

# KeyFinder

How It Works

---

## Preamble

KeyFinder is a 2D game where the player's goal is to find the treasure at the end of each level by unlocking a series of doors using matching keys. The game was developed and submitted for credit in my computer science 12 course, taking me just over two months to complete. A lot of hard work went into creating this project as everything was designed by myself, including the music, sprites, and the other artwork to be found in the game. The goal of this project was to creatively explore one of the major units that we studied during our time in the course, 2D arrays, object-oriented programming, inheritance and polymorphism, recursion, and algorithm efficiency to name a few of the topics that we studied. KeyFinder started off as my experimentation with java class structures, the unit I originally intended to focus on, and from this jumping-off-point grew into the complex treasure hunting adventure game that it is today.

In this document I aim to explain how the project functions from a general perspective. I won't be going too much into the nitty gritty that is, the specifics of the code, but I will demonstrate the general workings of the project's seven important 'domains.' These domains are as follows: The Main class and GamePanel class, Tile and TileManager classes, Map classes, Object classes, Collision functionality, Entity classes and finally a miscellaneous group that I've labelled the Utility classes. I will explain how each of these domain's function in the order mentioned above, outlining how they contribute to the functionality of the game. In doing so I hope to demonstrate the workings of KeyFinder. Without further ado, let us begin.

## Main and GamePanel Classes

Let us begin by exploring the Main class and GamePanel class. KeyFinder utilizes the JFrame and JPanel classes to create the game window and display the game. In the Main class of the project a JFrame is created, and its properties are setup. Aspects such as resizability, whether the window can be closed using the 'x' button, and the title of the window are declared before the program creates an instance of the most important class in the game: The GamePanel class.

Most fundamentally the GamePanel class handles the game loop structure. The game loop is how the game updates the picture on the screen, essentially, it's how the game is presented to the user. In the run method, the game update and repaint methods are called once per-frame. The update method checks the player (and other possible entities) for the possible translations that an entity might experience. In KeyFinder, the only entity is the player, so this method isn't particularly large. The other method called in the run method, the repaint method, is also tremendously important. It is responsible

for re-drawing the game's visual components, like the UI elements, as well as player sprites and tiles. The repaint method redraws the elements of the game by calling their draw methods each frame. This ensures that the visuals are updated to reflect the changes in translation that occur within the update method. As stated earlier, these methods are called within the run method meaning that they are called once per-frame, which for KeyFinder is 60 times per second. This ensures that the game information is updated and that the game runs smoothly. The game loop is a hugely important aspect of the GamePanel class.

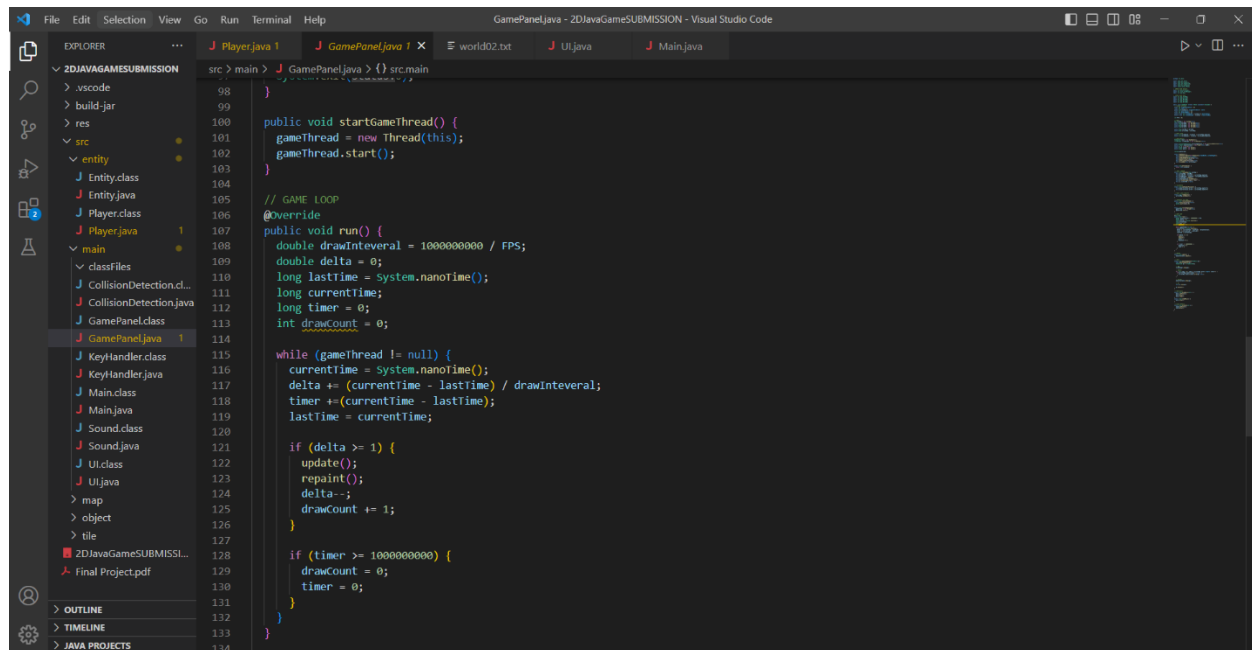


Figure 1 Run method overview.

Additionally, the GamePanel is responsible for the initialization of the KeyHandler, TileManager, CollisionDetection, Map, PlayerInstance, UI, and Sound classes. These classes are the fundamental game objects, and the GamePanel acts as the place where they all meet and work together.

There are also smaller, but still important tasks that the GamePanel is responsible for handling. The different sound functions find their home within the GamePanel class, allowing any of the other game objects to access and utilize their functionality. Additionally, level loading is controlled in the GamePanel class via the reloadMap, setPlayerPosition and setObjects methods. Again, as these methods are located within the GamePanel class so they can be utilized by any of the other game objects.

The GamePanel provides the sturdy foundation for the rest of the game. It handles both the updating of the game objects' position and imagery as well as their creation. It is also responsible for facilitating the interactions between the numerous game elements behaving almost like a town square where the game objects can communicate with one another with the exchange of information. The GamePanel is the basis upon which all the interesting things occur and was designed to be a robust foundation for the rest of the game features.

## The TileManager and Tile Classes

I mentioned in the preamble that one of my goals with KeyFinder was to experiment with java class interactions. Well, as the project began to increase in scale and complexity, I needed to develop an efficient way to store and load map information. Simple 2D arrays were no longer cutting it. So, I decided to explore the possibility of incorporating another topic that I covered in my grade 12 Computer Science course: Text input and output.

My idea was to have maps load from a text file that contained a bunch of numbers. These numbers would represent the different tiles that could exist in each space and thus maps could be created and stored in these files. Now all I had to do was implement the idea into my game.

First, I created a dummy map, 16 X 16 tiles and filled with only dirt (represented by the number 0). I placed this map into the map directory within the res folder (the folder where all the graphical information is stored). Then I got to developing the loadMap method within the TileManager class. In this method, text files are grabbed directly from a given maps mapPath variable (the loadMap method accesses this information via the GamePanel's currnetMap Map object). The file is imported as an InputStream and a BufferedReader is assigned to the map file. This BufferedReader then reads the file line by line storing the numbers in a 2D integer array. The boundaries for the BufferedReader set in the GamePanel's currnetMap Map object.

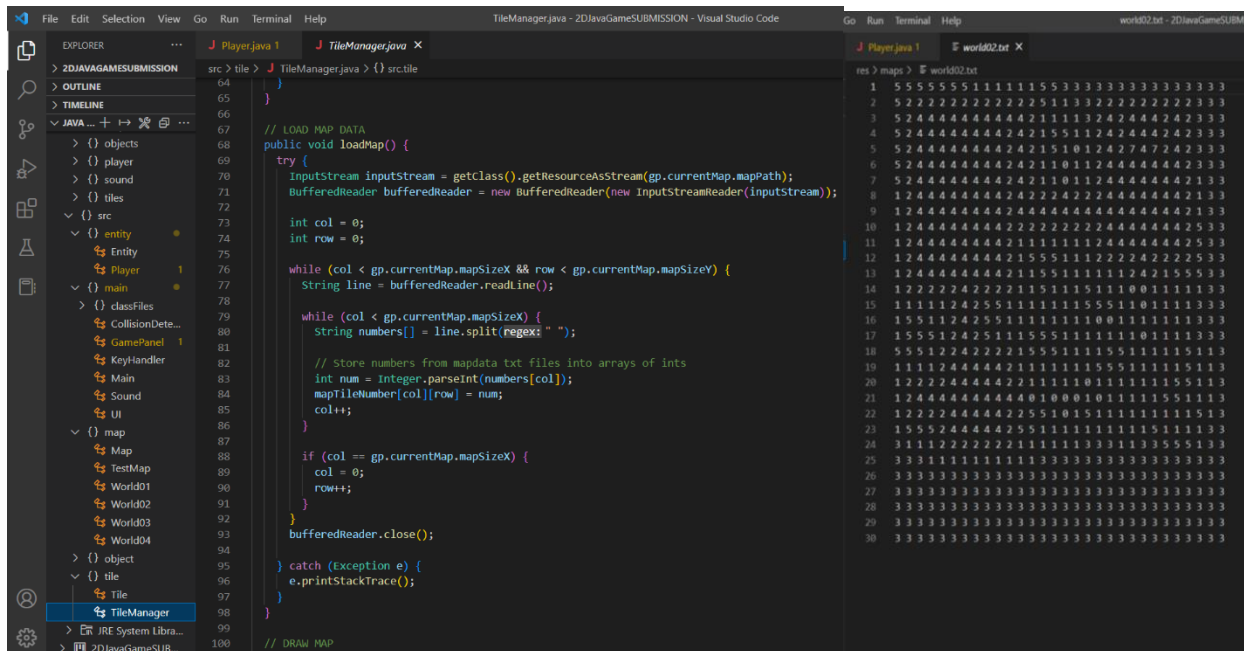


Figure 2 Left, loadMap method. Right, map file example.

Once the 2D array has been filled with integers, the TileManager class draw method is called. This method reads the 2D integer array and draws the correct tile type for the integers that it finds. To do this, the draw method references an array of tile objects. A tile object simply has a BufferedImage and collision Boolean which tells the collision handler if the tile is solid or not. The tiles are created and stored into the tile object array within the TileManager class getTileImage method. The draw method

uses the integers that it finds as index positions in the tile object array. Thus, maps can be loaded from any file so long as the Map class for that object is created.

## The Map class and Subclasses

Speaking of the Map class, it was created to make level loading and new level creation easy. I decided to create a Map superclass / subclass system. Every map, while technically having its own class, is an instance of the Map class. The Map class serves to provide KeyFinder's maps with an overarching framework while the individual map files serve as the places where specific information regarding a given map is determined. For example, I mentioned in the TileManger section that each map has its own mapPath variable, that way different tile sets can be loaded for different maps. Well, in our superclass / subclass system, the mapPath String exists in the Map class and is defined to be `"/res/maps/EXAMPLEMAP.txt"` within a maps constructor. In a similar fashion, a maps GamePanel, name, mapSizeX and mapSizeY (the number of columns and rows the map has), width and height are all set individually within a given map file.

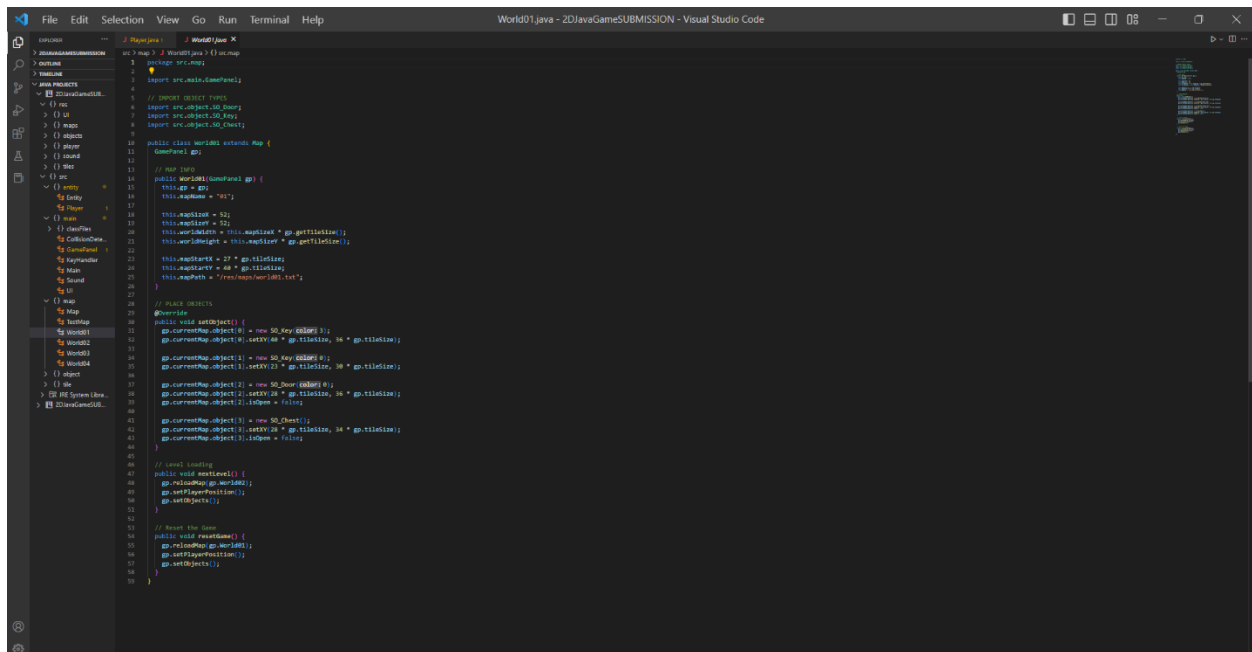


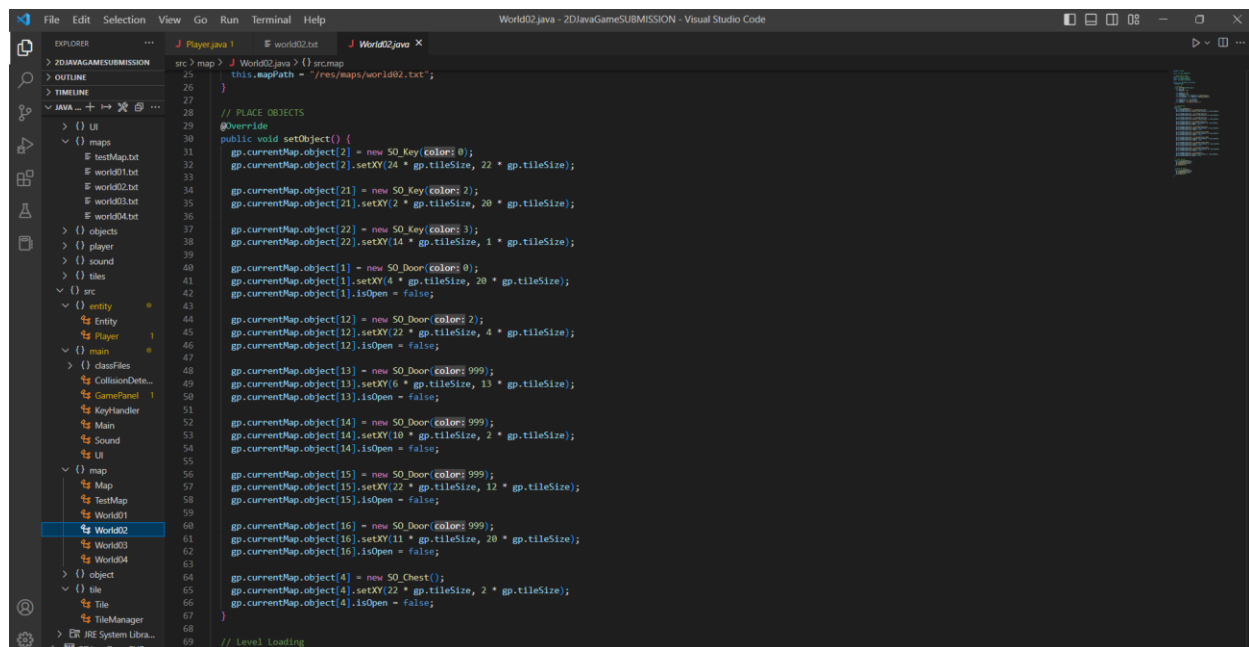
Figure 3 Map file overview.

This system allows for the easy creation of new maps. All that needs to be done to create a new level for KeyFinder is to generate a text file, fill it with integers that represent tiles (which are initialized within the TileManager's getTileImages method), create a new subclass of the Map class (it can be called whatever you want it to be, but naming conventions are a nice organizational touch), set the mapSizeX and mapSizeY to be the number of columns and rows the map has, and define the player's start X and start Y values. Congratulations, you've created your first KeyFinder map.

Map objects are also loaded within a given Map class; this process is handled by the setObject method. The setObject method will look different for any given Map class, but it was necessary for it to be

included within the Map superclass so that other classes could access the object information, namely the GamePanel's setObject method which is where object loading starts.

It is within the setObject method of a map that the various objects the player will encounter are created and placed. Similarly, to how tiles are created, the setObject method fills a given map's object array with a new Instance of the SuperObject class (overarching object class which we will get into during the next section) at each index. The setObject method then specifies which type of object exists at this index in addition to defining the object's colour (if applicable) as well as its X/Y coordinates. Each map sets its objects using the setObject method which is called through the GamePanel class (this process, though it's appears to be a roundabout way of handling things ensures that only the correct map information is loaded).



```
src> map> J World02.java { } srcmap
25 this.mapPath = "/res/maps/world02.txt";
26
27
28 // PLACE OBJECTS
29 @Override
30 public void setObject() {
31     gp.currentMap.object[2] = new SO_Key(color: 0);
32     gp.currentMap.object[2].setXY(24 * gp.tileSize, 22 * gp.tileSize);
33
34     gp.currentMap.object[21] = new SO_Key(color: 2);
35     gp.currentMap.object[21].setXY(2 * gp.tileSize, 20 * gp.tileSize);
36
37     gp.currentMap.object[22] = new SO_Key(color: 3);
38     gp.currentMap.object[22].setXY(14 * gp.tileSize, 1 * gp.tileSize);
39
40     gp.currentMap.object[1] = new SO_Door(color: 0);
41     gp.currentMap.object[1].setXY(4 * gp.tileSize, 20 * gp.tileSize);
42     gp.currentMap.object[1].isOpen = false;
43
44     gp.currentMap.object[12] = new SO_Door(color: 2);
45     gp.currentMap.object[12].setXY(22 * gp.tileSize, 4 * gp.tileSize);
46     gp.currentMap.object[12].isOpen = false;
47
48     gp.currentMap.object[13] = new SO_Door(color: 999);
49     gp.currentMap.object[13].setXY(6 * gp.tileSize, 13 * gp.tileSize);
50     gp.currentMap.object[13].isOpen = false;
51
52     gp.currentMap.object[14] = new SO_Door(color: 999);
53     gp.currentMap.object[14].setXY(10 * gp.tileSize, 2 * gp.tileSize);
54     gp.currentMap.object[14].isOpen = false;
55
56     gp.currentMap.object[15] = new SO_Door(color: 999);
57     gp.currentMap.object[15].setXY(22 * gp.tileSize, 12 * gp.tileSize);
58     gp.currentMap.object[15].isOpen = false;
59
60     gp.currentMap.object[16] = new SO_Door(color: 999);
61     gp.currentMap.object[16].setXY(11 * gp.tileSize, 20 * gp.tileSize);
62     gp.currentMap.object[16].isOpen = false;
63
64     gp.currentMap.object[4] = new SO_Chest();
65     gp.currentMap.object[4].setXY(22 * gp.tileSize, 2 * gp.tileSize);
66     gp.currentMap.object[4].isOpen = false;
67 }
68
69 // Level Loading
```

Figure 4 setObject method for World02 map.

*Side Note:* The index naming conventions for these objects are as follows: the first number stands for the type of object (1 = door, 2 = key, 4 = chest), the following digits are to be interpreted as the instance of that type of object (114 would mean door number 14). Given that KeyFinder is a small-scale game, the length of an object's index is limited to 3 digits, you probably won't need more.

There is also one smaller helper method that the map classes of KeyFinder possess. Each map has its own nextLevel method. This method is called by the player and tells the GamePanel to execute the tasks necessary for map loading. The tiles for the next map are loaded and drawn, the player's X and Y coordinates are set to the startX and startY of the new map, the keys that he has are reset, and the next map's objects are placed into the world.

The Map class is one of the more integral aspects of this project as it is the manifestation of my goal for this project: to experiment with java class structures if you recall. Not only this, but this method of programming maps ensures that the process to engineer a new map for KeyFinder is not complicated and that the necessary information for the games other classes is easily accessible.

## Objects

In the preceding section we referenced objects within a maps object array. A maps object array is an array of SuperObjects, another important class within the structure of KeyFinder. The SuperObject is, like the Map class, the overarching class responsible for housing the universally required methods and variables for each specific object class. The SuperObject stores the images that an instance might have, its name, collision, isOpen and isHit statuses (represented by Boolean variables), the colour that the object might have, its position (worldX and worldY) and the rectangle object which represents where its collision boundaries are.

There are three subclasses which are derived from the SuperObject class, the SO\_Chest, SO\_Door and SO\_Key classes. These classes define the unique properties of the Chest, Door, and Key objects respectively. The images for these objects are loaded using the same InputStream and Buffered image techniques to be found within the getTileImage method. Some subclasses of the SuperObject, such as the doors and keys, have several different possible colour's (yellow, blue, red, and purple for the key). This information is determined by the switch case within the subclass's constructor. An integer (given in the arguments) is used to tell subclass which type of key a given instance is, if the arguments say 0, then the key is a yellow key and will load the yellow key image, if 1, then blue and so on. These subclasses are created and placed into the world as needed by a map classes setObject method, a process which we have already explored in the previous section.

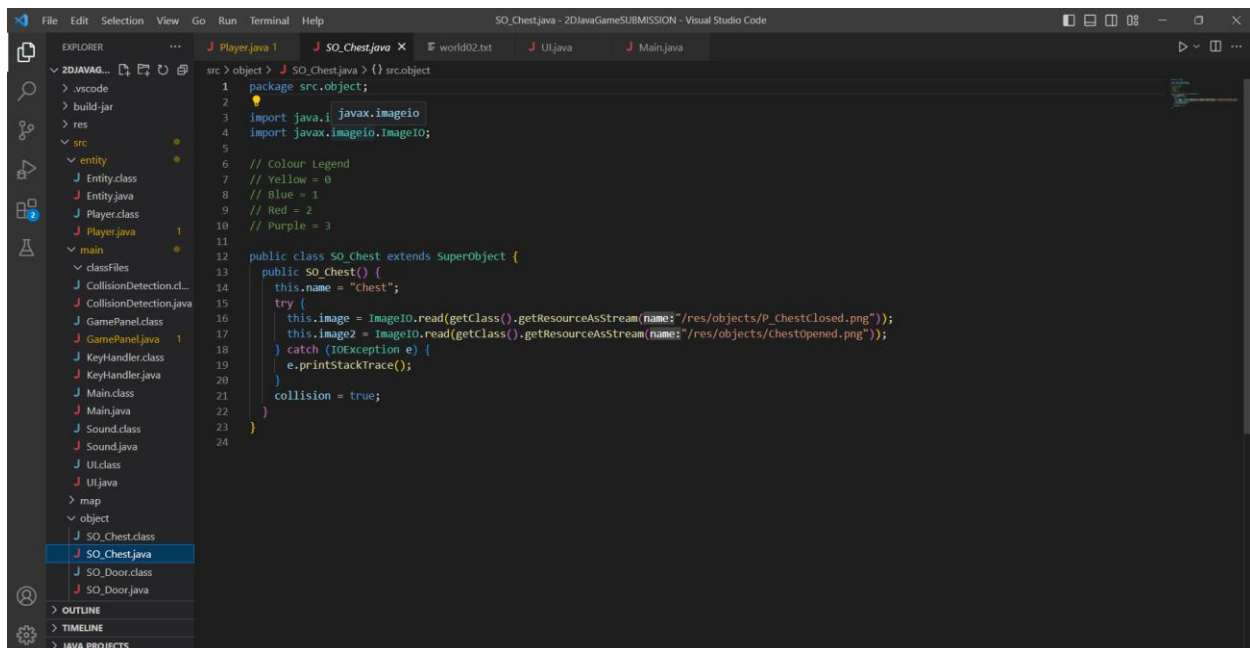


Figure 5 SO\_Chest class.

Next, we shall discuss how the player interacts with these objects and the game world itself. That's right, it's time to talk about collisions.



## Collisions

The player's task in KeyFinder is to scour the map for keys which he uses to unlock the doors blocking his path and secure the treasure. It wouldn't be much of a job, however, if the player could simply walk through the various obstacles he is faced with. Hence the need for a process which handles collisions between the player and the game world. Collisions between entities, tiles and objects are all processed within the CollisionDetection class.

First let's discuss the tile collision system. You might recall in the tile section that the various game tiles are assigned a true or false Boolean (collision) that determines whether the tile is a solid tile. If this Boolean is set to true, then the player cannot walk through any instance of this tile on a given map. To check if such a collision might occur between a solid tile and an entity, the checkTile method, based off of the entity's current direction, checks if the surface of the entities collision box will hit a tile that is solid if the entity continues to move in the same direction. If the checkTile method determines that a collision is going to occur the setCollision method is called. This method tells the entity that it can no longer move in the direction that it was prior by setting the entities collisionOn Boolean to be true, thus prohibiting movement. However, if the entity switches directions and collision can occur in the new direction, the collisionOn variable is deactivated and movement can resume.

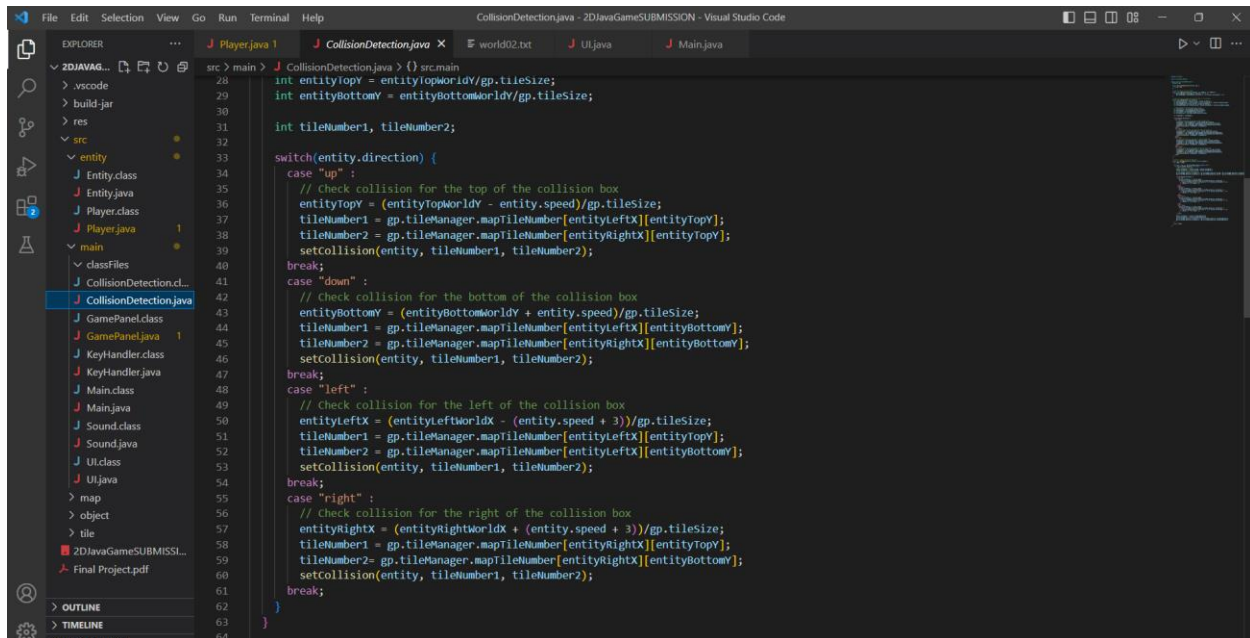


Figure 6 Tile collisions.

In addition to processing collisions between entities and tiles, the CollisionDetection class also handles the collisions between the game's entities and objects. Whereas in the checkTile method we only check the surfaces of the entity's collision box (to conserve processing power), within the checkObject method we scan through every object within a maps object array to see if there are any intersections between the entity and objects solidArea rectangles. This is accomplished by utilizing the .intersects method for the rectangle class. Because there are far less objects than tiles, this process doesn't use as much memory, and we can account for possible collisions with a far simpler chunk of logic. The checkObject class, like

the checkTile class, first determines the entities direction. If an intersection exists between an entity and an object a couple of tasks are performed to determine what to do with this interaction. Some objects such as locked doors and the chest are solid meaning that the player cannot pass through them, if the object the entity is currently colliding with is solid, then the entities collisionOn variable is set to true and the entity class will determine what to do with this information.

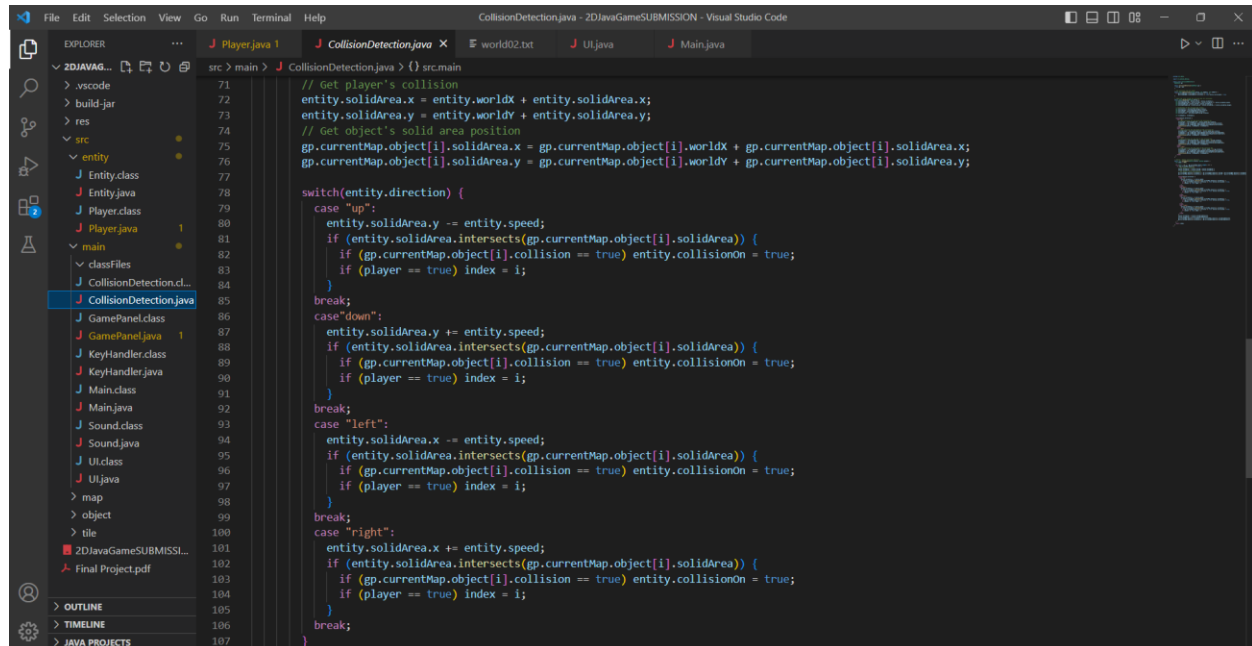


Figure 7 Object collisions.

Side Note: You might notice that within the checkTile method of the CollisionDetection class that the method checks a little further in a given direction than the player would actually travel. Early in development, KeyFinder had an issue where the player would get stuck in some tiles when he collided with them. This would happen because the player's hit box would technically still be colliding with a tile and thus wouldn't be able to move in certain directions even though he wouldn't travel into any new solid tiles. Thus, the checkTile method scans just a little further ahead of the hit box. This ensures that the collision box will never accidentally get stuck and that the player's movement is smooth.

## Entities

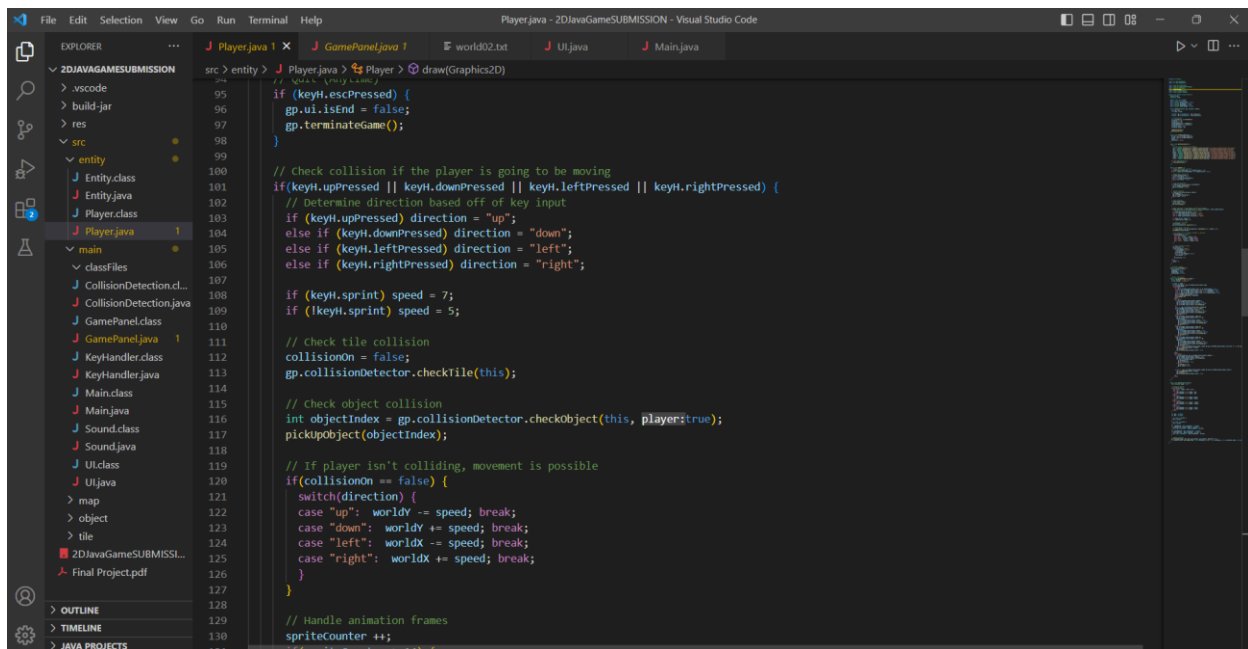
In KeyFinder, movable components are classified as entites. The Entity class serves as the overarching framework for all entities in the KeyFinder game. However, in KeyFinder there only exists one type of entity: The player.

The Player class is a subclass of the Entity class and thus inherits the following attributes: the player possesses worldX and worldY coordinates, an integer representing his speed, a rectangle object which describes the dimensions of his hitbox (the same type of object is used for object hitboxes), BufferedImages representing the sprites for movement directions, a string which stores the players current direction, as well as sprite counter and sprite number integers. The are properties that all entities possess. Upon the initialization of the Player class, these properties are defined.



First, let us discuss how the player's inputs are processed in the player's update method. Using the KeyHandler class, an instance of which is created in the GamePanel and assigned to the Player class within the constructor statements, the various inputs for the player are processed. These inputs are detected and organized within a collection of if/else statements in the beginning of the update method. Recall that the update method is called once per-frame and thus is continuously executing during KeyFinder's runtime. The program constantly scans for player input. When it determines that an assigned keystroke has been pressed, the appropriate values are adjusted, usually a Boolean which tells the other parts of the Player class which state he is in.

The player's movement information is further processed within the player's update method, just below where inputs are received and determined. First the program determines whether the player is pressing one of the movement keys. If this is the case then the direction is determined, again, based off the appropriate keys. The direction exists as a String variable, so "up" is assigned if the player is pressing the up key (which is 'w'). Then the program checks if collisions are going to occur. First, it sets the collisionOn variable to be false, otherwise the player would never move, then it calls both the checkTile and checkObject methods in the CollisionDetection class. If neither of these methods return true, then the player is permitted to continue moving.



```

src > entity > J Player.java > Player > draw(Graphics2D)
95
96 if (keyH.escPressed) {
97     gp.ui.isEnd = false;
98     gp.terminateGame();
99 }
100
101 // Check collision if the player is going to be moving
102 if (keyH.upPressed || keyH.downPressed || keyH.leftPressed || keyH.rightPressed) {
103     // Determine direction based off of key input
104     if (keyH.upPressed) direction = "up";
105     else if (keyH.downPressed) direction = "down";
106     else if (keyH.leftPressed) direction = "left";
107     else if (keyH.rightPressed) direction = "right";
108
109     if (keyH.sprint) speed = 7;
110     if (!keyH.sprint) speed = 5;
111
112     // Check tile collision
113     collisionOn = false;
114     gp collisionDetector.checkTile(this);
115
116     // Check object collision
117     int objectIndex = gp collisionDetector.checkObject(this, player);
118     pickupObject(objectIndex);
119
120     // If player isn't colliding, movement is possible
121     if (collisionOn == false) {
122         switch (direction) {
123             case "up": worldY -= speed; break;
124             case "down": worldY += speed; break;
125             case "left": worldX -= speed; break;
126             case "right": worldX += speed; break;
127         }
128     }
129
130     // Handle animation frames
131     spriteCounter++;
132     if (spriteCounter > 30) {
133

```

Figure 8 Player movement.

Object interactions are also handled in the Player class, specifically within the pickupObject method. This complicated method is responsible for handling all the different responses that the player should have when interacting with a given object. Firstly, this method is called whenever the player collides with an object. When this occurs, the pickupObject method is called and the index of the object the player is colliding with is passed as an argument. Using this index, the pickupObject method determines what type of object the player has touched (done via determining the objects name). Depending on the type of object, and occasionally additional factors such as what keys the player possesses, different actions are performed. Say the player touches a yellow key. First, the pickupObject determines the object to be

a key by grabbing the name attached to the SuperObject at the index it receives in its arguments. Then, because the key is a yellow one, the player now possesses a yellow key (the hasYellowKey Boolean is set to true). Once this action is complete the key disappears, becoming a null object.

Another example of a player-object interaction processed by the pickUpObject method would be the case of a locked door. First, the program checks if the door is open, meaning that the player has already interacted with the object, or it was set to open by default during its placement on the map. If this is the case, the player is permitted to pass through without any trouble. However, if the door is closed then the program checks the player's inventory to see if he possesses the matching key. If he does, great, he is again permitted to pass through. If he doesn't then an error message is displayed telling the player that the door is locked, and the appropriate collision values are set.

The different objects all have different responses to interaction with the player. Keys disappear and are added to his inventory, doors are opened (done by changing the object sprite via the switchImage method) or remain locked if the player doesn't have the necessary keys, the same process occurs for the chest.

Finally, there is the player's draw method. This draw method is called within the repaint method of the GamePanel and is another essential component of the Player class. Firstly, the sprites themselves are loaded within the getPlayerSprite method. Then, using the direction and spriteNumeber variables, determined in the update method, the draw method is responsible for switching between these sprites to the appropriate ones for the player's direction and spriteNumeber. The logic for determining the player's sprite is as follows: When the player is initialized, his sprite is set to an idle sprite (this is accomplished in the setDefaultValues method). As the player moves and switches direction, his sprites are also updated to reflect the change. If the player is moving upwards, the sprites switch between the two different upward sprites via a sprite counter. For every direction there are two sprites that are switched back and forth while the player moves. If the player stops moving the sprite freezes on the last sprite it was assigned. Thus, the draw method depicts the player as he moves through the world.

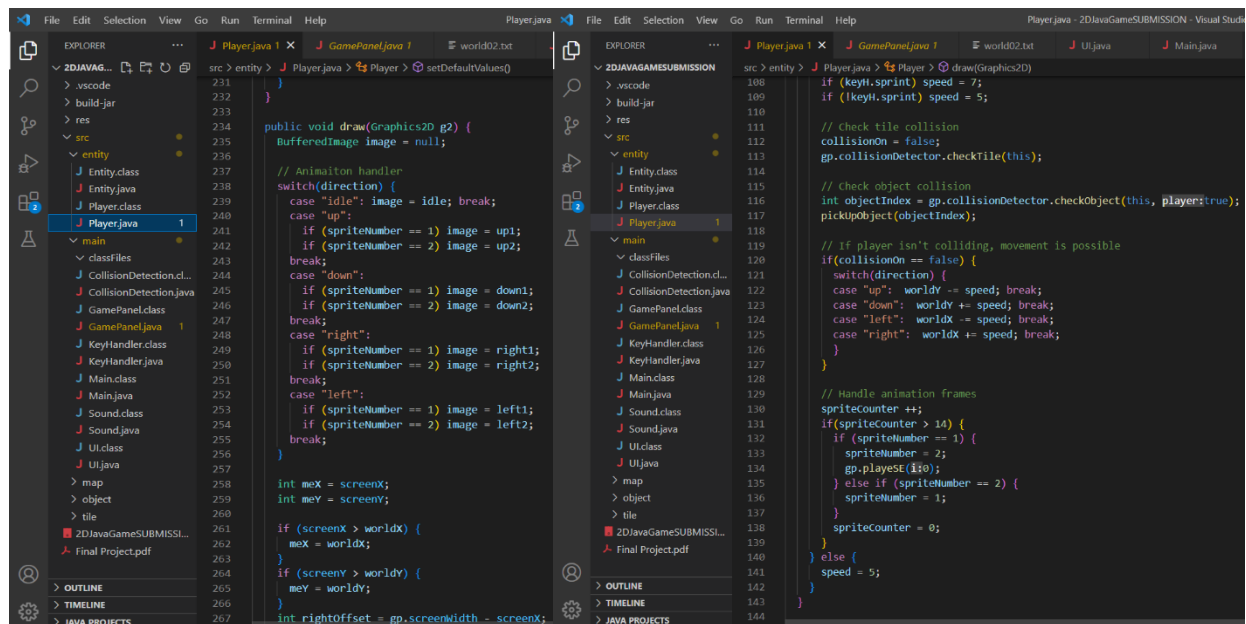


Figure 9 Left, switching between sprites. Right, movement and animation counter.

## Utility Classes

In this last section we'll discuss some of the other classes in KeyFinder. These classes perform the tasks necessary for collecting keystrokes, displaying UI images, and playing sound. As all these classes perform integral tasks, but for the sake of not over-complicating this document they've been lumped together into the Utility classes section.

We'll first discuss the KeyHandler class. This class, as you might expect, processes the game's input. The KeyHandler class, which is a subclass of the KeyListener, overrides its parent class keyPressed and keyReleased methods. In these methods, if a key which KeyHandler uses is pressed or released, it's activated status is set to either true or false. This status is used by the other classes (namely the UI and Player classes) to determine various elements, such as the visibility of certain windows in the UI class or the movement of the player entity in the Player class.

*Side Note:* Some cases have a unique way of being toggled and untoggled. When the key is pressed, the key is activated as with other keystrokes. However, an additional variable, known as the count variable, is also incremented. This means that the next time this key code is pressed, the case that handles the keycode checks to see if the count is greater than zero (the default value). If this is true, then the variable is deactivated. In this way, certain keys are activated and deactivated by repressing them instead of automatically being disabled upon their release.

The UI class is responsible for presenting the games information to the player. This task is accomplished by displaying text and icon images to the user via the methods in the UI class. To start, the UI images and fonts are imported within the UI class constructor. This process is in turn accomplished with the usage of two additional methods, the loadFont and loadUI methods. Both these methods import the external graphical components using InputStreams and a filepath. The text components of the UI, such as the message that displays every time the player receives a key or the prompt at the beginning of a new level, are managed with a series of methods. These methods, when called, display the different possible messages to the GamePanel's graphics 2D component. Where are these methods called? Well, the draw of course.

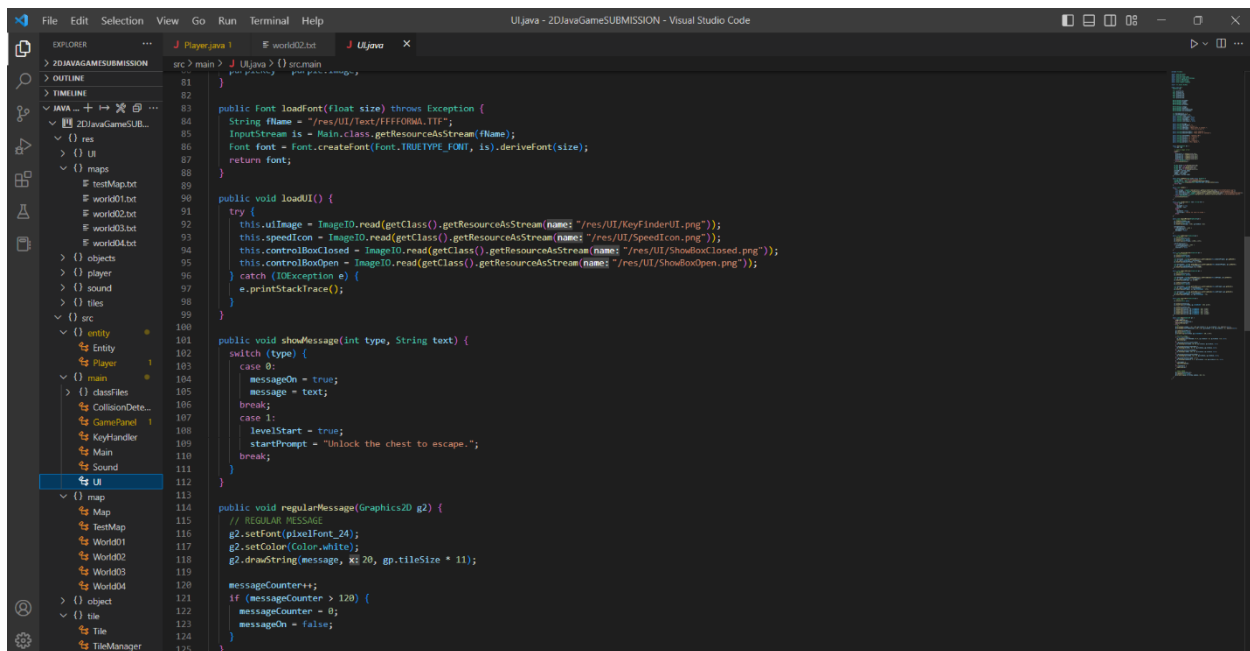


Figure 10 UI class loadFont, loadUI and some of the message methods

In addition to displaying text to the screen, the UI class draw method also handles the drawing process of the UI components. The GamePanel's Graphics2D component is assigned in the methods arguments and thus all the fonts and images are displayed within the KeyFinder window alongside the maps, player, and the game objects. KeyFinder determines what elements of the UI to display by checking what state the game is currently in. If the player has unlocked the chest on the final level, then the isEnd state is triggered and only the UI elements required at this stage are displayed. If the player unlocks a chest on a regular level, then the isLevelEnd state is triggered and only these elements are displayed. If the player is currently completing a level, then the regular UI state is put into effect. In this regular state, the player can access the control menu, see what level he is currently on, and view whether he is sprinting as well as what keys he has through a variety of image icons.

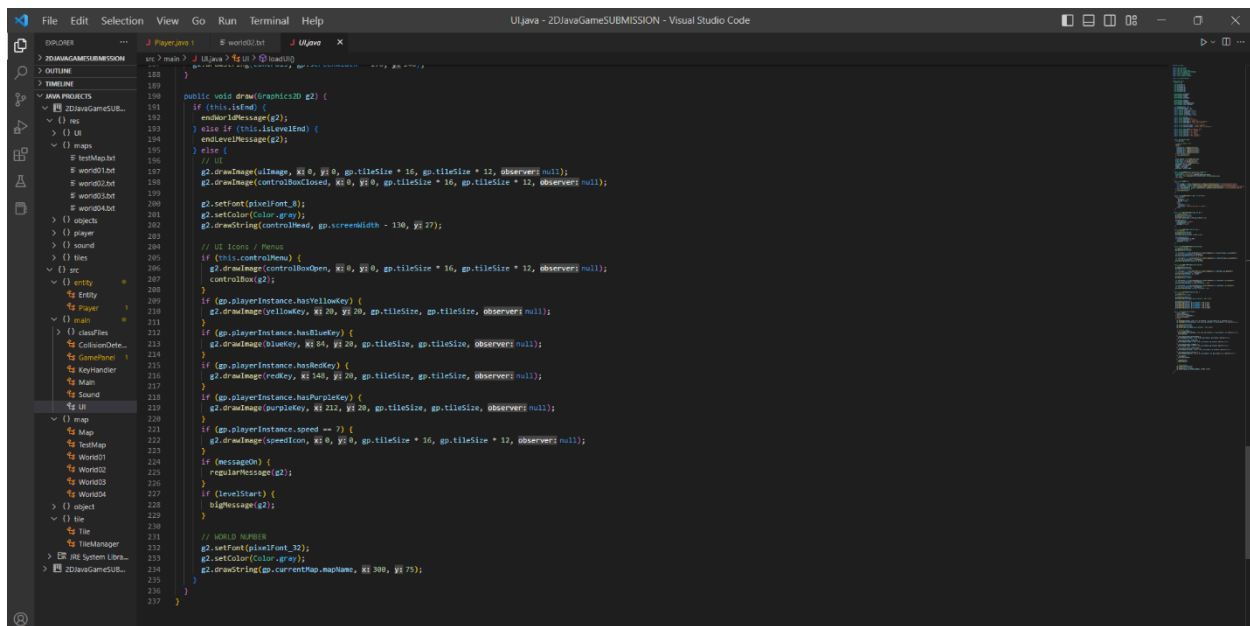


Figure 11 UI class draw method.

Lastly, we'll discuss the Sound class. In the Sound class constructor, the various sound files, which are of the .wav format, are stored into a Uniform Resource Locator array named soundURL. In the setFile method, a necessary sound file is located within the soundURL array via an integer that is passed in the methods arguments. Once the file is found within the soundURL array, a clip is created from it. The status of this clip is controlled by the three other methods found inside the Sound class. Clips are started with the play method, stopped with the stop method, and can be continuously looped with the loop method.

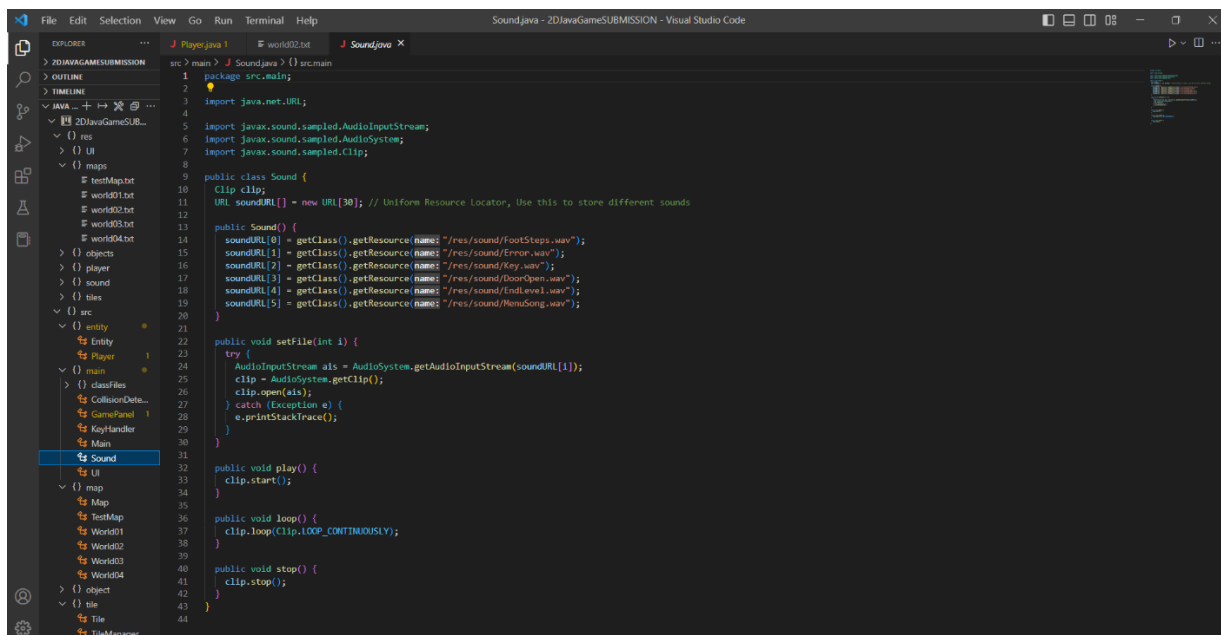


Figure 12 Sound class overview.

In the GamePanel class, the actual methods which trigger music / sound effects within the game are found. Two instances of the Sound class are instantiated along with the other KeyFinder classes, one for music and the other for regular sound effects. The playMusic method plays whatever sound file is requested in the method's arguments using the music instance of the Sound class. First the index of the sound file is set using the setFile method. Then the Sound class starts the music with the play method before proceeding to loop the track with the loop method. Whatever clip is started by triggering this method must be stopped using the stopMusic method (which only contains the stop method), or else it will loop forever. Sound effects are triggered using the sound effects instance of the Sound class. In the playSE method a clip is selected by passing an integer which represents a sound file index. Then, the clip is started using the play method. Once all the clip's duration is complete, it will automatically stop.

And there we have it, the utility classes and how they work.

## Conclusion

Congratulations! You've reached the end. I would like to take a moment to thank you for taking the time to read through this rather lengthy document. I've done my best to explain everything as concisely as my abilities can provide while also striving to provide a sufficient explanation of KeyFinder's functionality. This project has been an outlet for my game design passion and a stone upon which to sharpen my abilities in computer science. I am both grateful to have been able to share the workings of this project with you and demonstrate my skills in the subject of computer science.

The focus of this project was to delve into one of the major units that the class studied during the course duration and create a project that creatively demonstrated our competency regarding our chosen unit. In terms of demonstrating my programming knowledge I ventured to incorporate the skills that I had acquired in several of the course's units of study. First and foremost, I wanted to demonstrate my understanding of our object-oriented programming units. There are many classes found within the game. Indeed, the entire structure of the game is a collection of super classes and subclasses which work together with great efficiency to accomplish the different tasks required for the game's function. Furthermore, these super classes and subclasses heavily utilize my understanding of the inheritance and polymorphism concepts that our class studied. Classes such as the Player class inherit various methods and variables from its superclass, the Entity class for the case of the Player. Additionally, I utilized the material we studied in the text Input / Output unit to handle the games external graphics and data. The player's textures, tiles, audio files, and maps are all stored in external files which must be imported into the game. All of this was accomplished to demonstrate my understanding of what it was we had studied during the course.

On the creative side of things, everything that you see and hear in the game was created by me, with the exception of the game's font which I downloaded from online. Now, I'm no artist or musician, so these elements were a bit of a stretch for me. Pixel art is surprisingly hard to create, and you might be surprised by the number of times that I re-drew the player sprites and some of the tiles. What matters is that the game doesn't look terrible. For the music, this was the area where I stretched the furthest. My only prior musical experience with the creation of music are the few songs that I know on the piano. Never-the-less, I was determined to try and create sound effects and music for my game. For many



games, it is the music and sound effects for which they are most remembered for, so this was a fair source of pressure for me. Inspired by classical titles such as Mario and Zelda, I got to work in the online music creator BeepBox. All told, the artistic elements of the KeyFinder are my attempt to push myself a little further as a developer and demonstrate that I could produce the artistic elements required for this little project.

But my goal wasn't merely to demonstrate that I understood what we had studied and that I could incorporate elements of creativity into the project. I wanted to expand my programming horizons by creating a project which was robust and well written. Every aspect of the project was written to be malleable and reusable with many processes being reimplemented or shared by different aspects of the game. My previous forays into game design have always felt chaotic and disorganized in terms of design and programming. While the code in my past projects certainly functioned, it was by no means organized. Classes and methods would be written all over the place, and conventions for naming methods and variables were rarely used. My past projects have been messy to say the least. KeyFinder is my attempt to push my boundaries as a programmer, to approach the design of a project with method and organization, and to create a robust program, hence such a focus on class structures. In the future I know that simply writing code that solves a problem will not suffice, thus this project was my attempt to develop my organizational capacity in terms of programming and software design. I would say that this goal was accomplished with success. This project incorporates all the skills that I've acquired up to this point, showcases my capacity to be creative, and is a robust little game. KeyFinder is the best program that I have created, and I am happy to have been able to showcase it.

Thank you,

Will Otterbein.