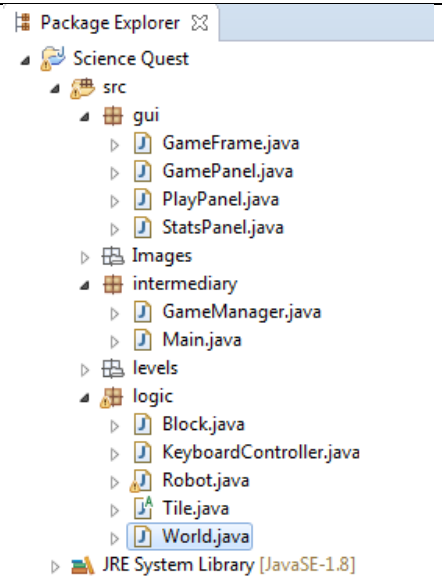
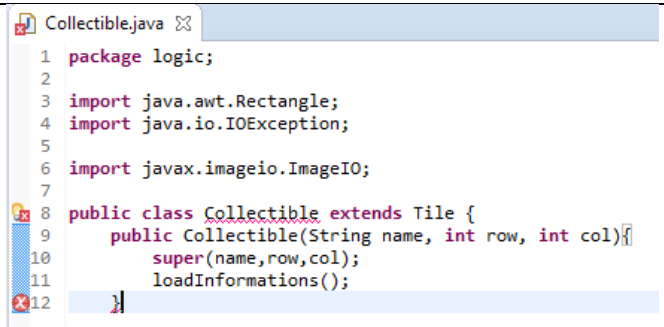
Developing the Coded Solution for Project *Version 4*

With the world created, version 4 will now create the classes that will be implemented in the world- namely objects and batteries.



From version 4, the **World** class now exists, along with the **Block** the next step to create another child class of the **Tile** the **Collectible** class.



```
Collectible.java
1  package logic;
2
3  import java.awt.Rectangle;
4  import java.io.IOException;
5
6  import javax.imageio.ImageIO;
7
8  public class Collectible extends Tile {
9      public Collectible(String name, int row, int col){
10         super(name,row,col);
11         loadInformations();
12     }
```

As a child class, I will be able to implement it on the *tiledmap*, by reusing the asset, it will allow collectables to be easily placed into a level, making development much faster.

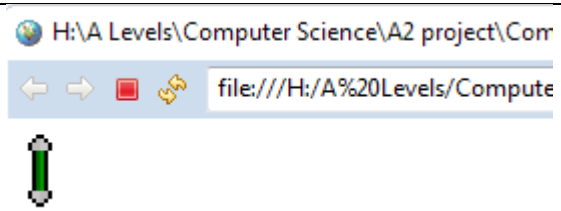
*Collectible.java

```java
package logic;

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

import javax.imageio.ImageIO;

public class Collectible extends Tile {
    public Collectible(String name, int row, int col){
        super(name,row,col);
        loadInformations();
    }

    @Override
    protected void initializeStuff() {
        currentX=col*TILE_SIZE+TILE_SIZE/2-width/2;
        currentY=row*TILE_SIZE;
        boundingBox=new Rectangle(currentX,currentY,width,height);
    }

    @Override
    protected void loadInformations() {
        try {
            image=ImageIO.read(getClass().getResource("../images/"+name+".png"));
            width=image.getWidth();
            height=image.getHeight();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private int width;
    private int height;
}
```

While collectables are a tile, they do not take up the all the space that is given per space (64x64 pixels), in fact, each collectable would vary in shape.
In terms of the battery, it is thinner than a block. The final parts of the **Collectible** class deal with the different dimensions compared to the other tiles, and then fetches the image. It will in turn, base the dimensions of the collectible on the dimensions of the image.

```java
                return new Block("terQ", i, j);
            case "terP":
                return new Block("terP", i, j);
            case "term":
                return new Block("term", i, j);
            case "mayC":
                return new Block("mayC", i, j);
            case "mayD":
                return new Block("mayD", i, j);
            case "mayU":
                return new Block("mayU", i, j);
            case "batt":
                return new Collectible("batt",i , j);
        }
        return null;
    }
```

With the **Collectible** class created, to allow it to be placed on the *tiledmap*, I have to add it to the **World** class. This means that when I type 'batt' onto the tiled map, I'll get a battery collectable back.

<table>
<tr>
<td> 
</td>
<td>The batteries on the tiledmap will be referred to as ‘batt’. This is the image that will be fetched for battery with image=ImageIO.read(getClass().getResource("../images/"+name+".png"));</td>
</tr>
<tr>
<td>

```
*Robot.java

241
242        public void checkCollectibles() {
243            if((boundingBox.getMaxX()/Tile.TILE_SIZE)<World.COLS){
244                if(World.tiledMap[currentRow][currentCol] instanceof Collectible){
245                    if(World.tiledMap[currentRow][currentCol].getBoundingBox().intersects(boundingBox)){
246                        if(World.tiledMap[currentRow][currentCol].getName().equalsIgnoreCase("batt")){
247                            batts++;
248                        }
249
250                        World.emptyTile(currentRow,currentCol);
251                    }
252                }
253            }
254        }
```

</td>
<td>The interaction with the robot can be found in the Robot class. If the space that the robot is in contains a battery, then if the robot intersects the battery’s boundingBox (remember it’s not the whole space), the battery will be removed from the level and the collected battery count will implement by one.</td>
</tr>
<tr>
<td>

```
448         //returns the number of darts currently owned by the robot
449         public int getBatts(){
450             return batts;
451         }
```

</td>
<td>This records the number of batteries for the StatsPanel.</td>
</tr>
<tr>
<td>

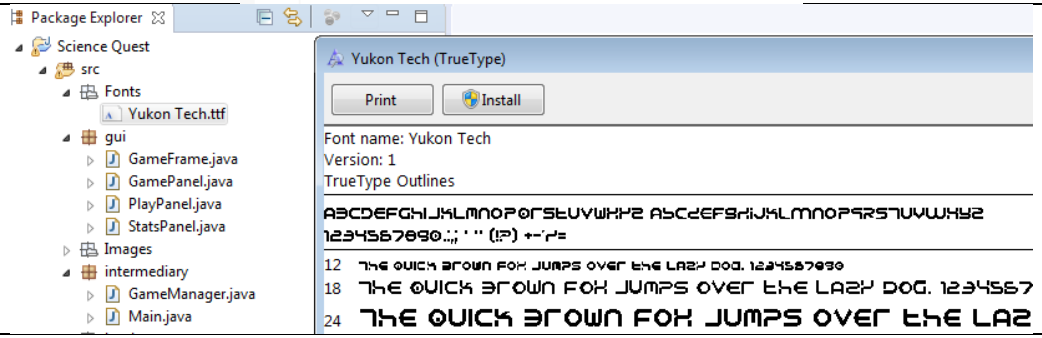
```
528     //batteries owned by the character. The robot starts with no batteries
529         private int batts=0;
530     }
```

</td>
<td>The robot should be starting the game with no batteries.</td>
</tr>
</table>

*GameManager.java

```
44
45          //manage the keys currently pressed
46          manageKeys();
47
48          robot.checkCollectibles();
49
50          robot.checkBlockCollisions();
51
52          robot.checkRestoringCount();
53
54          gamePanel.repaintGame();
55
56          try {
57              Thread.sleep(MAIN_SLEEP_TIME);
58          } catch (InterruptedException e) {
59              e.printStackTrace();
60          }
61      }
62  }
```

The **GameManager** needs to be updated, line 48 allows the *checkCollectibles* method in the **Robot** class to be active while the game plays.



With the **Logic** and **Intermediary** packages for the collectibles done, the next step is the **GUI**. While there is a built-in text font in the IDE I have imported a True Text File to use in my game, I picked it based on a futuristic design.

*StatsPanel.java

```java
package gui;

import java.awt.Color;
import java.awt.Font;
import java.awt.FontFormatException;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JPanel;

import logic.Robot;

public class StatsPanel extends JPanel{

    private static final long serialVersionUID = 1L;

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

    private void loadInformations() {
        try {
            statsPanel=ImageIO.read(getClass().getResource("../images/statsBar.png"));
            YukonFont=Font.createFont(Font.TRUETYPE_FONT, getClass().getResourceAsStream("../fonts/Yukon Tech.ttf")).deriveFont(35.0f);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (FontFormatException e) {
            e.printStackTrace();
        }
    }
```

With collectibles finally in the game, the **StatsPanel** can finally serve its purpose. The number of collected batteries will be shown on the panel, therefore an incrementing number will have to be placed on it, and this is where the TTF font I imported comes into play. I also updated the statsBar.png to include a battery on it.

```java
37      @Override
38      protected void paintComponent(Graphics g) {
39          super.paintComponent(g);
40          Graphics2D g2=(Graphics2D)g;
41          g2.setColor(Color.WHITE);
42          g2.setFont(YukonFont);
43          g2.drawImage(statsPanel,0,0,GameFrame.WIDTH-5,STATS_HEIGHT,null);
44
45          g2.drawString("x"+robot.getBatts(), BATT_COUNT_START_X, BATT_COUNT_START_Y);
46              }
47
48
49      public void addRobot(Robot robot) {
50          this.robot=robot;
51      }
52
53      private Font YukonFont;
54      private BufferedImage statsPanel;
55      public static final int STATS_HEIGHT=40;
56      private Robot robot;
57      private static final int BATT_COUNT_START_X=785;
58      private static final int BATT_COUNT_START_Y=30;
59  }
```

Using the *getBatts* method from line 449 in the **Robot** class, the time of batteries is drawn onto the panel in white Yukon Tech font. When the robot gets another battery, the *getBatts* will implement again, causing the recorded number to increase. This allows the user to keep track of the number of collected batteries.

PlayPanel.java

```
43  @Override
44  protected void paintComponent(Graphics g) {
45      super.paintComponent(g);
46
47      collectibleAnimationCount++;
48      if(collectibleAnimationCount%30==0){
49          collectible_y_offset=-collectible_y_offset;
50      }
51
52      if(collectibleAnimationCount>60){
53          collectibleAnimationCount=0;
54      }
55
56
57      Graphics2D g2=(Graphics2D)g;
58
59      //use antialiasing to draw smoother images and lines
60      g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
61
62      g2.drawImage(World.CURRENT_BACKGROUND,0,-Tile.TILE_SIZE,GameFrame.WIDTH,PLAY_PANEL_HEIGHT, null);
63
64      for(int i=0; i<World.ROWS; i++){
65          for(int j=0; j<World.COLS; j++){
66              if(World.tiledMap[i][j] instanceof Block){
67                  g2.drawImage(World.tiledMap[i][j].getImage(), World.tiledMap[i][j].getCurrentX(),
68                          World.tiledMap[i][j].getCurrentY(), null);
69              } else if(World.tiledMap[i][j] instanceof Collectible){
70                  g2.drawImage(World.tiledMap[i][j].getImage(), World.tiledMap[i][j].getCurrentX(),
71                              World.tiledMap[i][j].getCurrentY()+collectible_y_offset, null);
72                  }
73              }
74          }
```

In the **PlayPanel,** to make the collectibles stand out, I have given them a simple animation. Lines 40 to 47 will cause the collectible to shift up and down in place, giving it the illusion of floating. I believe that this animation will remove the batteries from the background, and make then noticeable as a key object in the foreground.

Lines 58 to 66 fetch the images of the collectible to allow this information. It does this my checking the *tiledMap* for the presence of batteries. If it finds any, it will constantly update the images for the changing of place.

*Object.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  import java.awt.Rectangle;
9
10 import javax.imageio.ImageIO;
11
12 public class Object {
13     public Object(String name, int row, int col,int numberOfSentences, int currentLevel, String ref, Boolean canTalk){
14         this.name=name;
15         this.row=row;
16         this.col=col;
17         this.numberOfSentences=numberOfSentences;
18         this.currentX=col*Tile.TILE_SIZE;
19         this.currentY=row*Tile.TILE_SIZE;
20         this.currentLevel=currentLevel;
21         sentences=new String[numberOfSentences+1];
22         idleLeft=new BufferedImage[FRAMES_NUMBER];
23         loadInformations();
24         currentFrame=idleLeft[0];
25         boundingBox=new Rectangle(currentX,currentY,width,height);
26         this.ref = ref;
27         this.canTalk = canTalk;
28     }
```

With the **Collectible** class complete, I will now begin to implement the object class. As objects are not **Tiles** I will have to completely redefine them.

Firstly, the object's name is stored as a string, this will be used as reference for the object. The row and col integers will be used a co-ordinates to reference were the object will appear on each level. The number of sentences exists to help the reader know how many lines exist for each object (if any). The currently level integer is straightforward, it references what level the information applies to, this allows for objects to exist in multiple levels. The ref is a reference to the name of another object for interaction dependency and finally, the canTalk boolean informs the game if the object will have lines or not.

```java
30    public void loadInformations(){
31        try {
32            idleLeft[0]=ImageIO.read(getClass().getResource("../object_sprites/"+name+"_off1.png"));
33            idleLeft[1]=ImageIO.read(getClass().getResource("../object_sprites/"+name+"_off2.png"));
34            idleLeft[2]=ImageIO.read(getClass().getResource("../object_sprites/"+name+"_off3.png"));
35
36            InputStream is=this.getClass().getResourceAsStream("/object_info/"+name+"_script_"+currentLevel+".txt");
37
38            if(is==null){
39                return;
40            }
41
42            BufferedReader reader=new BufferedReader(new InputStreamReader(is));
43            String line=null;
44
45            try {
46                int i=0;
47                while((line=reader.readLine())!=null){
48                    sentences[i]=line;
49                    i++;
50                }
51            } catch (IOException e) {
52                e.printStackTrace();
53            }
54
55        } catch (IOException e) {
56            e.printStackTrace();
57        }
58    }
```

Every object will have two states, off; before the robot has interacted with it, and on; after the robot has interacted with it. The first *loadinformations(){* concerns with the latter. Each object will have 3 still frames that will loop to create animations (if any). Information about what the object will say on the level is also fetched, with instructions on how to read them.

```java
60    public void loadAfterInformations(){
61        try {
62            idleLeft[0]=ImageIO.read(getClass().getResource("../object_sprites/"+name+"_on1.png"));
63            idleLeft[1]=ImageIO.read(getClass().getResource("../object_sprites/"+name+"_on2.png"));
64            idleLeft[2]=ImageIO.read(getClass().getResource("../object_sprites/"+name+"_on3.png"));
65        } catch (IOException e) {
66            e.printStackTrace();
67        }
68    }
```

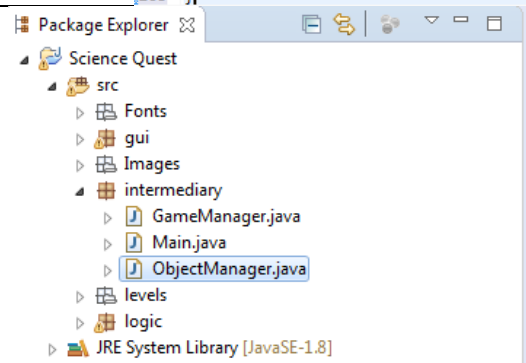
After an object has been interacted with, it will no longer need any information about the lines it would have said. Now, it only as a different set of still images, to show a different animations as it has now been interacted with.

70 public BufferedImage getCurrentFrame() {
71 if(interacted){
72 return idleLeft[0];
73 }
74
75 if(frame_count>=2){
76 add_value=-1;
77 }
78 if(frame_count<=0){
79 add_value=1;
80 }
81
82 int currentFrame=frame_count;
83
84 if(time_count%10==0){
85 frame_count+=add_value;
86 }
87
88 time_count++;
89 if(time_count>100){
90 time_count=1;
91 }
92
93 return idleLeft[currentFrame];
	These conditions are set up to run the objects animations in the order 1, 2, 3, in a constant loop. The each condition also has timings to make the animations fluid.

public int getRow() {
return row;
}
public int getCol() {
return col;
}
public int getCurrentX() {
return currentX;
}
public int getCurrentY() {
return currentY;
}
 	
public String getName() {
return name;
}
 	
public void interact() {
interacted=true;
talking=true;
loadAfterInformations();
}
 	
public boolean isTalking() {
return talking;
}
 	
public boolean continueTalking() {
currentSentence++;
if(currentSentence>=numberOfSentences){
currentSentence=numberOfSentences;
talking=false;
return false;
}
return true;
}
 	
public String getSentence(){
return sentences[currentSentence];
}
	Lines 96+ is mostly declaring variables. The *interact* method is important for objects and the two states of an object depend on the boolean. The *continueTalking* boolean is also important as it allows objects to have more than one line of text, it also manages the *talking* boolean, that changes the state the robot is in.

```java
    //if the object is not talking (talking='false') the game will not display
    //any the speech balloon. otherwise it will display a specific speech
    //depending on how many times the player has talked to this object
    private boolean talking=false;
    private String[] sentences;
    private int currentSentence=0;
    private int numberOfSentences;
    private int add_value=1;
    private int time_count=1;
    private int frame_count=0;
    public boolean interacted=false;
    private static final int FRAMES_NUMBER=3;
    protected Rectangle boundingBox;
    private BufferedImage currentFrame;
    private String name;
    private BufferedImage[] idleLeft;
    private int row;
    private int col;
    private int width;
    private int height;
    private int currentX;
    private int currentY;
    private int currentLevel;
    public static final int OBJECT_SIZE=128;
    public String ref;
    public Boolean canTalk;
}
```

The final variables are declared. Objected start in an un-interacted state (interacted=false) and haven't talked until the robot interacts with them.




```java
package intermediary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

import logic.Object;
import logic.Tile;

public class ObjectManager {
    public ObjectManager(int currentLevel){
        this.currentLevel=currentLevel;
        currentObjects=new ArrayList<Object>();
        loadInformations();
    }
```

The next step was to create a new intermediary class, the **ObjectManager**. This class stores all of a stages objects in a simple array of objects (objects in terms of OOP, not the class name).

```java
19      private void loadInformations() {
20          InputStream is=this.getClass().getResourceAsStream("/object_info/level"+String.valueOf(currentLevel)+".txt");
21          if(is==null){
22              return;
23          }
24          BufferedReader reader=new BufferedReader(new InputStreamReader(is));
25          String line=null;
26          String[] singleObjectInfo;
27          try {
28              while((line=reader.readLine())!=null){
29                  singleObjectInfo=line.split(" ");
30                  currentObjects.add(new Object(singleObjectInfo[0],Integer.valueOf(singleObjectInfo[1]),
31                          Integer.valueOf(singleObjectInfo[2]),Integer.valueOf(singleObjectInfo[3]),currentLevel,singleObjectInfo[4],Boolean.valueOf(singleObjectInfo[5])));
32              }
33          } catch (IOException e) {
34              e.printStackTrace();
35          }
36      }
37
38      public ArrayList<Object> getObjects() {
39          return currentObjects;
40      }
```

Information about where to put the object, what the object has to say, any dependency and if it has something to say, will be stored in text files for each level. The **ObjectManager** will consult these files according to the level currently played by the user.

```java
42      //returns the closest enemy within two tiles of distance (which is
43      //the maximum distance allowed for an interaction with an object)
44      public Object closestObject(int robotRow, int robotCol) {
45          Object closestObject=null;
46          int currentDistance;
47          for(int i=0; i<currentObjects.size(); i++){
48              if(robotRow!=currentObjects.get(i).getRow()){
49                  continue;
50              }
51              if((Math.abs(currentObjects.get(i).getCol()-robotCol))<=MAXIMUM_TALKING_DISTANCE){
52                  currentDistance=Math.abs(currentObjects.get(i).getCurrentX()-(robotCol*Tile.TILE_SIZE));
53                  if(closestObject==null){
54                      closestObject=currentObjects.get(i);
55                  } else {
56                      if(currentDistance<Math.abs(closestObject.getCurrentX()-robotCol+Tile.TILE_SIZE)){
57                          closestObject=currentObjects.get(i);
58                      }
59                  }
60              }
61          }
62
63          return closestObject;
64      }
65      public boolean collidingObject(int robotRow, int robotCol) {
66          Object closestObject=null;
67          int currentDistance;
68          //return false;
69          for(int z=0; z<currentObjects.size(); z++){
70              if(!currentObjects.get(z).interacted) {
71                  while((Math.abs(currentObjects.get(z).getCol()-robotCol))<=MAXIMUM_COLLISON_DISTANCE){
72                      currentDistance=Math.abs(currentObjects.get(z).getCurrentX()-(robotCol*Tile.TILE_SIZE));
73                      if(closestObject==null){
74                          closestObject=currentObjects.get(z);
75                          return true;
76                      }
77                  }
78              }
79          }
80          return false;
81      }
```

The *closestObject()* function is called every time the robot tries to interact with an object. This function finds the closest object given the robot's current position. But the robot can't interact with things further than two spaces away, that is due to the control at line 51 based on the constant MANIMUM_TALKING_DISTANCE.

Likewise, if the robot gets within one space of an object, it will be stopped by the *collidingObject()* function. Using the same logic as *closestObject(),* the boolean will be determined by the robot's distance from the object.

```java
83      private static final int MAXIMUM_TALKING_DISTANCE=2;
84      private static final int MAXIMUM_COLLISON_DISTANCE=1;
85      private ArrayList<Object> currentObjects;
86      private int currentLevel;
87      }
```

Finally, there are the two constants that are key for managing interactions with objects. The two distances are in spaces (or tiles), one space being the size of a tile, 64 pixels.

*GameManager.java

```java
//pressed, associating them to actions
public class GameManager extends Thread {
    public GameManager(GamePanel gamePanel){
        this.world=new World();
        this.world.initializeStage(currentLevel);

        this.objectManager=new ObjectManager(currentLevel);

        //Initialise the protag of the game
        this.robot=new Robot();

        //stores the gamePanel and adds the robot and the objects to it
        this.gamePanel=gamePanel;
        this.gamePanel.addRobot(robot);

        if(objectManager.getObjects().size()>0){
            gamePanel.addObjects(objectManager.getObjects());
        } else {
            gamePanel.clearObjects();
        }
```

An instance of the **Objectmanager** class is stored in the **Gamemanager**, the main thread of the game, which also controls a set number of currently pressed keys and reacts to them in the proper way via the *manageKeys()* function.

```java
    if(!listening){
        //manage the two possible run direction
        if(currentKeys.contains(KeyEvent.VK_RIGHT)){
            // to simulate a boundingBox, while an object hasn't been interacted
            // collidingObject will be set to true, removing the ability to move
             //forward until the object has been interacted with.
             if (!objectManager.collidingObject(robot.getRow(),robot.getCol())) {
                 //move right
                 robot.move(KeyEvent.VK_RIGHT);
             }

        } else if (currentKeys.contains(KeyEvent.VK_LEFT)){
             if (!objectManager.collidingObject(robot.getRow(),robot.getCol())) {
             //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();
        }
    }
```

To mimic collisions for objects, the collidingObject boolean has been added as a condition to move the robot. This means that if the robot is within one space of an object, it won't be able to move until the object has been interacted with.

```
107         if(currentKeys.contains(KeyEvent.VK_ENTER)){
108             Object tempObject;
109             //find the closest object according to the character's position
110             if((tempObject=objectManager.closestObject(robot.getRow(),robot.getCol()))!=null){
111
112                 //if the object is already talking, keep talking...
113                 if(tempObject.isTalking()){
114                     if(!(tempObject.continueTalking())){
115                         listening=false;
116                     }
117
118                 //otherwise interact with the object
119                 } else {
120                     tempObject.interact();
121
122                     //put the character in <idle> status when it's talking
123                     robot.stop();
124
125                     //prevent the character from moving when talking
126                     listening=true;
127
128                     //this object has been interacted with, get the name
129                     String name = tempObject.getName();
130                     ArrayList<Object> objects = objectManager.getObjects();
131                     System.out.println("Current obj: "+name);
132                     for(int i=0; i<objects.size(); i++){
133                         Object objToInspect = objects.get(i);
134                         System.out.println("Inspecting: "+objToInspect.getName() + " | ref: " + objToInspect.ref);
135                         if(objToInspect.ref.equals(name)){
136                             objToInspect.interact();
137                             System.out.println("Interact with it!");
138                         }
139                     }
140                 }
141             }
142             currentKeys.remove(KeyEvent.VK_ENTER);
143         }
144     }
```

As I said before the **GameManager** controls a set number of keys, lines 107 to 144 are used to link the enter key to object interactions.

Now if you press enter within the effective distance of an object you will interact with it. While if the object can talk, pressing the enter key will make it say the next line.

Finally, object dependency is addressed here. If an object has a dependency on another object (e.g. a lightbulb and a switch), then when the object enters an interacted state- the dependant object will also enter an interacted state without being interacted with by the 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 all it 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;

    private ObjectManager objectManager;
}
```

The *currentLevel* integer starts at one and is used to determine what text file to read for the object information, it is incremented at the same time as the level.

```
44    private void loadInformations() {
45        try {
46            speechBalloon=ImageIO.read(getClass().getResource("../images/speechBalloon.png"));
47            YukonFont=Font.createFont(Font.TRUETYPE_FONT, getClass().getResourceAsStream("../fonts/Yukon Tech.ttf")).deriveFont(25.0f);
48        } catch (IOException e) {
49            e.printStackTrace();
50        } catch (FontFormatException e) {
51            e.printStackTrace();
52        }
53    }
```

The next edit is in the **PlayPanel**, when an object talks, it should have a comic like speech bubble to indicate this. Lines 46 to 51 import a speech bubble PNG I have created.

```
94            if(currentObjects.size()>0){
95                Object currentObject;
96                for(int i=0; i<currentObjects.size(); i++){
97                    currentObject=currentObjects.get(i);
98                    // the object sprite is drawn one row above there real position simply because the image of an object
99                    //is twice as tall as the protag's
100                   g2.drawImage(currentObject.getCurrentFrame(),currentObject.getCurrentX(),currentObject.getCurrentY()-Tile.TILE_SIZE,null);
101                   if(currentObject.isTalking()&&currentObject.canTalk){
102
103                       if(++balloonCount%30==0){
104                           if(dynamicBalloonOffset==2){
105                               dynamicBalloonOffset=0;
106                           } else {
107                               dynamicBalloonOffset=2;
108                           }
109                       }
110
111                       g2.drawImage(speechBalloon,currentObject.getCurrentX()-speechBalloon.getWidth()/2+
112                               Tile.TILE_SIZE/2-SPEECH_BALLOON_X_OFFSET,currentObject.getCurrentY()+Tile.TILE_SIZE+1
113                               +dynamicBalloonOffset,null);
114                       tempSentence=currentObject.getSentence();
115
116                       if(tempSentence.contains("NEWLINE")){
117                           g2.drawString(tempSentence.split(" NEWLINE ")[0], currentObject.getCurrentX()-speechBalloon.getWidth()/3-
118                                   SPEECH_BALLOON_X_OFFSET*5,
119                                   currentObject.getCurrentY()+speechBalloon.getHeight()-20+dynamicBalloonOffset);
120                           g2.drawString(tempSentence.split(" NEWLINE ")[1], currentObject.getCurrentX()-speechBalloon.getWidth()/3,
121                                   currentObject.getCurrentY()+speechBalloon.getHeight()+dynamicBalloonOffset);
122                       } else {
123                           g2.drawString(tempSentence, currentObject.getCurrentX()-speechBalloon.getWidth()/3-SPEECH_BALLOON_X_OFFSET*5,
124                                   currentObject.getCurrentY()+speechBalloon.getHeight()-20+dynamicBalloonOffset);
125                       }
126
```

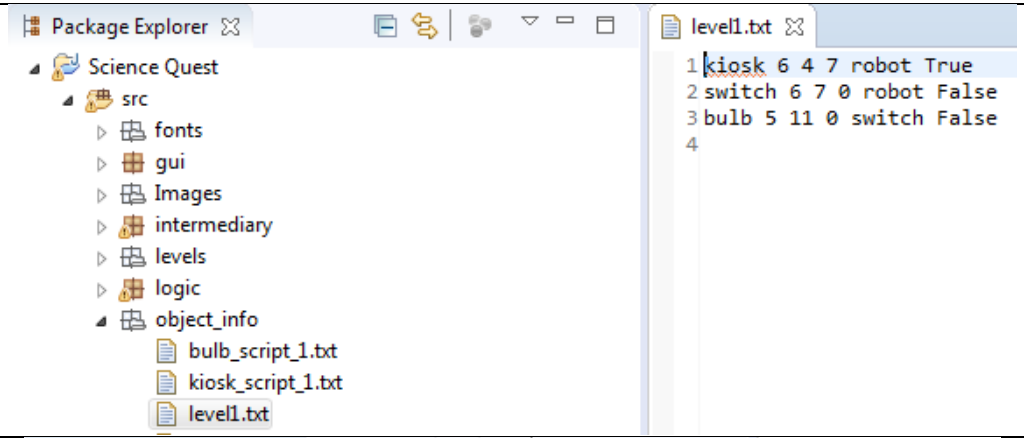
Much like the robot, the objects need to be drawn onto the screen. Lines 103 to 125 are all concerned about drawing and redrawing the speech bubble in the right place and animating it (it floats up and down like a collectable).

```
141    public void addObjects(ArrayList<Object> currentObjects) {
142        this.currentObjects=currentObjects;
143    }
144
145    public void clearObjects() {
146        currentObjects.clear();
147    }
148
149    //height of the terrain in pixels - this is basically the distance of the robot's feet
150    //from the bottom border of the window you play the game in
151    public static final int TERRAIN_HEIGHT=192;
152
153    //height of the PlayPanel
154    public static final int PLAY_PANEL_HEIGHT=640;
155
156    //reference to the protag of the game
157    private Robot robot;
158
159    private ArrayList<Object> currentObjects;
160
161    private BufferedImage speechBalloon;
162
163    private static final int SPEECH_BALLOON_X_OFFSET=5;
164
165    private Font YukonFont;
166
167    private int dynamicBalloonOffset=0;
168    private int balloonCount=0;
169
170    private String tempSentence;
171
172    private int collectibleAnimationCount=0;
173    private int collectible_y_offset=2;
```

The object array has been added into the **PlayPanel** allowing it to read off it to draw objects. I have also used the same font used in the stats panel, although I may (and can) change it if I feel it isn't clear enough.

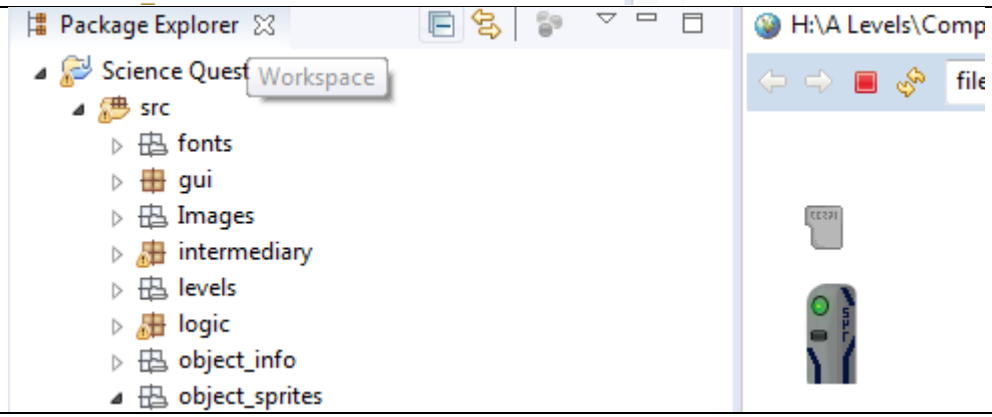
```
43    public void clearObjects() {
44        playPanel.clearObjects();
45    }
46
47    public void addObjects(ArrayList<Object> currentObjects) {
48        playPanel.addObjects(currentObjects);
49    }
```

The objects have been added to the **GamePanel**, lines 43-44 remove them from the panel at the end of the level. Lines 47-49 read the array and place them into the panel at the start of the transition of the level.



With all the logic behind objects complete, the next step was to provide the text files that provide information for each object. These files are saved in a new folder, the object_info folder.

There are two types of this information. Level.txt, what gives placement, script and dependency information for all the objects on the level. Script.txt, just provides the lines that the object will say.



Finally, I created the object_sprites folder. Like the robot, the every object has still PNG files to make up there animation, but as each object will have six images, I created its own folder as it will get very large very quickly.

	This will serve as a testing level. It consists of one talking object (the kiosk), one dependant object (the bulb) and it's trigger (the switch) and three batteries. Firstly, everything is placed in the level, this is a good start. Due to the objects being twice the height of a tile, when using tile coordinates to map them onto a level, you have to plot them one tile higher on the y-axis.
	As I get close to the kiosk (the first object), my robot stops dead in its tracks. I can walk away from it, but unlike block collisions I don't get the animation of trying to walk into it. While ascetically this is a shame, mechanically, the system is working just fine.

	Pressing enter to talk works just fine, I can freely move past the object after as well. While I wish I could move in closer to the object, I don't want to use a non-tile unit of measurement, and I believe that using a fractional measurement would break some calculations. I believe this text is too hard to read, I will change it in the next version of the game.
	When I interacted with the switch the light turned on and I can freely walk past both. Dependency seems to be working with no issues.

When I touch a battery, it disappears and my battery count on the stats panel goes up by one. The collectibles are working like intended.

Review

With the implementation of intractable objects and collectables and collectables, the game is as mechanically complete as I intend for it to get. Objects and collectables have been designed to promote ease of rapid implementation. This allows for version 5 to be implemented much faster than if I had to create a new class for each object, at the cost of some loss of animation and interactions (e.g. I could add timings before an object with dependency entered an on state). With the changes to level transition batteries now just become a feature that doesn't serve much of a purpose, they float around and you're told to collect them, hopefully that'll be enough to entice players to grab them. Finally, with problems attempting to use the same logic to create collisions between the robot and objects, I've had to take the desperate measure of cutting off movement instead to mimic a collision. This is a bit of a shame as it lacks the animation that trying to run into a block will give, but mechanically will work as long as the level progresses from left to right (meaning further straying away from the original layout of levels, although now that each designed level can exist as multiple in-game levels, this was going to happen regardless).