUMP_COUNTER_THRESH is also declared here. I have already explained why the threshold is set at 20.



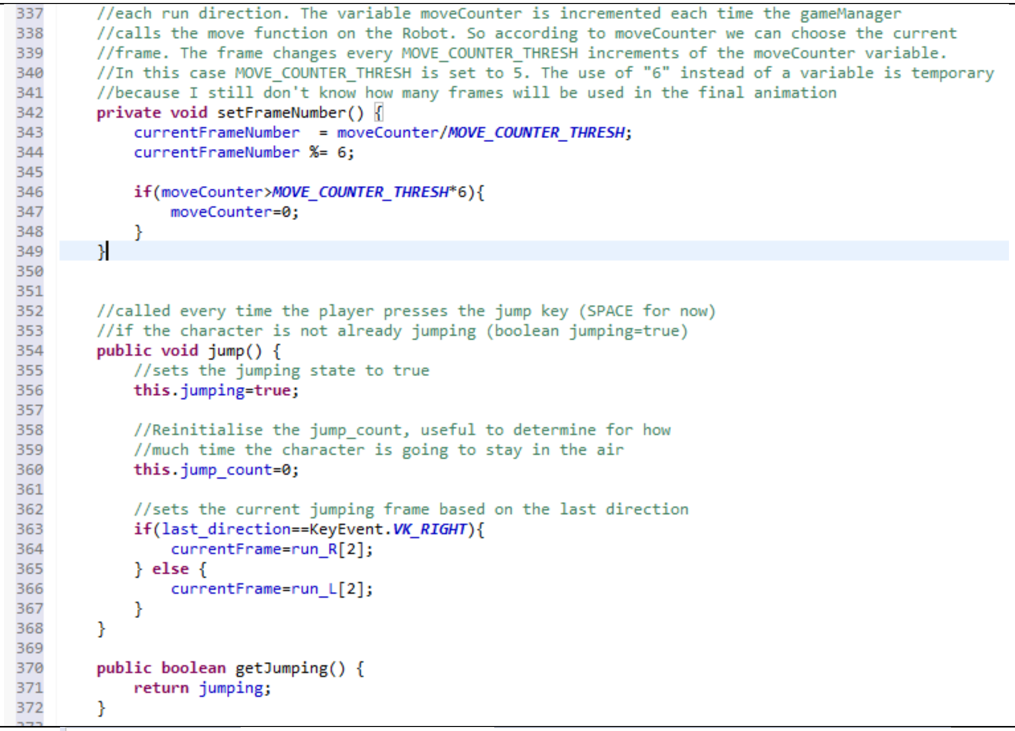
```
    private int currentY=GameFrame.HEIGHT-PlayPanel.TERRAIN_HEIGHT-ROBOT_HEIGHT;

    private int currentCol=currentX/Tile.TILE_SIZE;

    private int currentRow=currentY/Tile.TILE_SIZE;

    //idle is 'true' if the character is not moving, false otherwise
    private boolean idle=true;
}
```

The last declarations are used in the block collisions. They help covert the locations on the *tiledMap* into x and y coordinates on the array that the robot operates on when moving.



```
    public void checkRestoringCount() {
        if(restoring_count>0){
            restoring_count--;
            if(restoring_count%RESTORING_MODULE==0){
                restoring=!restoring;
            }
        }
    }
```

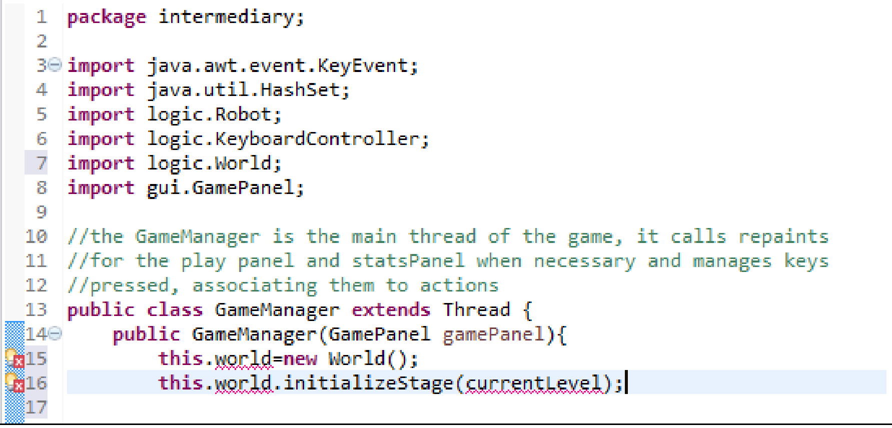
After creating the death system, I needed to go back and create a method for the restoring mechanic. This method allows the user to be in a restoring state for a given period and then leaves the state.



```
    //each run direction. The variable moveCounter is incremented each time the gameManager
    //calls the move function on the Robot. So according to moveCounter we can choose the current
    //frame. The frame changes every MOVE_COUNTER_THRESH increments of the moveCounter variable.
    //In this case MOVE_COUNTER_THRESH is set to 5. The use of "6" instead of a variable is temporary
    //because I still don't know how many frames will be used in the final animation
    private void setFrameNumber() {
        currentFrameNumber  = moveCounter/MOVE_COUNTER_THRESH;
        currentFrameNumber %= 6;

        if(moveCounter>MOVE_COUNTER_THRESH*6){
            moveCounter=0;
        }
    }


    //called every time the player presses the jump key (SPACE for now)
    //if the character is not already jumping (boolean jumping=true)
    public void jump() {
        //sets the jumping state to true
        this.jumping=true;

        //Reinitialise the jump_count, useful to determine for how
        //much time the character is going to stay in the air
        this.jump_count=0;

        //sets the current jumping frame based on the last direction
        if(last_direction==KeyEvent.VK_RIGHT){
            currentFrame=run_R[2];
        } else {
            currentFrame=run_L[2];
        }
    }

    public boolean getJumping() {
        return jumping;
    }
```

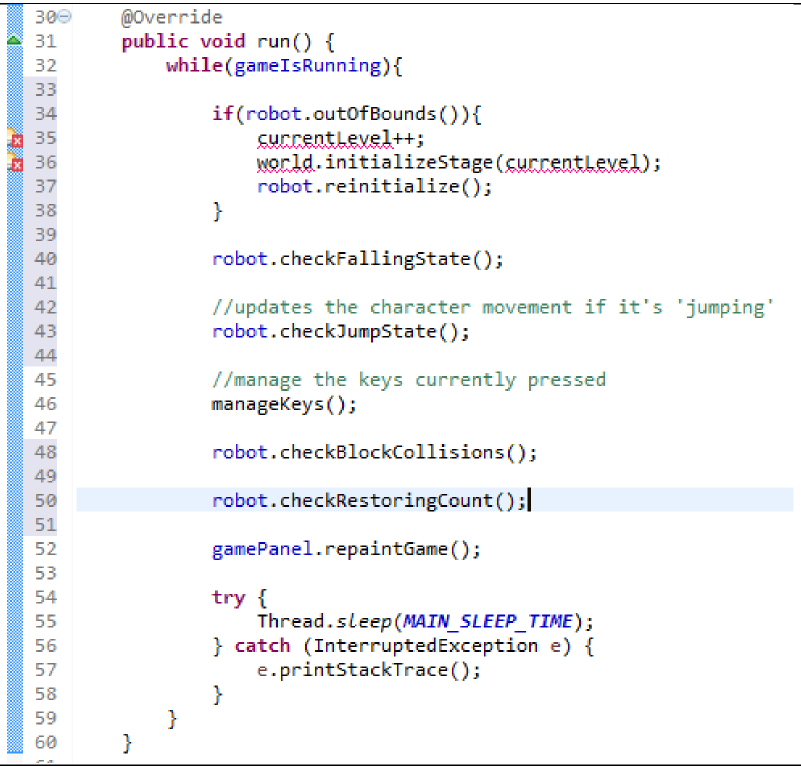
Finally, I have gone back and updated the *setFrameNumber* method. In version 2, the method consists seventeen lines that make up an inefficient selection of cases. After reviewing the conditions and writing down the solution algebraically, I was able to replace most of the repetitive code with a condition that will change based on variables.

With the **Robot** class beginning to use more and more lines, it becomes increasingly important to tackle redundancy and reduce the line size. This makes navigation, modification and de-bugging easier in the long run and is worth the time it takes to review code that could potentially be shortened.



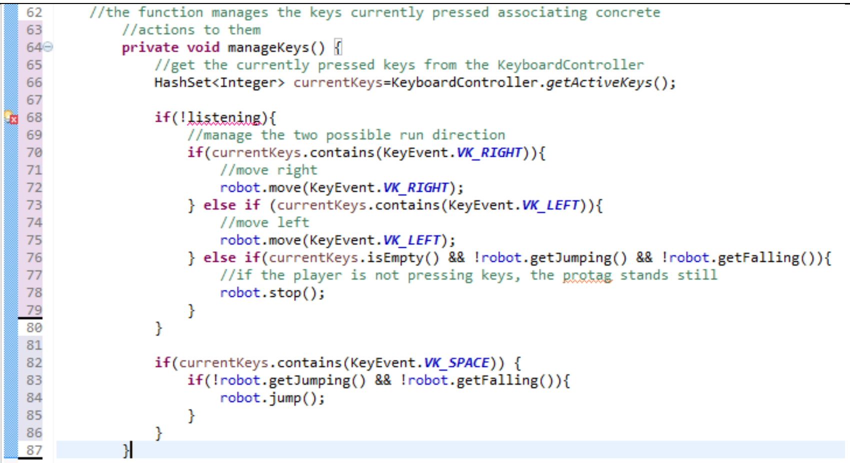
```
package intermediary;

import java.awt.event.KeyEvent;
import java.util.HashSet;
import logic.Robot;
import logic.KeyboardController;
import logic.World;
import gui.GamePanel;

//the GameManager is the main thread of the game, it calls repaints
//for the play panel and statsPanel when necessary and manages keys
//pressed, associating them to actions
public class GameManager extends Thread {
    public GameManager(GamePanel gamePanel){
        this.world=new World();
        this.world.initializeStage(currentLevel);

```

Finally, I have to up the **GameManager** class to run the new environmental additions to the project. The first step is to include the level mechanic into the manager. It will need to get levels from the world in order to implement them.



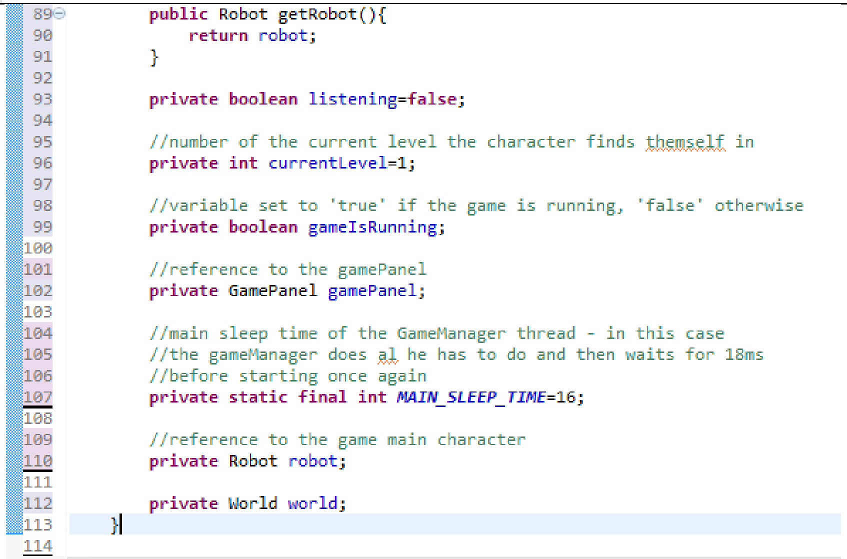
```
    @Override
    public void run() {
        while(gameIsRunning){

            if(robot.outOfBounds()){
                currentLevel++;
                world.initializeStage(currentLevel);
                robot.reinitialize();
            }

            robot.checkFallingState();

            //updates the character movement if it's 'jumping'
            robot.checkJumpState();

            //manage the keys currently pressed
            manageKeys();

            robot.checkBlockCollisions();

            robot.checkRestoringCount();

            gamePanel.repaintGame();

            try {
                Thread.sleep(MAIN_SLEEP_TIME);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

Now, while the game is running, if the robot hits the end of the level (as in the end of the screen) the *outOfBounds* boolean becomes true. When this happens, the value of the current level is incremented and the level is initialized with the rules for the next level. The robot is then reinitialized back onto the new level.



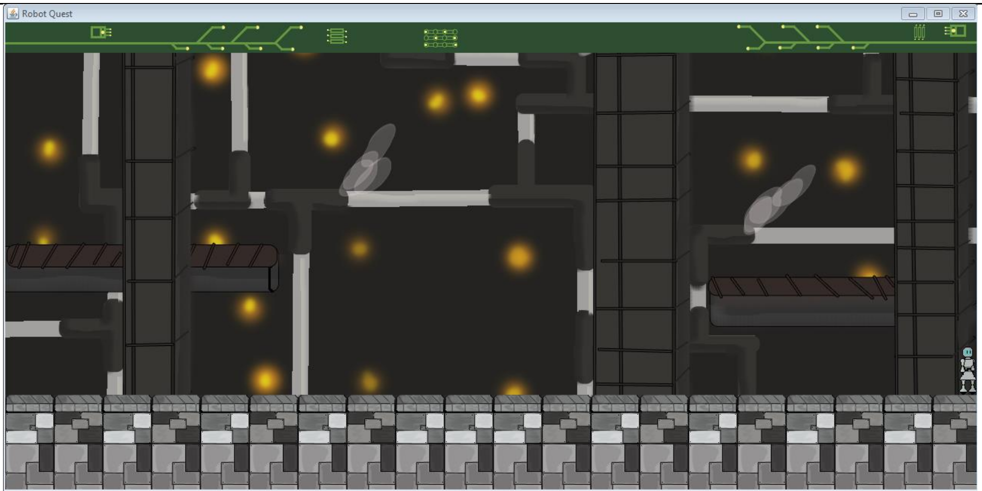
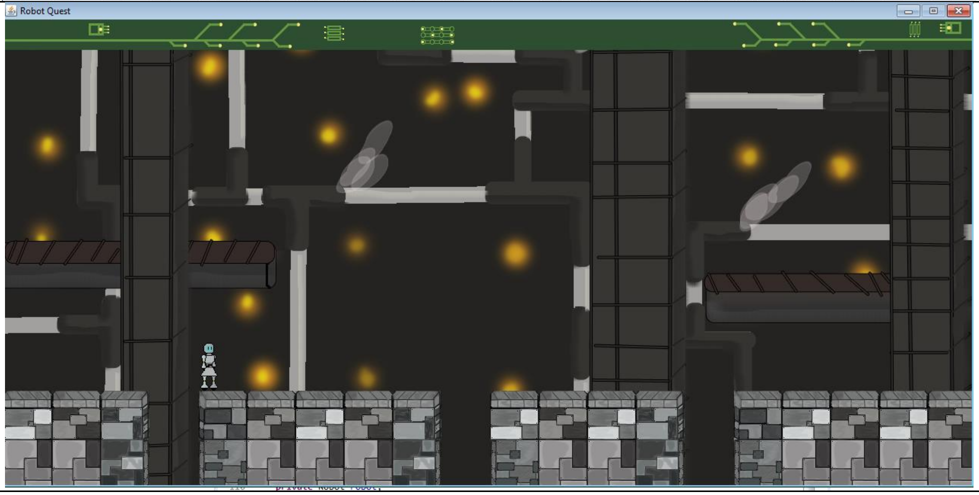
```
//the function manages the keys currently pressed associating concrete
    //actions to them
    private void manageKeys() {
        //get the currently pressed keys from the KeyboardController
        HashSet<Integer> currentKeys=KeyboardController.getActiveKeys();

        if(!listening){
            //manage the two possible run direction
            if(currentKeys.contains(KeyEvent.VK_RIGHT)){
                //move right
                robot.move(KeyEvent.VK_RIGHT);
            } else if (currentKeys.contains(KeyEvent.VK_LEFT)){
                //move left
                robot.move(KeyEvent.VK_LEFT);
            } else if(currentKeys.isEmpty() && !robot.getJumping() && !robot.getFalling()){
                //if the player is not pressing keys, the protag stands still
                robot.stop();
            }
        }

        if(currentKeys.contains(KeyEvent.VK_SPACE)) {
            if(!robot.getJumping() && !robot.getFalling()){
                robot.jump();
            }
        }
    }
```

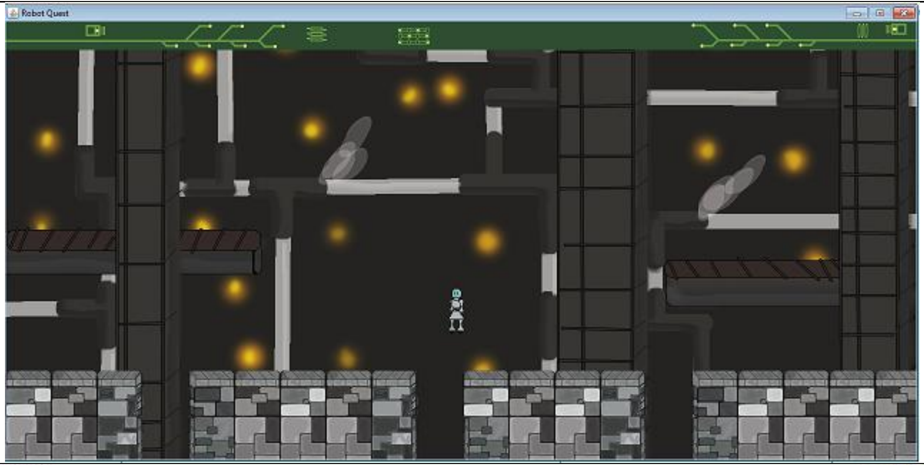
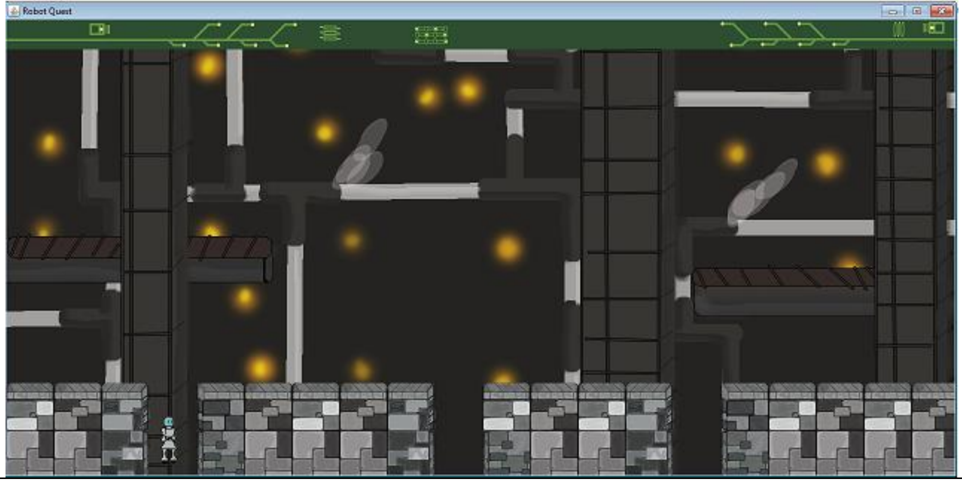
The listening mechanic has been added to allow my new method of jumping to work with the keyboard controller. This is important, as it disables the ability to double jump. The player and not start the jumping process until the last jump (both the jumping and falling stages) has been completed.

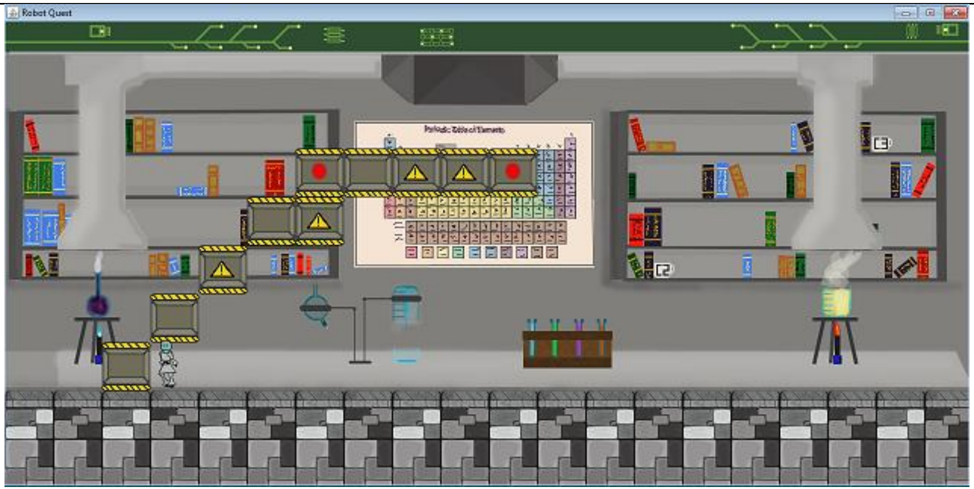


```
    public Robot getRobot(){
        return robot;
    }

    private boolean listening=false;

    //number of the current level the character finds themself in
    private int currentLevel=1;

    //variable set to 'true' if the game is running, 'false' otherwise
    private boolean gameIsRunning;

    //reference to the gamePanel
    private GamePanel gamePanel;

    //main sleep time of the GameManager thread - in this case
    //the gameManager does al he has to do and then waits for 18ms
    //before starting once again
    private static final int MAIN_SLEEP_TIME=16;

    //reference to the game main character
    private Robot robot;

    private World world;
}
```

Finally, the variables are declared. The boolean *listening* is used with *manageKeys()* to regulate jumping. The integer *currentLevel* is used to get information about the levels that will be fetched when the robot reaches the end of the current level.

	The level has been created, there is my first background set and two layers of tiles to create the floor. When I move I do not sink through the tiles.
	Walking to the edge of the screen cases the level to change. This is now the layout for level 2.

	Jumping still works as intended.
	I can't really evidence the death and restoring mechanics with a screenshot, but it did work. I also could not merge into the sides of the blocks, so the collisions are working to.



EDIT: Implementing more levels to test the fetching of the backgrounds worked, the tiles also work to prevent the robot from entering a fall state and fill force it into a fall state when faced with upwards collisions.

Review

To balance the lack of modules implemented in version two, version three is massive. While everything it does can be easily summed up as ‘implementing the environment’, this is actually a huge part of the game, shown by the empty space the robot was tested in compared to the tiled map and backgrounds shown in version 3. This is where the actual product starts to sway away from my original design. This was due to lack of expertise and a difficulty finding ways of implementing designs. This led me to look at existing methods that I could implement, like the common level transition of simply touching the end of the screen. This is a massive difference compared to my battery transition system, which was too complex for me to implement. While I still hold the belief that being able to choose the order of the levels you played would have been a great feature to have, the simplicity of the transition and game progression might sit better with my target audience. The lack of mechanical commitment to changing level also allowed me to implement one designed level across multiple in-game levels, making the whole game feel less cramped.