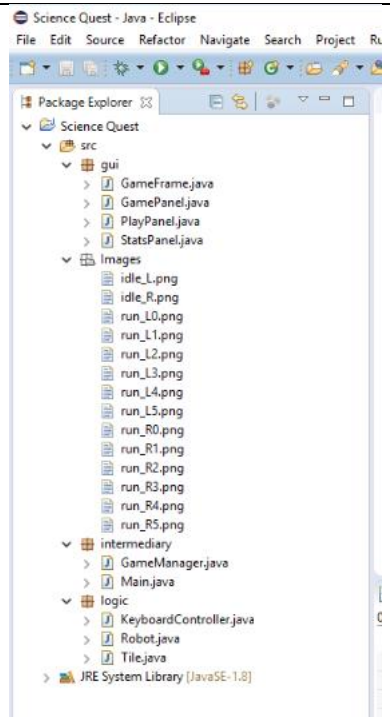
Developing the Coded Solution for Project *Version 2*

Version 2 will focus on finishing the character so that the player will be able to control it to travel across the game. From the last version, this would entitle solving the problem of allowing the character to jump and to match the animation frames of the robot to the sprites I have created.



This is the project as I left in the last version. As this version is only concerned with finishing the character, I believe that there will be no new class creation, only modification of existing classes to allow the character to jump, and thus completing all its required movements for the game.

KeyboardController.java | Robot.java | GameManager.java

```java
package logic;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.HashSet;

//the keyboard controller is the KeyListener of the game, it register
//all the keys currently pressed in the activeKeys HashSet
public class KeyboardController implements KeyListener{

    public KeyboardController(){
        activeKeys=new HashSet<Integer>();
    }

    @Override
    public void keyPressed(KeyEvent e) {
        activeKeys.add(e.getKeyCode());
    }

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

    @Override
    public void keyTyped(KeyEvent e) {
    }

    public static HashSet<Integer> getActiveKeys(){
        return activeKeys;
    }

    private static HashSet<Integer> activeKeys;
```

The first step to get the robot jumping is to allocate a key for input. While In my original algorithm I planned to bind the <up> arrow key to jumping, by binding <space> the user has to use both hands to play the game, it forces the user to become just that bit more physically involved.

With the **KeyboardController** now detecting the <space> key, it is added to the HashSet. Now The **GameManager** checks *activeKeys* continuously. When it finds a <space> key in it, the **GameManager** will tell the robot to jump (if it's not already in the process of jumping).

KeyboardController.java | GameManager.java

```java
51          if(currentKeys.contains(KeyEvent.VK_RIGHT)){
52              //move right
53              robot.move(KeyEvent.VK_RIGHT);
54          } else if (currentKeys.contains(KeyEvent.VK_LEFT)){
55              //move left
56              robot.move(KeyEvent.VK_LEFT);
57          } else if(currentKeys.isEmpty()){
58              //if the player is not pressing keys, the protag stands still
59              robot.stop();
60          }
61
62          if(currentKeys.contains(KeyEvent.VK_SPACE)) {
63              if(!robot.getJumping()){
64                  robot.jump();
65              }
66          }
67
```

As you can see, since the last version I only added a few lines of code to modify the **GameManager** class. This code is how things work from the perspective of the **GameManager**, but you may notice there's no evidence of the actual jumping action here. That's because the **GameManager** does not move the character, it simply tells the character to move.

From the perspective of the Robot, in fact, the first thing that happens is that it receives an "order" from the GameManager, that order is the one you see above at line 64: *robot.jump*().

Robot.java | GameManager.java

```
146     }
147
148
149     public void jump() {
150         this.jumping=true;
151         this.jump_count=0;
152     }
153
154     public boolean getJumping() {
155         return jumping;
156     }
157
158     private int jump_count=0;
159
160     private boolean jumping;
161
```

From the perspective of the **Robot** class the function simply sets the jumping state to true and the *jump_count* to zero. The variable *jump_count* works with a static final integer called JUMP_COUNTER_THRESH: in particular the *jump_count* is incremented every time the main thread (GameManager) calls the *checkState*() function. It will continue to increment this variable until it reaches JUMP_COUNTER_THRESH.
During this period of time, the position of the character on the y-axis gets smaller and smaller. This is the ascending phase of the jump. When the *jump_count* exceeds JUMP_COUNTER_THRESH, while the jump_count keeps going up, the position of the robot on the y-axis is incremented for the character is in the descending phase of the jump. It goes on incrementing until *jump_count* reaches JUMP_COUNTER_THRESH*2, then the jumping Boolean is set to false and the count is reinitialized- the jump process is over.

Robot.java

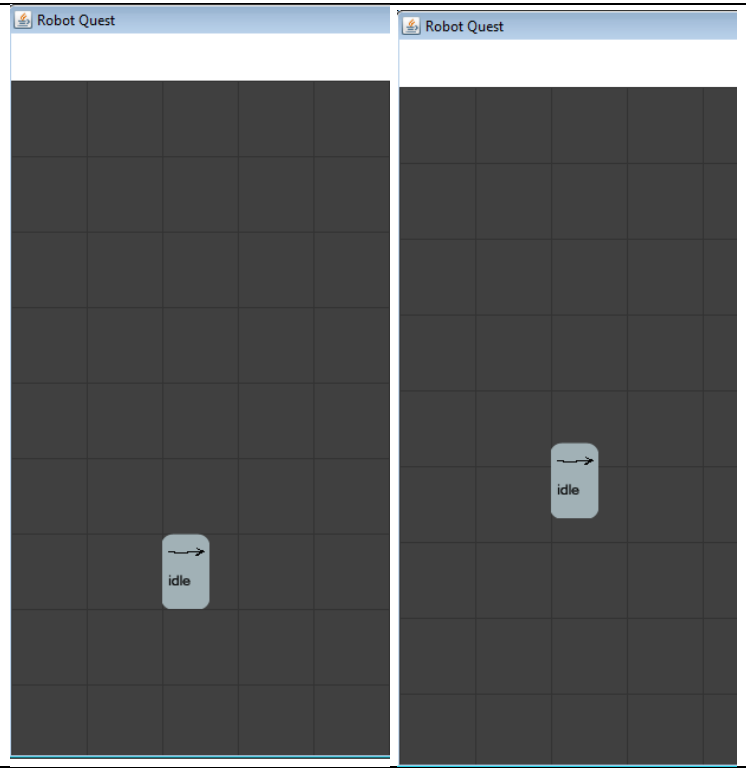
```
82
83                  //update the character's bounding box position
84                  boundingBox.setLocation(currentX, currentY);
85
86                  //change the current frame in animation
87                  if(!jumping){
88                      setFrameNumber();
89                      currentFrame=run_R[currentFrameNumber];
90                  } else {
91                      currentFrame=run_R[0];
92                  }
93
94                  //set the right direction as the last one
95                  last_direction=KeyEvent.VK_RIGHT;
96                  break;
97
98              default:
99                  break;
100         }
101         moveCounter++;
102     }
```

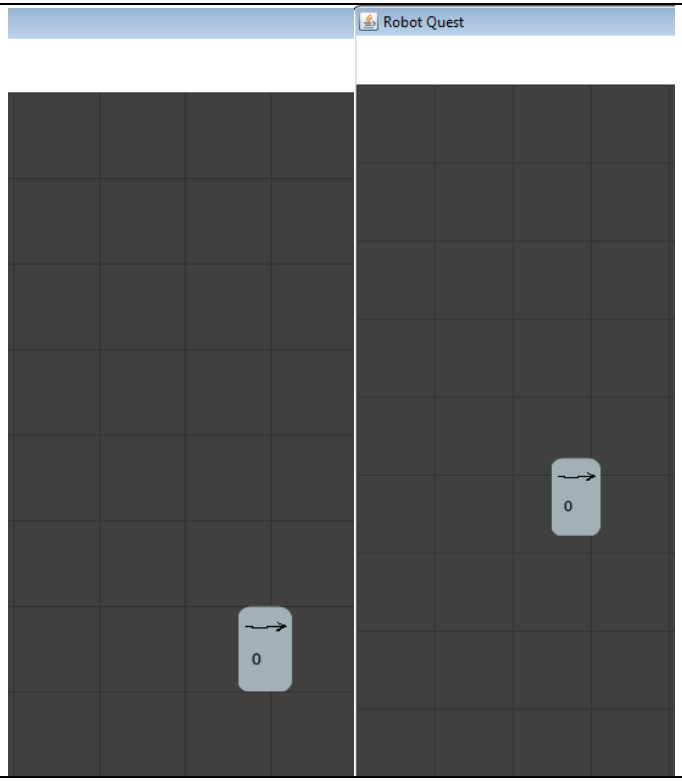
The next modification of the Robot class is the move method. I have included a new case for the jumping animation. If the robot is not in the process of jumping, the current frame will remain untouched and will loop like defined before. But now, if the robot is in the process of jumping, the case set up on line: 90 takes place and the current frame will be set to frame 0 for the duration of the jump. This should make the robot look like in actually goes through the process of jumping rather than just moving up on the Y-axis.

Robot.java

```
104     public void checkState() {
105         if(jumping){
106             if(jump_count<15){
107                 currentY-=6;
108                 boundingBox.setLocation(currentX, currentY);
109             } else {
110                 currentY+=6;
111                 boundingBox.setLocation(currentX, currentY);
112             }
113
114             jump_count++;
115
116             if(jump_count>=30){
117                 jumping=false;
118                 jump_count=0;
119             }
120         }
121     }
122
```

Finally, the boundingBox is updated to allow it to follow the character when jumping (lines 108 and 111). This is critical for traversing platforms and stopping mechanics from being bypassed by jumping (e.g. jumping through walls)

	With the code finished, the next step was black box testing. The first test was jumping from an idle state. Both the jumping mechanic and animation works for both left and right facing directions.

	Currently the left movement animations are not working. All the jumping logic holds up with movement, but animations while that is happening do not exist for any left movements, jumping or not. The animations still transition as intended in positive movement across the X-axis, but not negative, therefore it is clear that any bug or error would be in the left key input logic. Edit: there was an issue where lines 73-79 got deleted in modification. After including them back into the ***VK_LEFT*** case the logic in the **Robot** class, the code now works as intended. With the code working as intended, testing was finished. All the movement of the character and testing finished.

Edit: Since I have already digitalised the sprites for the **Robot** class, I used version 2 to test out the animations frames before implementing them into version 3. This was simple to implement, as the process only requires the graphic to be replaced in the **Images** folder.

Review

Probably the smallest change in any version, version two only concerned itself with the final issues in the characters ability to navigate a level. This meant that only the jumping mechanic was implemented into the version, with the character sprites being an addition after the original development. This was interesting, as version two served more as a testing platform rather than key developmental stage. All types of animation frame allowances (that would be later used as reference in the animation of objects, how much gap between each frame is possible) and movement speeds was tested in this version.