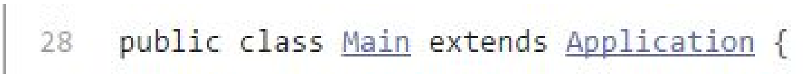
Criterion C - Product Development

The following techniques were used to develop the product:

- JavaFX graphics and key listeners
- JavaFX AnimationTimer
- Development of a collision detector method
- Parent classes
- Method for creating levels

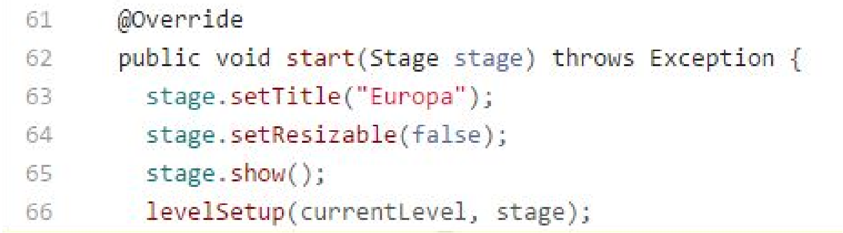
1. JavaFX graphics and key listeners

As a game, the product needs a way to display images on the screen when it executes. Oracle, the creator of Java, has also developed a library called JavaFX, which allows the developer to create a program, written in Java code, with an interface (separate from the command line). The program is a subclass of Application, which is the framework class in the JavaFX library.



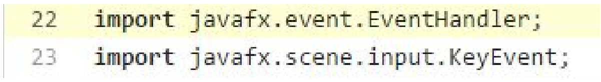
```
public class Main extends Application {
```

The program must include the start method, inherited from Application, which allows the program to generate a Stage and create the window, on which the content will be displayed. The integral parts of the start() method are shown below:



```
@Override
public void start(Stage stage) throws Exception {
  stage.setTitle("Europa");
  stage.setResizable(false);
  stage.show();
  levelSetup(currentLevel, stage);
```

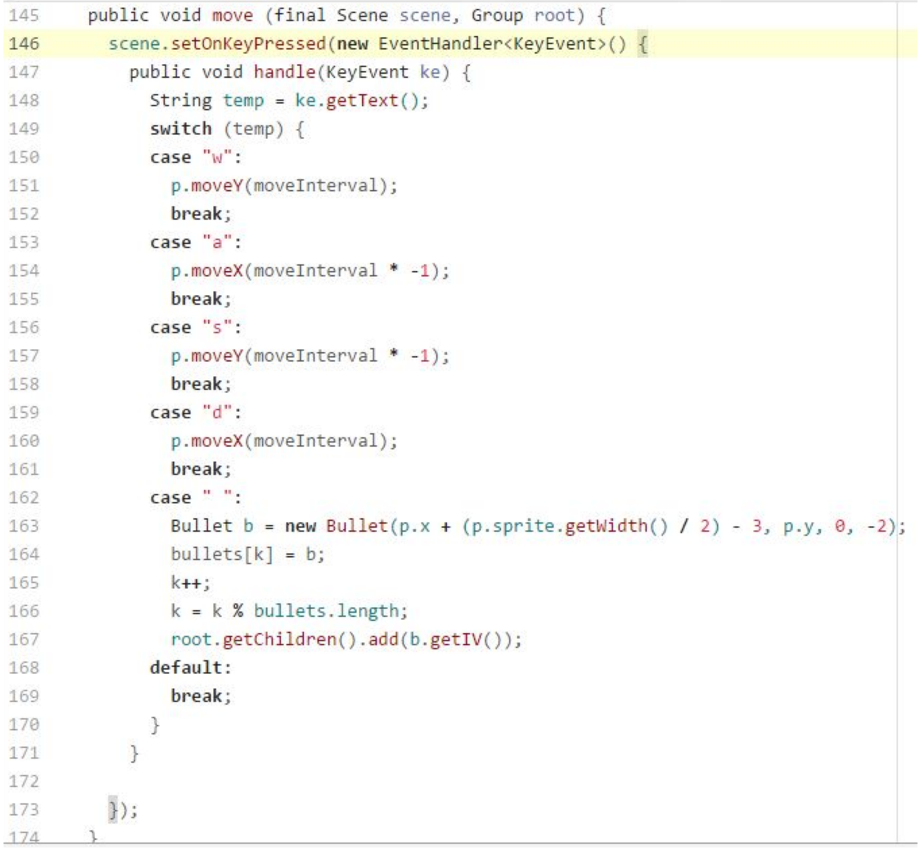
The product also needs to interact with the hardware. In order for the program to be able to detect keystrokes, it imports the EventHandler and KeyEvent classes from the JavaFX library.



```
import javafx.event.EventHandler;
import javafx.scene.input.KeyEvent;
```

EventHandler provides a framework for managing and processing certain events on Nodes, which are objects in the interface. The EventHandler class has a method handle(), which requires an input of an event type, in this case a KeyEvent.

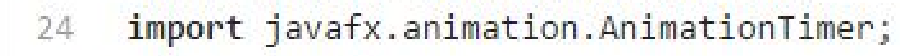
The code below demonstrates how EventHandler and KeyEvent are implemented in the product. This method, move(), detects keystrokes. If the key pressed is “w”, “a”, “s”, or “d”, the player is translated in the corresponding direction, by a certain amount moveInterval. If the key pressed is the space bar, the game spawns a bullet at the player’s location, which begins moving upwards at a constant rate. If a key other than these 5 is pressed, nothing happens.



```
public void move (final Scene scene, Group root) {
  scene.setOnKeyPressed(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
      String temp = ke.getText();
      switch (temp) {
      case "w":
        p.moveY(moveInterval);
        break;
      case "a":
        p.moveX(moveInterval * -1);
        break;
      case "s":
        p.moveY(moveInterval * -1);
        break;
      case "d":
        p.moveX(moveInterval);
        break;
      case " ":
        Bullet b = new Bullet(p.x + (p.sprite.getWidth() / 2) - 3, p.y, 0, -2);
        bullets[k] = b;
        k++;
        k = k % bullets.length;
        root.getChildren().add(b.getIV());
      default:
        break;
      }
    }

  });
}
```

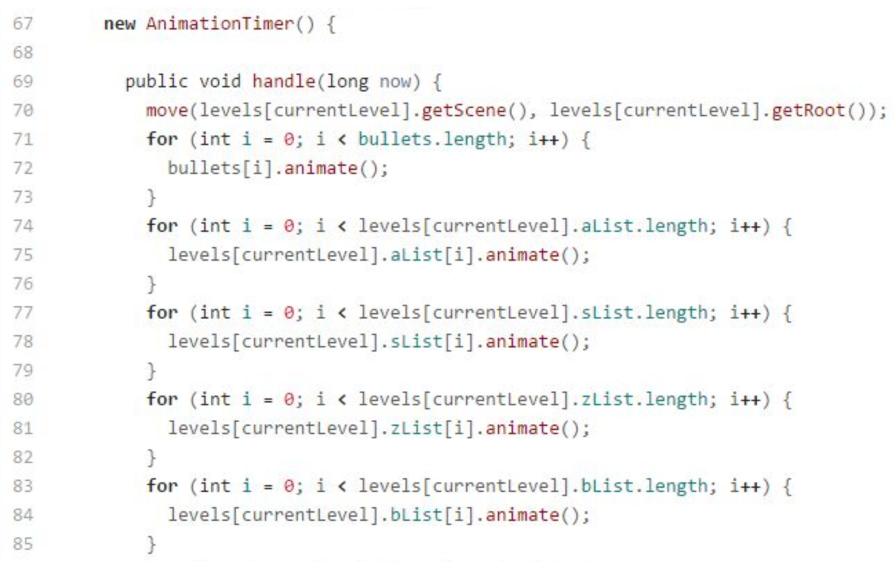
2. JavaFX AnimationTimer

Due to the style of the game, objects must be moving around the screen at all times. Thus, the product implements the AnimationTimer class to animate the player, enemies, and bullets.



```
import javafx.animation.AnimationTimer;
```

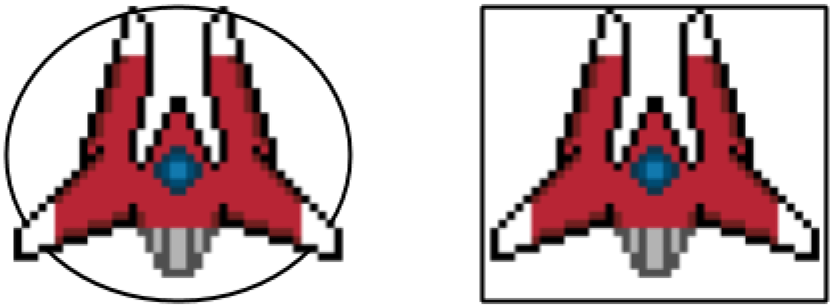
When an instance of AnimationTimer is created, it must also contain the method handle(), which contains the code that loops while the AnimationTimer runs. The method .start() begins the animation, and .stop() ends it. In the code segment below, the game animates the player, then the bullets, followed by each group of enemies present in the level. If a certain type of enemy is absent from the level, the level will not try to animate them.



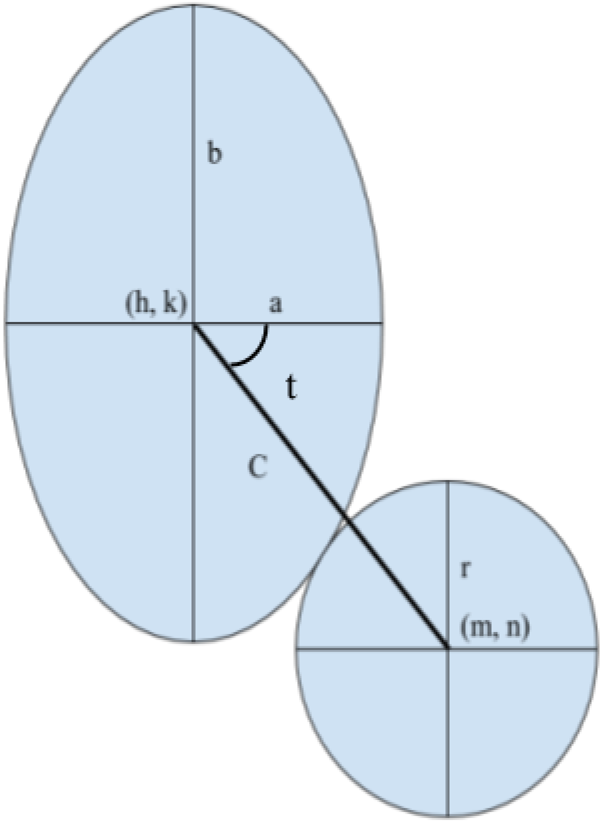
```java
new AnimationTimer() {

  public void handle(long now) {
    move(levels[currentLevel].getScene(), levels[currentLevel].getRoot());
    for (int i = 0; i < bullets.length; i++) {
      bullets[i].animate();
    }
    for (int i = 0; i < levels[currentLevel].aList.length; i++) {
      levels[currentLevel].aList[i].animate();
    }
    for (int i = 0; i < levels[currentLevel].sList.length; i++) {
      levels[currentLevel].sList[i].animate();
    }
    for (int i = 0; i < levels[currentLevel].zList.length; i++) {
      levels[currentLevel].zList[i].animate();
    }
    for (int i = 0; i < levels[currentLevel].bList.length; i++) {
      levels[currentLevel].bList[i].animate();
    }
```

3. Hitbox Collision Detection Algorithm

In order for the game to function, there needed to be a way to determine roughly when objects were colliding (when bullets were colliding with enemies). Thus, the game includes invisible hitboxes around each bullet and enemy:



The hitboxes are ellipses, instead of rectangles, to minimize the average distance between the hitbox boundary and the object's edge while still being a geometric shape and easy to use in calculations. Hitboxes have a vertical diameter of the object's height and a horizontal diameter of the object's width. All collisions in the game follow the following model:



Every collision is between an enemy (elliptical) and a bullet (perfect circle). With these dimensions, the two ellipses have the following equation definitions:

$$(x-h)^2 + \frac{a^2}{b^2}(y-k)^2 = a^2$$

$$(x-m)^2 + (y-n)^2 = r^2$$

The distance between the two ellipses at any given moment is the distance between their centers:

$$C = \sqrt{(h-m)^2 + (k-n)^2}$$

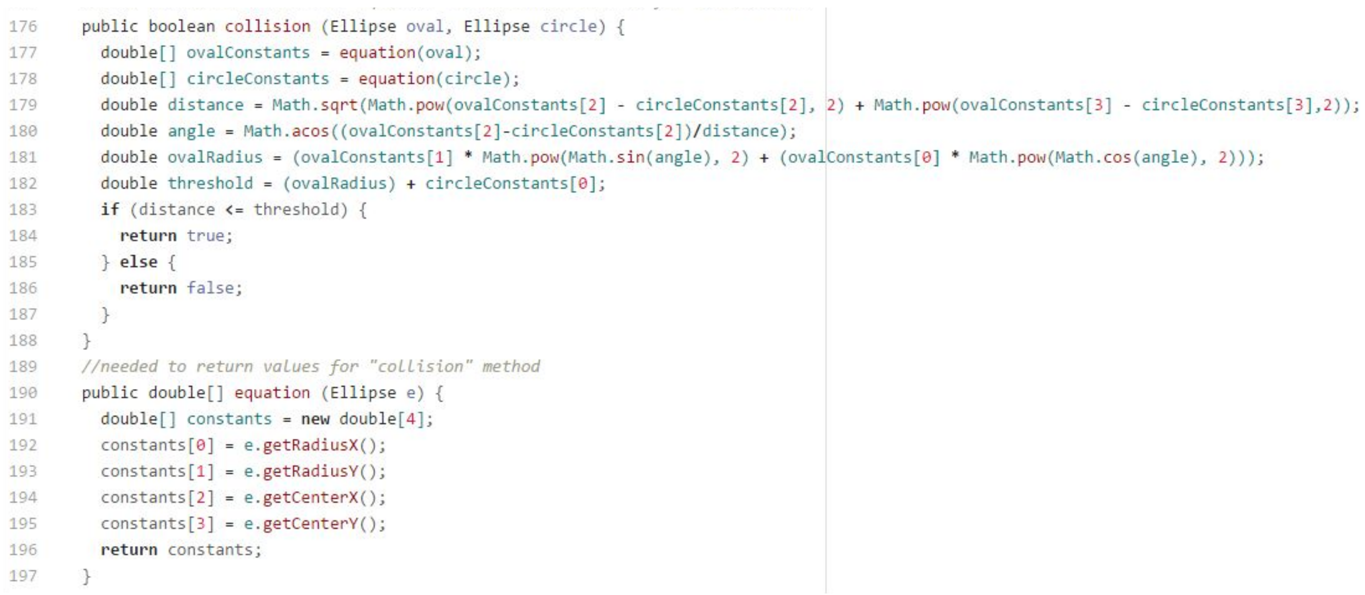
The angle between the centers can also be calculated, which determines at what angle the two are colliding:

$$t = arccos(\frac{m-h}{C})$$

The radius of the ellipse at that angle can now be calculated:

$$q = b \cdot sin^2(t) + a \cdot cos^2(t)$$

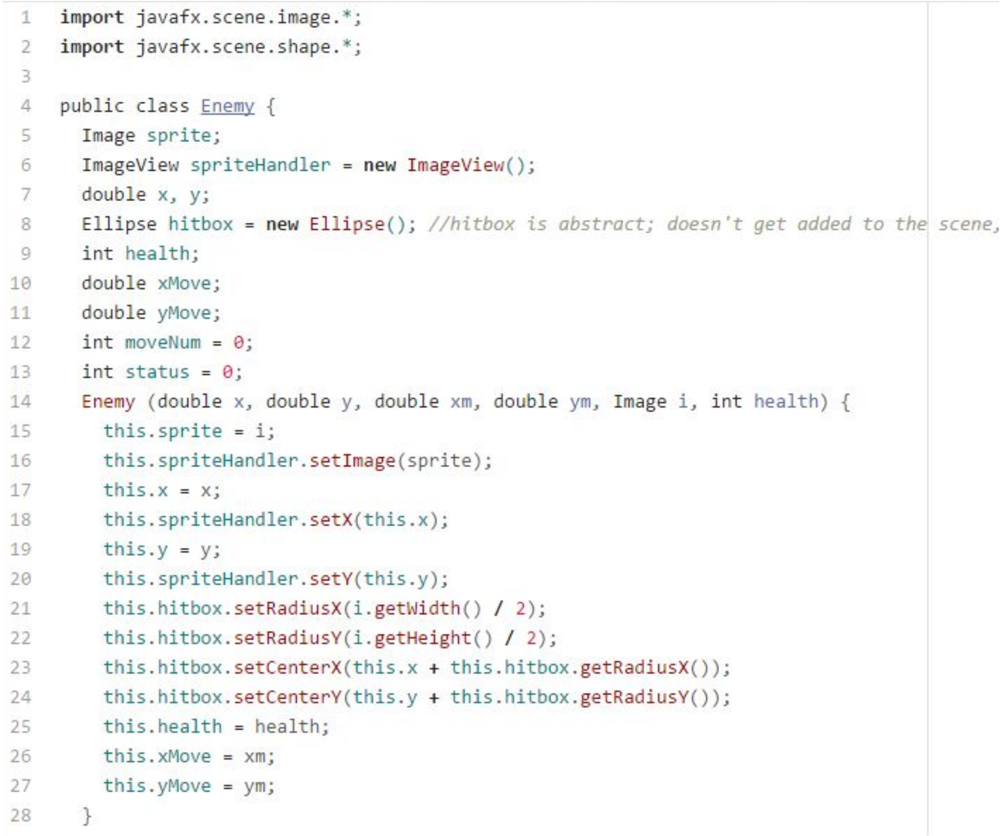
The collision threshold can then be determined as $d = q + r$, the sum of the radii of both ellipses. If the distance, *C*, is less than or equal to *d*, then a collision has occurred. Below is the implementation in the product:



```
  public boolean collision (Ellipse oval, Ellipse circle) {
    double[] ovalConstants = equation(oval);
    double[] circleConstants = equation(circle);
    double distance = Math.sqrt(Math.pow(ovalConstants[2] - circleConstants[2], 2) + Math.pow(ovalConstants[3] - circleConstants[3],2));
    double angle = Math.acos((ovalConstants[2]-circleConstants[2])/distance);
    double ovalRadius = (ovalConstants[1] * Math.pow(Math.sin(angle), 2) + (ovalConstants[0] * Math.pow(Math.cos(angle), 2)));
    double threshold = (ovalRadius) + circleConstants[0];
    if (distance <= threshold) {
      return true;
    } else {
      return false;
    }
  }
  //needed to return values for "collision" method
  public double[] equation (Ellipse e) {
    double[] constants = new double[4];
    constants[0] = e.getRadiusX();
    constants[1] = e.getRadiusY();
    constants[2] = e.getCenterX();
    constants[3] = e.getCenterY();
    return constants;
  }
```

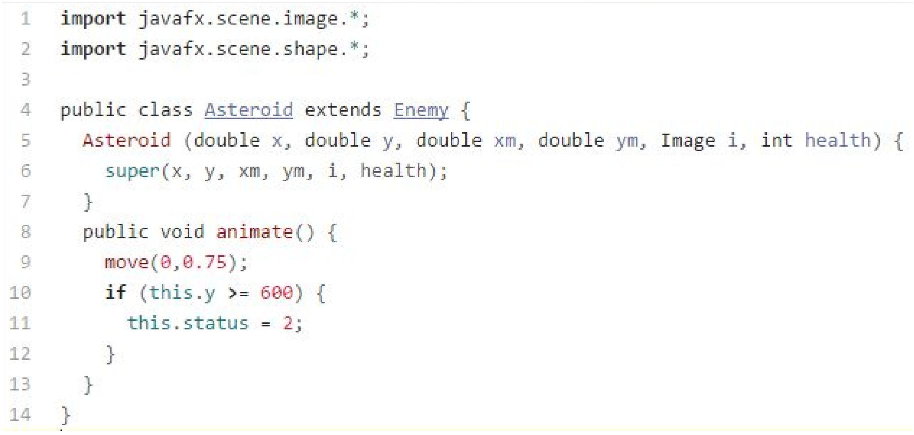
4. Parent classes

The premise of the game requires multiple variations of enemies, which all have the same basic qualities, but with minor differences. Thus, all types of enemies stem from the same parent class, Enemy:



```
import javafx.scene.image.*;
import javafx.scene.shape.*;

public class Enemy {
  Image sprite;
  ImageView spriteHandler = new ImageView();
  double x, y;
  Ellipse hitbox = new Ellipse(); //hitbox is abstract; doesn't get added to the scene,
  int health;
  double xMove;
  double yMove;
  int moveNum = 0;
  int status = 0;
  Enemy (double x, double y, double xm, double ym, Image i, int health) {
    this.sprite = i;
    this.spriteHandler.setImage(sprite);
    this.x = x;
    this.spriteHandler.setX(this.x);
    this.y = y;
    this.spriteHandler.setY(this.y);
    this.hitbox.setRadiusX(i.getWidth() / 2);
    this.hitbox.setRadiusY(i.getHeight() / 2);
    this.hitbox.setCenterX(this.x + this.hitbox.getRadiusX());
    this.hitbox.setCenterY(this.y + this.hitbox.getRadiusY());
    this.health = health;
    this.xMove = xm;
    this.yMove = ym;
  }
```

Enemy contains most of the code needed for the enemies to function. The only addition in each subclass of Enemy is the code determining its animation path:

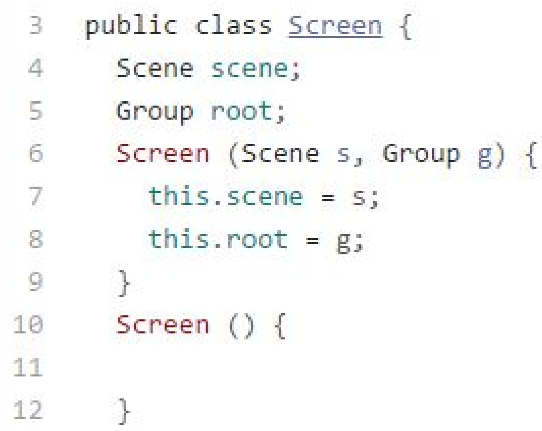


```java
import javafx.scene.image.*;
import javafx.scene.shape.*;

public class Asteroid extends Enemy {
  Asteroid (double x, double y, double xm, double ym, Image i, int health) {
    super(x, y, xm, ym, i, health);
  }
  public void animate() {
    move(0,0.75);
    if (this.y >= 600) {
      this.status = 2;
    }
  }
}
```

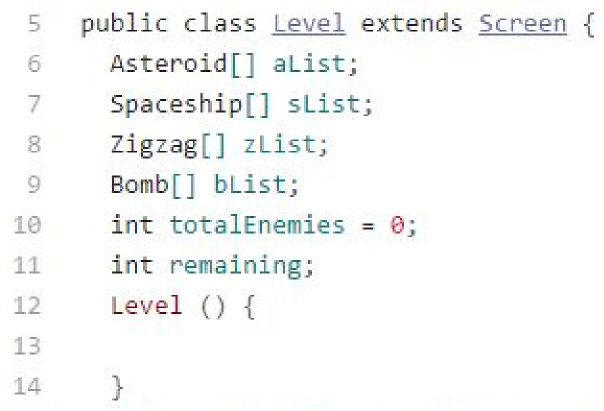
The only other difference, the sprites, are assigned during initialization of the game.

5. Creation of levels

Another important facet of the product is the different levels of the game. Levels are similar, but differ in the enemies they carry, and the quantities of these enemies. The Level object is a subclass of Screen, which is used for images on-screen that are not levels, such as the win and loss screens.

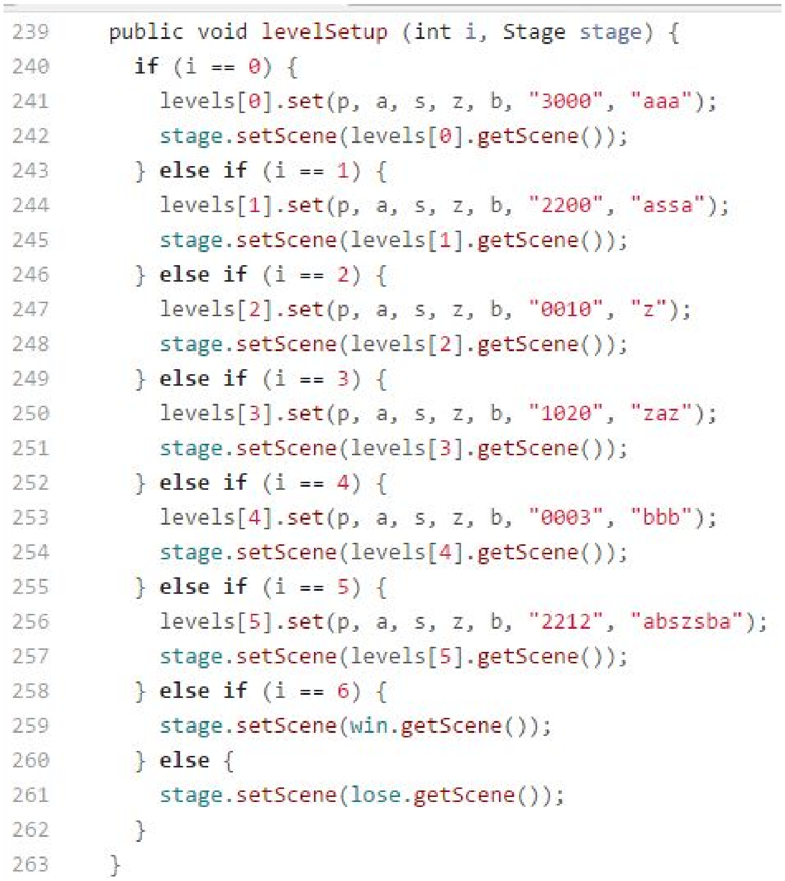


```java
public class Screen {
  Scene scene;
  Group root;
  Screen (Scene s, Group g) {
    this.scene = s;
    this.root = g;
  }
  Screen () {

  }
```



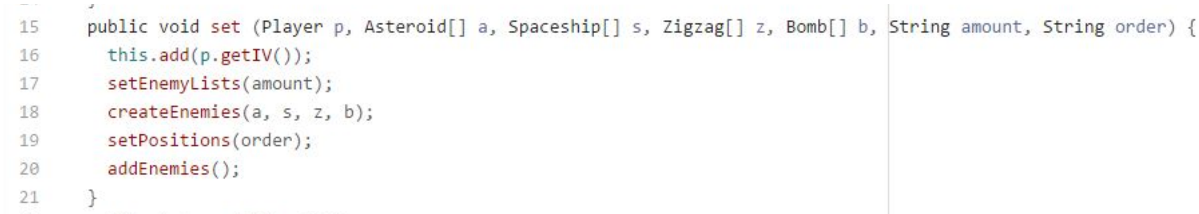
```java
public class Level extends Screen {
  Asteroid[] aList;
  Spaceship[] sList;
  Zigzag[] zList;
  Bomb[] bList;
  int totalEnemies = 0;
  int remaining;
  Level () {

  }
```

Levels are initialized as empty classes, but are assigned values and objects in the levelSetup() method in the Main class:



```
  public void levelSetup (int i, Stage stage) {
    if (i == 0) {
      levels[0].set(p, a, s, z, b, "3000", "aaa");
      stage.setScene(levels[0].getScene());
    } else if (i == 1) {
      levels[1].set(p, a, s, z, b, "2200", "assa");
      stage.setScene(levels[1].getScene());
    } else if (i == 2) {
      levels[2].set(p, a, s, z, b, "0010", "z");
      stage.setScene(levels[2].getScene());
    } else if (i == 3) {
      levels[3].set(p, a, s, z, b, "1020", "zaz");
      stage.setScene(levels[3].getScene());
    } else if (i == 4) {
      levels[4].set(p, a, s, z, b, "0003", "bbb");
      stage.setScene(levels[4].getScene());
    } else if (i == 5) {
      levels[5].set(p, a, s, z, b, "2212", "abszsba");
      stage.setScene(levels[5].getScene());
    } else if (i == 6) {
      stage.setScene(win.getScene());
    } else {
      stage.setScene(lose.getScene());
    }
  }
```

For each level in the list of levels, an array, the method gives the level the player, the list of each type of enemy, and two Strings which tell the Level which and how much of each enemy to include, and in what order. The method set(), within the Level class, processes the information and adds the necessary components to each Level:



```
public void set (Player p, Asteroid[] a, Spaceship[] s, Zigzag[] z, Bomb[] b, String amount, String order) {
  this.add(p.getIV());
  setEnemyLists(amount);
  createEnemies(a, s, z, b);
  setPositions(order);
  addEnemies();
}
```

Word Count: 864