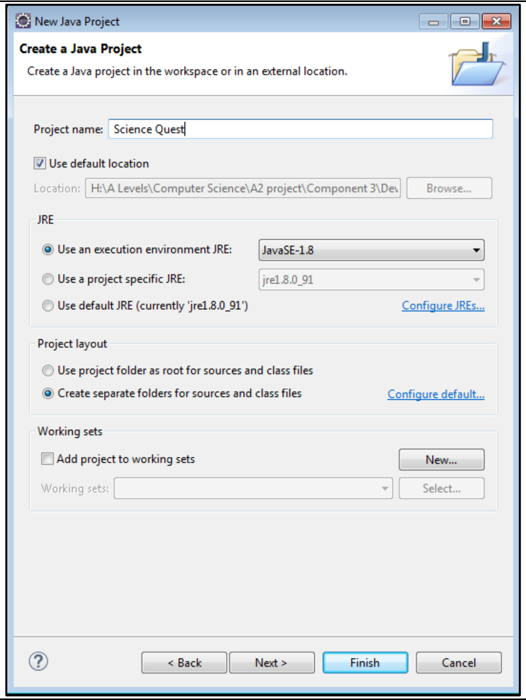
Developing the Coded Solution for Project *Version 1*

Version 1 exists to create the character and allow it to move. To allow this to happen, a basic version of a graphical user interface (GUI) had to also be created to show the character. The next version will include jumping, finishing all the moment aspects of the robot.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: Science Quest

[✓] Use default location

Location: H:\A Levels\Computer Science\A2 project\Component 3\Dev Browse...

JRE

[✓ Use an execution environment JRE:] JavaSE-1.8

Use a project specific JRE: jre1.8.0_91

Use default JRE (currently 'jre1.8.0_91') Configure JREs...

Project layout

Use project folder as root for sources and class files

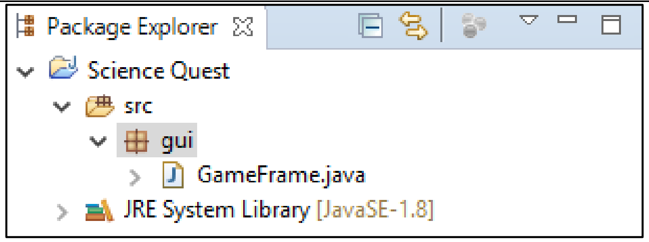
[✓ Create separate folders for sources and class files] Configure default...

Working sets

Add project to working sets New...

Working sets: Select...

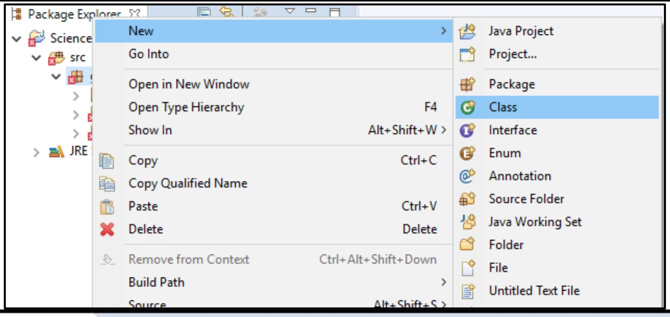
The first thing I did was to make a project. This is the screen for creating a new java project folder in the Eclipse IDE. The project folder will contain all the packages, folders and classes that the program will use. While the IDE makes this process, very straight forward - it is, important that attention is paid when selecting the execution environment and library you plan to use as we are using some of the more complex **APIs** (Application Program Interface) in this project. For this reason, I have chosen the most recent JRE, that being 1.8. The project name 'Science Quest' has been chosen as the final name for the program, as decided by my client.

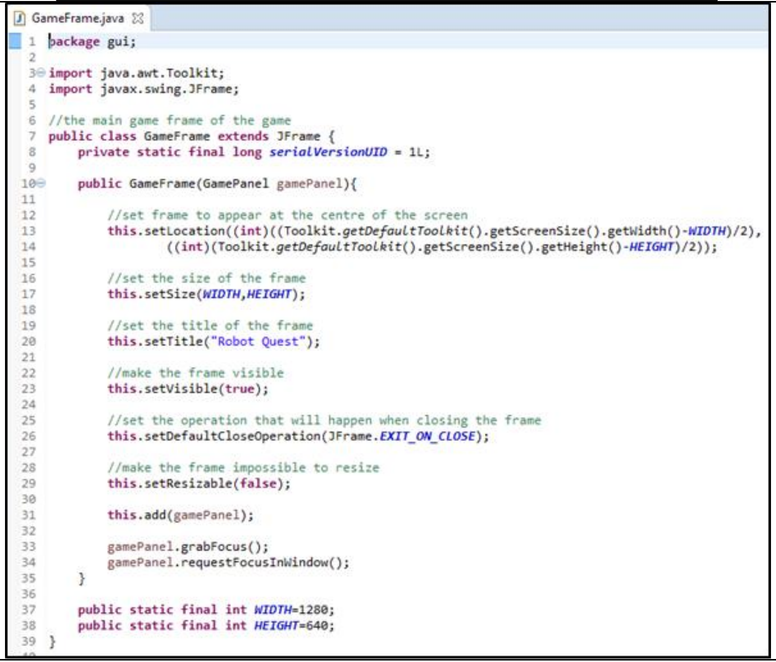


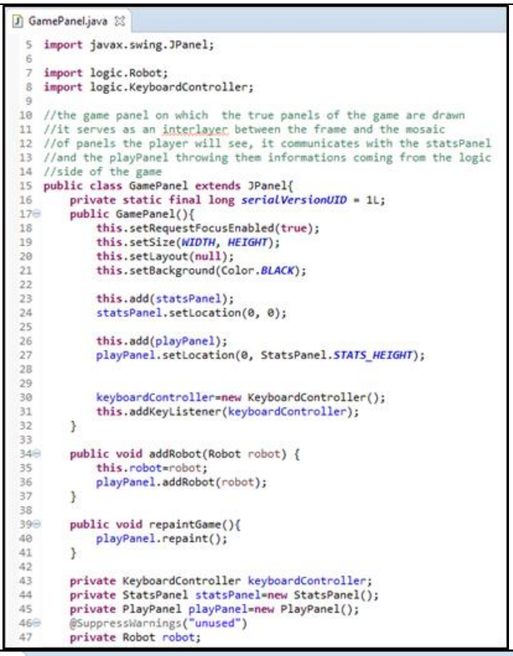
Package Explorer

```
Science Quest
  src
    gui
      GameFrame.java
  JRE System Library [JavaSE-1.8]
```

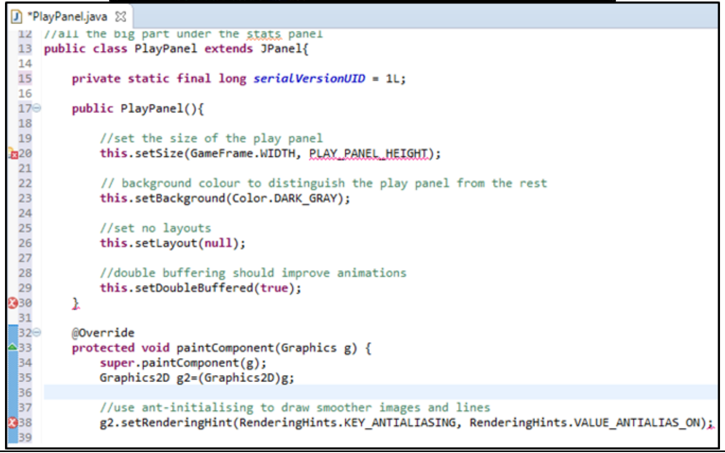
After making the project, the src folder is automatically created by Eclipse. The src will contain all of the projects source folders that contain the .java files. Within the src, I have created the first package **'gui'.** A package is namespace (or area) that organises a set of related classes or interfaces. For the **'gui'** package, all the classes that make up the GUI fall into this category.

 Package Explorer Science src JRE New Go Into Open in New Window Open Type Hierarchy F4 Show In Alt+Shift+W Copy Ctrl+C Copy Qualified Name Paste Ctrl+V Delete Delete Remove from Context Ctrl+Alt+Shift+Down Build Path Source Alt+Shift+S Java Project Project... Package Class Interface Enum Annotation Source Folder Java Working Set Folder File Untitled Text File	To create a new class I selected the package then New>Class. This will lead to a panel similar to the project creation; while the JRE offers additional settings, I did not use any for the creation of any classes. After entering a name, the class was created. This is the process taken for creating any class in the project.

 GameFrame.java
`package gui;`
 	
`import java.awt.Toolkit;`
`import javax.swing.JFrame;`
 	
`//the main game frame of the game`
`public class GameFrame extends JFrame {`
`    private static final long serialVersionUID = 1L;`
 	
`    public GameFrame(GamePanel gamePanel){`
 	
`        //set frame to appear at the centre of the screen`
`        this.setLocation((int)((Toolkit.getDefaultToolkit().getScreenSize().getWidth()-WIDTH)/2),`
`                ((int)(Toolkit.getDefaultToolkit().getScreenSize().getHeight()-HEIGHT)/2));`
 	
`        //set the size of the frame`
`        this.setSize(WIDTH,HEIGHT);`
 	
`        //set the title of the frame`
`        this.setTitle("Robot Quest");`
 	
`        //make the frame visible`
`        this.setVisible(true);`
 	
`        //set the operation that will happen when closing the frame`
`        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
 	
`        //make the frame impossible to resize`
`        this.setResizable(false);`
 	
`        this.add(gamePanel);`
 	
`        gamePanel.grabFocus();`
`        gamePanel.requestFocusInWindow();`
`    }`
 	
`    public static final int WIDTH=1280;`
`    public static final int HEIGHT=640;`
`}`
	I created this first class in the **'gui'** package - **GameFrame**. For my GUI, I have imported **JSwing**; this allows its users to create GUI components, such as a window frame, that are independent of the windowing system for specific operating systems. The class itself is a basic **Jframe** class, creating a single window for my application with a border, title and supports a button component to close the window. I did not include the option to maximise the frame because of the potential for this game to be used on smaller devices that are increasing popular with my target audience, such as tablets and smart phones.

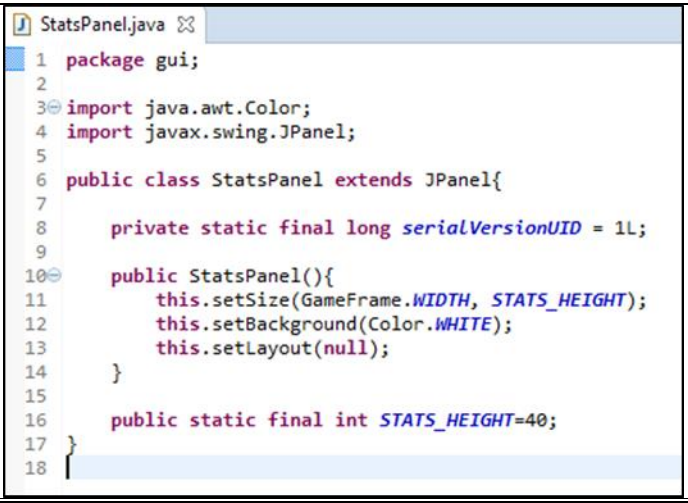

GamePanel.java

```java
import javax.swing.JPanel;

import logic.Robot;
import logic.KeyboardController;

//the game panel on which  the true panels of the game are drawn
//it serves as an interlayer between the frame and the mosaic
//of panels the player will see, it communicates with the statsPanel
//and the playPanel throwing them informations coming from the logic
//side of the game
public class GamePanel extends JPanel{
    private static final long serialVersionUID = 1L;
    public GamePanel(){
        this.setRequestFocusEnabled(true);
        this.setSize(WIDTH, HEIGHT);
        this.setLayout(null);
        this.setBackground(Color.BLACK);

        this.add(statsPanel);
        statsPanel.setLocation(0, 0);

        this.add(playPanel);
        playPanel.setLocation(0, StatsPanel.STATS_HEIGHT);


        keyboardController=new KeyboardController();
        this.addKeyListener(keyboardController);
    }

    public void addRobot(Robot robot) {
        this.robot=robot;
        playPanel.addRobot(robot);
    }

    public void repaintGame(){
        playPanel.repaint();
    }

    private KeyboardController keyboardController;
    private StatsPanel statsPanel=new StatsPanel();
    private PlayPanel playPanel=new PlayPanel();
    @SuppressWarnings("unused")
    private Robot robot;
```


*PlayPanel.java

```java
//all the big part under the stats panel
public class PlayPanel extends JPanel{

    private static final long serialVersionUID = 1L;

    public PlayPanel(){

        //set the size of the play panel
        this.setSize(GameFrame.WIDTH, PLAY_PANEL_HEIGHT);

        // background colour to distinguish the play panel from the rest
        this.setBackground(Color.DARK_GRAY);

        //set no layouts
        this.setLayout(null);

        //double buffering should improve animations
        this.setDoubleBuffered(true);
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2=(Graphics2D)g;

        //use ant-initialising to draw smoother images and lines
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

```

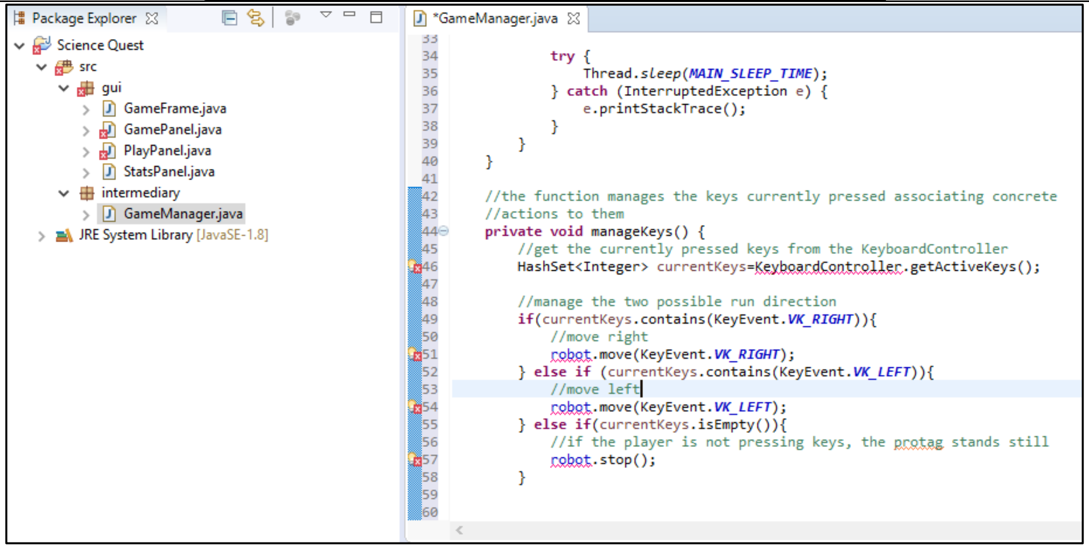
The next two classes in the '**gui**' package are the **GamePanel** and **PlayPanel** classes (as shown below). These are both JPanels, JSwing's lightweight container that is used to group a set of components together. The game panel acts as a sort of canvas, communicating with the other panels to connect the **logic** with the '**gui**'. The **PlayPanel** is where the game itself is drawn and set into motion - it makes up most of what you will see when playing the game.



StatsPanel.java

```java
package gui;

import java.awt.Color;
import javax.swing.JPanel;

public class StatsPanel extends JPanel{

    private static final long serialVersionUID = 1L;

    public StatsPanel(){
        this.setSize(GameFrame.WIDTH, STATS_HEIGHT);
        this.setBackground(Color.WHITE);
        this.setLayout(null);
    }

    public static final int STATS_HEIGHT=40;
}
```

The **StatsPanel** will hold statistical information like the amount of batteries the player has and is located above the **PlayPanel**. At the moment this class is incomplete, as there currently is no statistical information in this version of the game (e.g. the batteries don't exist at this stage). The **StatsPanel** has been included so that the GUI formats properly (it currently is a white strip at the top of the **PlayPanel**).



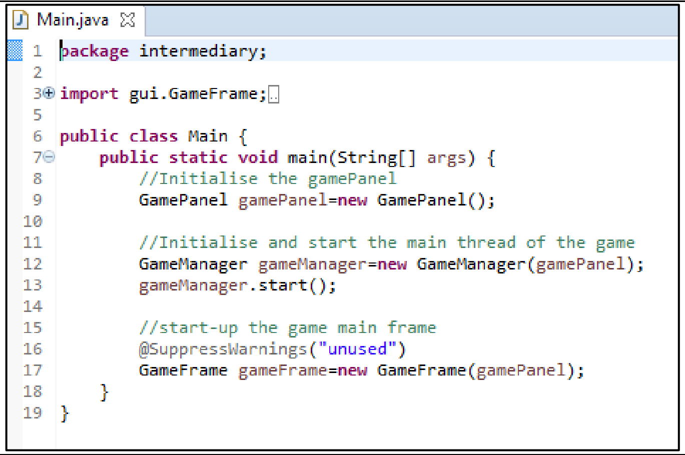
```
Package Explorer
Science Quest
  src
    gui
      GameFrame.java
      GamePanel.java
      PlayPanel.java
      StatsPanel.java
    intermediary
      GameManager.java
  JRE System Library [JavaSE-1.8]
```

*GameManager.java

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

    //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();

        //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()){
            //if the player is not pressing keys, the protag stands still
            robot.stop();
        }
```

With the '**gui**' package complete, I can start development of my next package, the **intermediary** package. This package serves as an intermediary layer linking logical processes with the GUI so that the user can play the game.
The first class in the **intermediary** package is the **GameManager**. This class is the main thread of the game, it calls repaints for the **PlayPanel** and **StatsPanel** when necessary and manages keyboard input. It will operate a sequence of instructions in a loop until the game is over.
As you can see in the run method, the **GameManager** continuously checks what is in the set of currently pressed keys by calling the *manageKeys* private function. This function is simple: if the user is pressing a right key *(KeyEvent.VK_RIGHT)* or a left key *(KeyEvent.VK_LEFT),* it calls the move function on the robot (the class that is the character) object, passing the direction as a parameter. If the player is not pressing

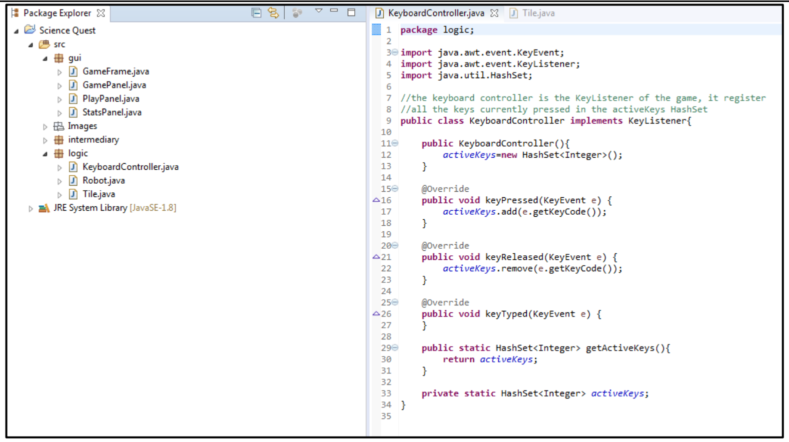
any key, *manageKeys* calls the *stop()* function on the robot, which only serves a graphic purpose turning the current moving frame of the robot into an "idle" frame. After the *managefunction()*, the main cycle of the GameManager (that you can see in the *run()* method) thread repaints the game, making the position changes visible on screen, and then sleeps for 18ms.



Main.java

```java
package intermediary;

import gui.GameFrame;

public class Main {
    public static void main(String[] args) {
        //Initialise the gamePanel
        GamePanel gamePanel=new GamePanel();

        //Initialise and start the main thread of the game
        GameManager gameManager=new GameManager(gamePanel);
        gameManager.start();

        //start-up the game main frame
        @SuppressWarnings("unused")
        GameFrame gameFrame=new GameFrame(gamePanel);
    }
}
```

Next, I created the **Main** class of the game. While I described the **GameManager** as the main thread, the **Main** class deals with the initial start-up of the game. Therefore, while the **GameManager** deals with most of the running of the game, the **Main** class only serves the purpose starting up the application.



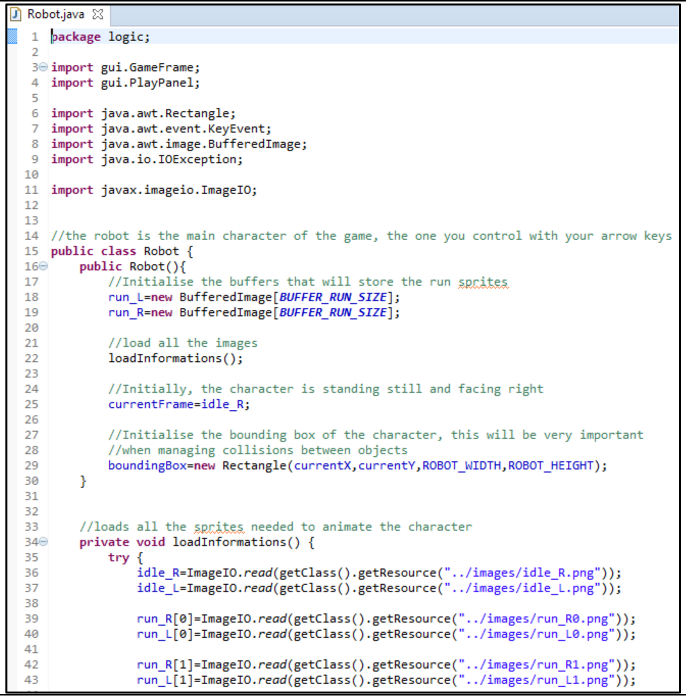
Package Explorer

```
Science Quest
  src
    gui
      GameFrame.java
      GamePanel.java
      PlayPanel.java
      StatsPanel.java
    Images
    intermediary
    logic
      KeyboardController.java
      Robot.java
      Tile.java
  JRE System Library [JavaSE-1.8]
```

KeyboardController.java

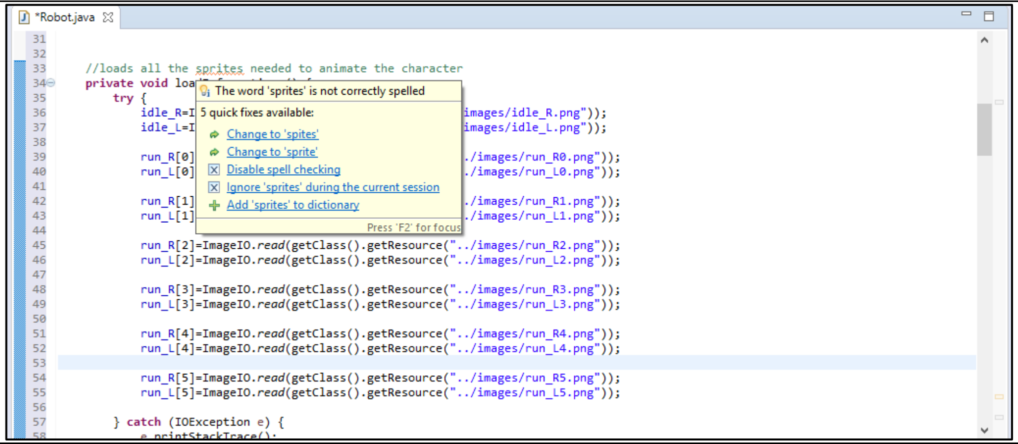
```java
package logic;

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 next package to be created is the **logic** package. This package contains all the logical processes that will run in the background of the game, unseen by the user. The first of the three classes is the **KeyboardController**, it serves as a means of allowing keyboard inputs. The keys you press are stored into the HashSet called "activeKeys ". The HashSet is the best data structure to store this kind of data because it automatically handles duplicates. This removes any worries like having multiple "*LEFT_ARROW_KEY*" characters inside activeKeys, because the *add(element)* function of HashSet will only add the element if not already

	contained in the set. The number of keys that can be used in this application have been limited to allow a more user-friendly experience.
	This is the **Robot** class. This class defines and controls all the movements and interactions of the **Robot** object. The images show the *loadInformations()* function. This fetches the listed images in the **Images** folder to use for the robot. These images will be used for when the robot is both standing (idle) still and walking. The image below shows an issue with the JRE's spellchecker.

Robot.java

```java
package logic;

import gui.GameFrame;
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 robot is the main character of the game, the one you control with your arrow keys
public class Robot {
    public Robot(){
        //Initialise the buffers that will store the run sprites
        run_L=new BufferedImage[BUFFER_RUN_SIZE];
        run_R=new BufferedImage[BUFFER_RUN_SIZE];

        //load all the images
        loadInformations();

        //Initially, the character is standing still and facing right
        currentFrame=idle_R;

        //Initialise the bounding box of the character, this will be very important
        //when managing collisions between objects
        boundingBox=new Rectangle(currentX,currentY,ROBOT_WIDTH,ROBOT_HEIGHT);
    }


    //loads all the sprites needed to animate the character
    private void loadInformations() {
        try {
            idle_R=ImageIO.read(getClass().getResource("../images/idle_R.png"));
            idle_L=ImageIO.read(getClass().getResource("../images/idle_L.png"));

            run_R[0]=ImageIO.read(getClass().getResource("../images/run_R0.png"));
            run_L[0]=ImageIO.read(getClass().getResource("../images/run_L0.png"));

            run_R[1]=ImageIO.read(getClass().getResource("../images/run_R1.png"));
            run_L[1]=ImageIO.read(getClass().getResource("../images/run_L1.png"));
```



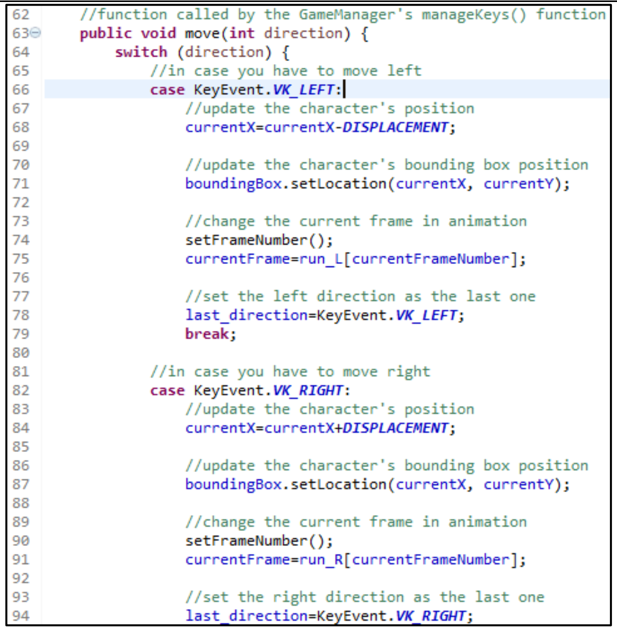
*Robot.java

```
    //loads all the sprites needed to animate the character
    private void loa
        try {
            idle_R=I                                            images/idle_R.png"));
            idle_L=I                                            images/idle_L.png"));

            run_R[0]                                            ./images/run_R0.png"));
            run_L[0]                                            ./images/run_L0.png"));

            run_R[1]                                            ./images/run_R1.png"));
            run_L[1]                                            ./images/run_L1.png"));

            run_R[2]=ImageIO.read(getClass().getResource("../images/run_R2.png"));
            run_L[2]=ImageIO.read(getClass().getResource("../images/run_L2.png"));

            run_R[3]=ImageIO.read(getClass().getResource("../images/run_R3.png"));
            run_L[3]=ImageIO.read(getClass().getResource("../images/run_L3.png"));

            run_R[4]=ImageIO.read(getClass().getResource("../images/run_R4.png"));
            run_L[4]=ImageIO.read(getClass().getResource("../images/run_L4.png"));

            run_R[5]=ImageIO.read(getClass().getResource("../images/run_R5.png"));
            run_L[5]=ImageIO.read(getClass().getResource("../images/run_L5.png"));

        } catch (IOException e) {
            e.printStackTrace();
```

The word 'sprites' is not correctly spelled
5 quick fixes available:
- Change to 'spites'
- Change to 'sprite'
- Disable spell checking
- Ignore 'sprites' during the current session
- Add 'sprites' to dictionary

Press 'F2' for focus

My issue with the IDE is that its spellchecker does not know some key game development terminology such as sprites, and gives me no way of ignoring the message.

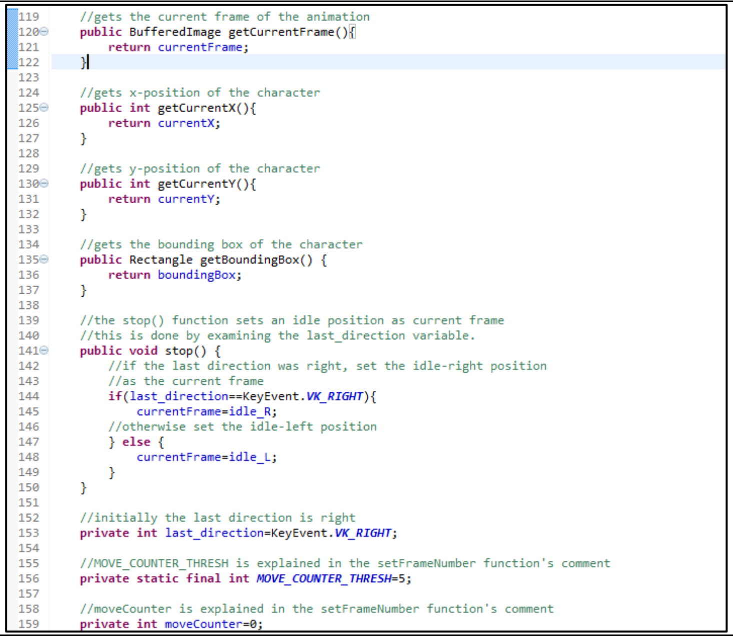


```
    //function called by the GameManager's manageKeys() function
    public void move(int direction) {
        switch (direction) {
            //in case you have to move left
            case KeyEvent.VK_LEFT:
                //update the character's position
                currentX=currentX-DISPLACEMENT;

                //update the character's bounding box position
                boundingBox.setLocation(currentX, currentY);

                //change the current frame in animation
                setFrameNumber();
                currentFrame=run_L[currentFrameNumber];

                //set the left direction as the last one
                last_direction=KeyEvent.VK_LEFT;
                break;

            //in case you have to move right
            case KeyEvent.VK_RIGHT:
                //update the character's position
                currentX=currentX+DISPLACEMENT;

                //update the character's bounding box position
                boundingBox.setLocation(currentX, currentY);

                //change the current frame in animation
                setFrameNumber();
                currentFrame=run_R[currentFrameNumber];

                //set the right direction as the last one
                last_direction=KeyEvent.VK_RIGHT;
```

The *move()* function to moves the robot and animates it. When the robot moves, three different things happen.

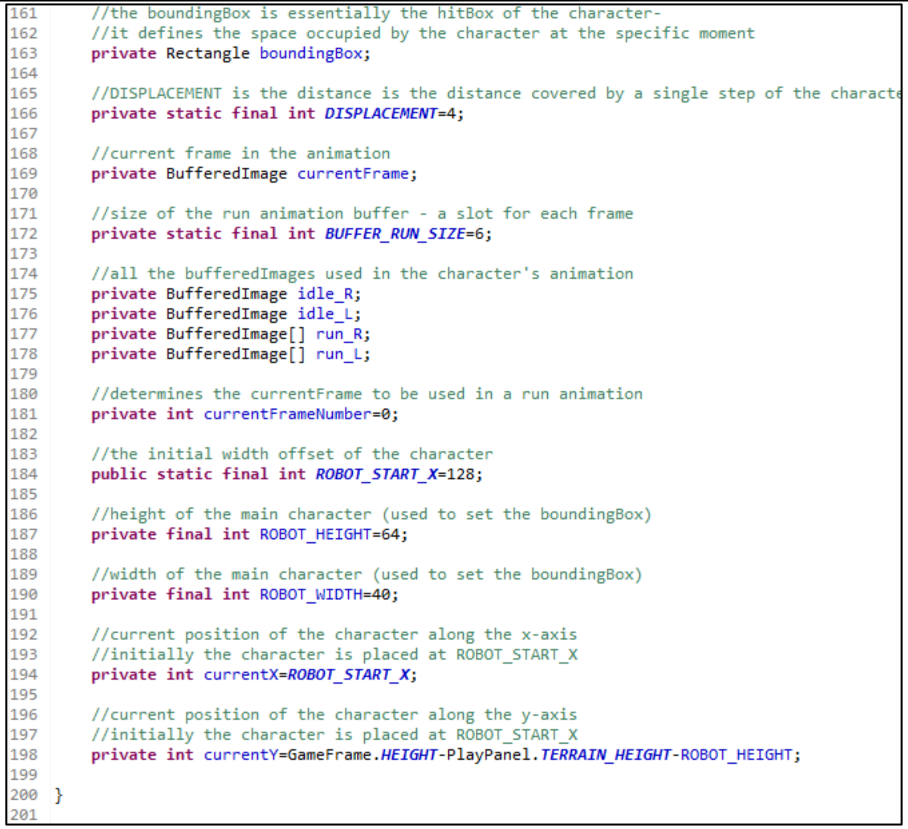
1. The object moves along the X-axis of the array.
2. The current frame (image) of the robot is changed to animate it.
3. The object's bounding box is relocated to its new position.

The *move()* has two cases-if the right or left key has been pressed. Each key will cause the characters position to update, leading the X displacement to either be to the left or the right of the character's last position. The robot's current frame either will be incremented by one or changed to the first frame of the robot facing the other side.
The final is the **boundingBox**. A **boundingBox** is just a rectangle, built around the character's sprite that moves with the character following them anywhere like a shadow. The *getBoundBox()* function ensures that the box constantly updates to stay around the character.



```java
    //gets the current frame of the animation
    public BufferedImage getCurrentFrame(){
        return currentFrame;
    }

    //gets x-position of the character
    public int getCurrentX(){
        return currentX;
    }

    //gets y-position of the character
    public int getCurrentY(){
        return currentY;
    }

    //gets the bounding box of the character
    public Rectangle getBoundingBox() {
        return boundingBox;
    }

    //the stop() function sets an idle position as current frame
    //this is done by examining the last_direction variable.
    public void stop() {
        //if the last direction was right, set the idle-right position
        //as the current frame
        if(last_direction==KeyEvent.VK_RIGHT){
            currentFrame=idle_R;
        //otherwise set the idle-left position
        } else {
            currentFrame=idle_L;
        }
    }

    //initially the last direction is right
    private int last_direction=KeyEvent.VK_RIGHT;

    //MOVE_COUNTER_THRESH is explained in the setFrameNumber function's comment
    private static final int MOVE_COUNTER_THRESH=5;

    //moveCounter is explained in the setFrameNumber function's comment
    private int moveCounter=0;
```

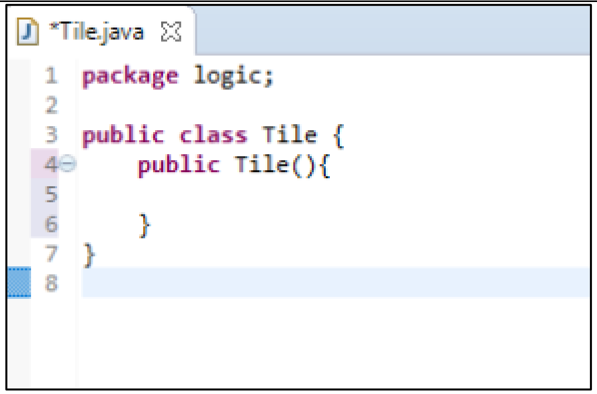
The box is important because in later versions of the game, it will play a key feature for object collision.



```
161     //the boundingBox is essentially the hitBox of the character-
162     //it defines the space occupied by the character at the specific moment
163     private Rectangle boundingBox;
164 
165     //DISPLACEMENT is the distance is the distance covered by a single step of the characte
166     private static final int DISPLACEMENT=4;
167 
168     //current frame in the animation
169     private BufferedImage currentFrame;
170 
171     //size of the run animation buffer - a slot for each frame
172     private static final int BUFFER_RUN_SIZE=6;
173 
174     //all the bufferedImages used in the character's animation
175     private BufferedImage idle_R;
176     private BufferedImage idle_L;
177     private BufferedImage[] run_R;
178     private BufferedImage[] run_L;
179 
180     //determines the currentFrame to be used in a run animation
181     private int currentFrameNumber=0;
182 
183     //the initial width offset of the character
184     public static final int ROBOT_START_X=128;
185 
186     //height of the main character (used to set the boundingBox)
187     private final int ROBOT_HEIGHT=64;
188 
189     //width of the main character (used to set the boundingBox)
190     private final int ROBOT_WIDTH=40;
191 
192     //current position of the character along the x-axis
193     //initially the character is placed at ROBOT_START_X
194     private int currentX=ROBOT_START_X;
195 
196     //current position of the character along the y-axis
197     //initially the character is placed at ROBOT_START_X
198     private int currentY=GameFrame.HEIGHT-PlayPanel.TERRAIN_HEIGHT-ROBOT_HEIGHT;
199 
200 }
201 
```

On lines 184 to 198, you can see functions that record the dimensions of the images that make the robot and its starting locations. This is used to set the **boundingBox**, allowing it to border the robot at the start of the game and any time after that. This allows constant interactions in later versions. Scattered around the thread is also lines for the location of the robot, e.g. line 184, 194 and 198. The robot's height, 64 pixels has been set so it is as tall as a single tile (that is 64x64), but the width is less at 40, as the robot is not as wide as a tile, allowing for the proportions of the robot to look correct. Allowing the character to feel aesthetically correct to the rest of the environment.

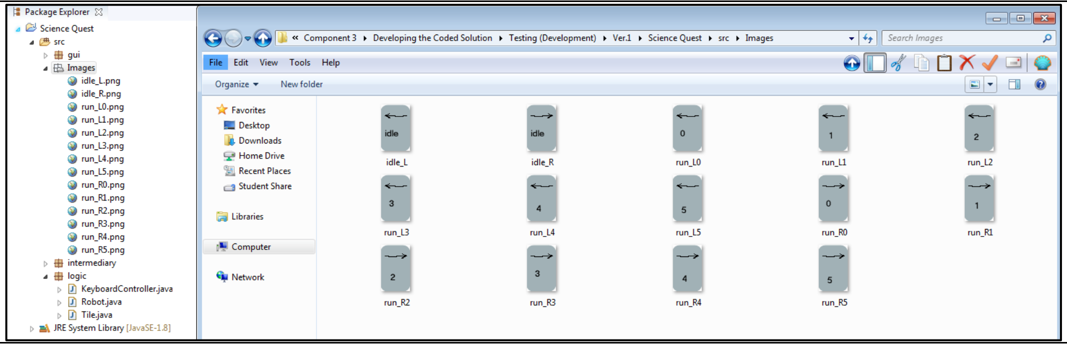
The final part to mention about the **Robot** class is frame switching; better known as animation. The last function is part of *setFrameNumber()* and it works effective animate the robot using still images, allowing all graphical input to be PNGs. Every time the *move()* function is called, the character moves and a *moveCounter* is incremented by one. The *moveCounter* variable is what makes the frame switching possible because every five increments of the *moveCounter*, it changes frame. Any faster than this and the robot would lose its desired animated effect when walking. On the other hand, any slower can give the impression that the program itself is undergoing graphical lag. With this in mind, a *MOVE_COUNTER_THRESH* of five is a good trade off to make the animation look visually appealing for the user.



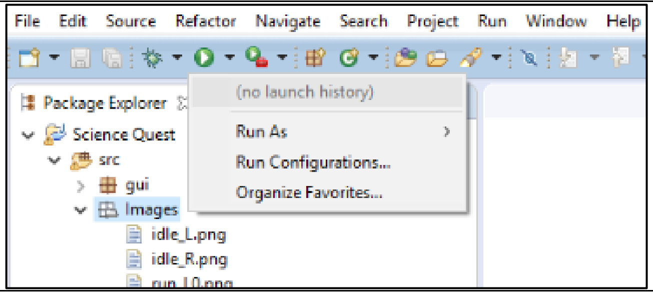
*Tile.java

```java
package logic;

public class Tile {
	public Tile(){
		
	}
}

```

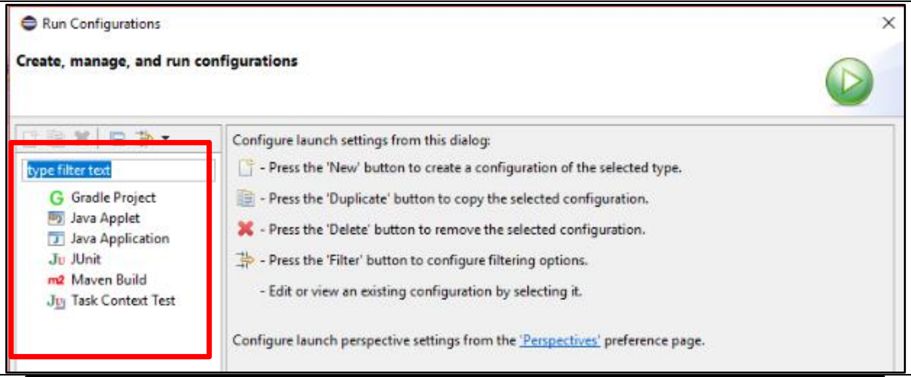
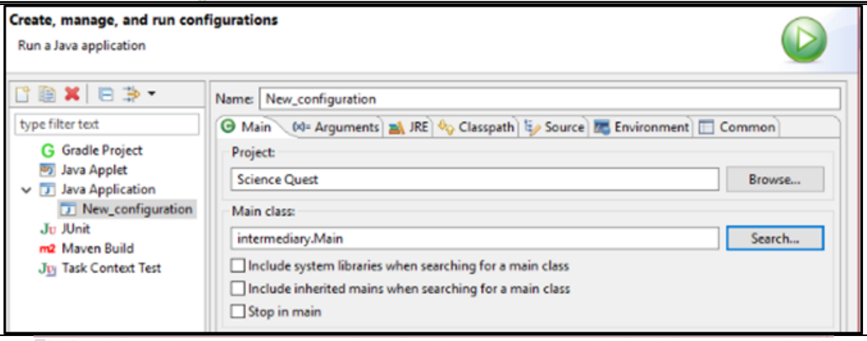
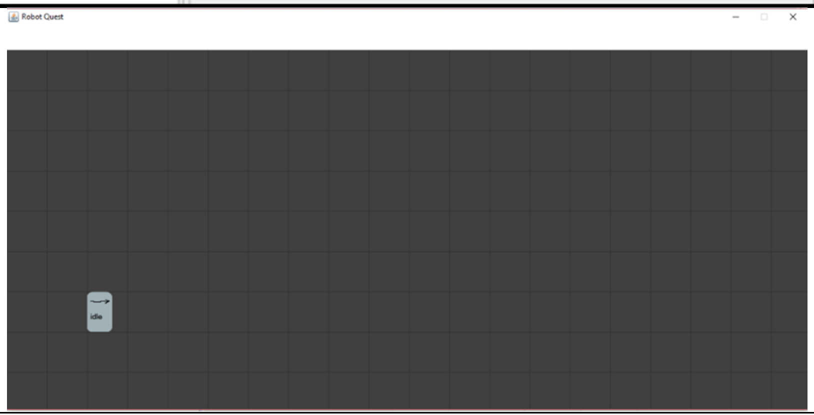
At this stage, the tile class is just a placeholder. It currently serves no purpose, but is part of the original logic module, so it was made with the rest of the classes. It will interact with the bounding box in later versions allowing platforms to exist.

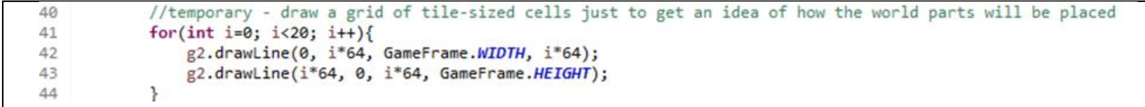


Above are the graphics that have been placed in the **Image** folder. While they are just placeholders for the animation of the movement, they will clearly show the direction the robot is facing and if the animation frames are working as intended. The graphics themselves are PNGs made in Photoshop; these will mimic the real character sprites that will be used in a later version when all the character animations are complete.



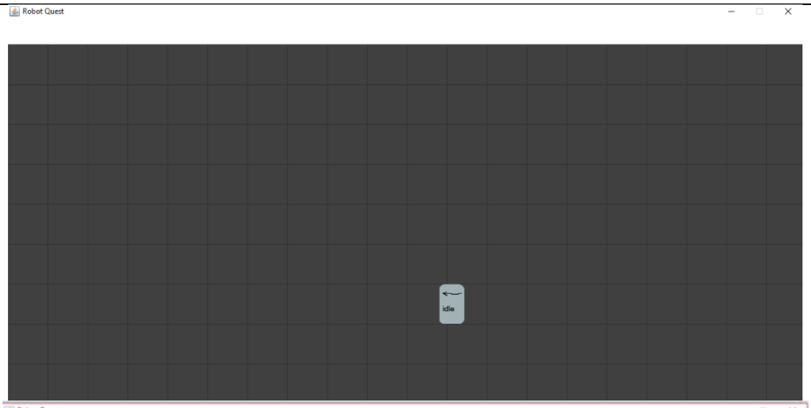
As this is the first time I will run this project, the JRE has no idea how to run it. This means that I will have to enter the run configurations and set up the project, as shown below.

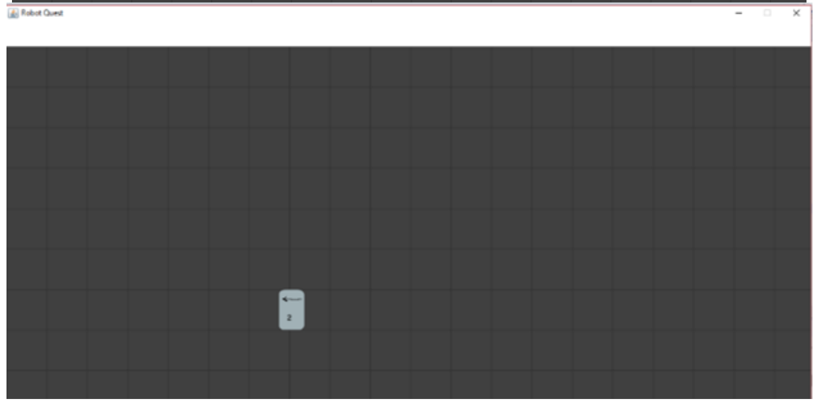
 Run Configurations **Create, manage, and run configurations** type filter text Gradle Project Java Applet Java Application JUnit Maven Build Task Context Test Configure launch settings from this dialog: - Press the 'New' button to create a configuration of the selected type. - Press the 'Duplicate' button to copy the selected configuration. - Press the 'Delete' button to remove the selected configuration. - Press the 'Filter' button to configure filtering options. - Edit or view an existing configuration by selecting it. Configure launch perspective settings from the 'Perspectives' preference page.	In this list (highlighted in the screenshot). I selected the Java Application option because my project is intended to run as an app (this was agreed upon with my client).
 **Create, manage, and run configurations** Run a Java application type filter text Gradle Project Java Applet Java Application New_configuration JUnit Maven Build Task Context Test Name: New_configuration Main | Arguments | JRE | Classpath | Source | Environment | Common Project: Science Quest Main class: intermediary.Main ☐ Include system libraries when searching for a main class ☐ Include inherited mains when searching for a main class ☐ Stop in main	For the next part, I entered my project 'Science Quest' and searched for my main class (that being the **Main** class in the **intermediary** package). I already have a system library for the project and no other mains, so I skipped the tick boxes.
	This is the first screen of the game upon initialisation.



```
40        //temporary - draw a grid of tile-sized cells just to get an idea of how the world parts will be placed
41        for(int i=0; i<20; i++){
42            g2.drawLine(0, i*64, GameFrame.WIDTH, i*64);
43            g2.drawLine(i*64, 0, i*64, GameFrame.HEIGHT);
44        }
```

Lines 41 to 43 in the **PlayPanel** are the reason the background is separated into 64x64 squares. This mimics the tiles that will make up the platforms of the game and serves as a measurement of distance in this version.





I finished with some final black box testing for this version. The character changes direction and changes frame in looping increments of 0-5 when the left or right directional key is held down. No other inputs will work, as the keys are restricted. The application can only be ended when it is closed via the close button on the frame.

Review

Taking the modular approach, I have used version one to focus on the frame my game will run in, and the simpler parts of my playable character. While I initially intended for every core aspect of the character to be completed in this version, the concept of jumping was just bit too advanced to do without going back and planning it. The result was still complete enough for a testable version.