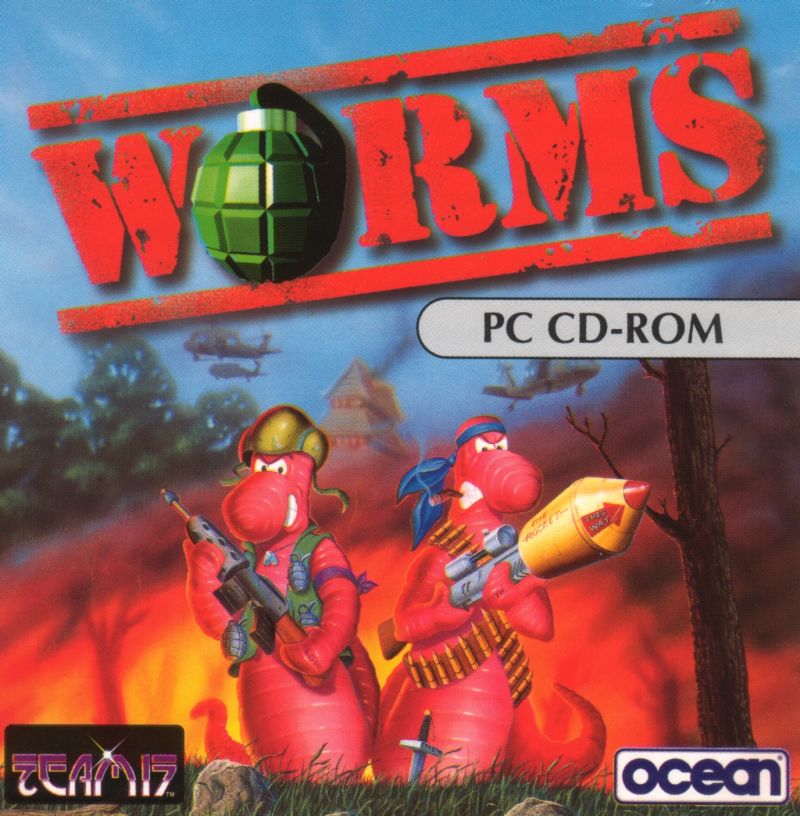
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Battle Tank  &  Lazarus |  | |  |  | |
| Team: 03 – CSC413 – Fall2018  Albert G Shevchuk  Peter Le  https://github.com/csc413-02-sp18/csc413-tankgame-Team03  https://github.com/csc413-02-sp18/csc413-secondgame-Team03 |

# Introduction

*“Happiness lies in the joy of achievement and the thrill of creative effort” – Franklin D. Roosevelt*

**Project Overview:**

Design and development of a realistic software applications. Object-oriented techniques including encapsulation, inheritance, and polymorphism as mechanisms for data design and video game solution. This project was designed to test your limits and your ability to work with a partner.

**Tank Game General Idea:**

 When we began working on Battle Tank, a swarm of ideas hit our minds like a train. After some discussion, it was decided that what we wanted out of this game was to be distinctive, have better graphics and create an artillery genre game such as Gunbound and Worms. We didn’t want a bird’s view of the game at all, the way we designed our game was to be viewed from the side. This meant that a lot of our requirements would have to be altered and with our professor’s blessing, the project launched at full speed. At first, our biggest challenge was to figure out a way for the projectile to fly at an arc. The way we split up the work was also very interesting; however, we expected this challenge. We had an idea of who was going to be in charge of what function and classes; however, as the game evolved, we quickly understood the sectors that we each knew better. With that, we started synchronizing our code and crafted a beautiful masterpiece.

**Lazarus Game General Idea:**

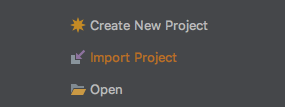
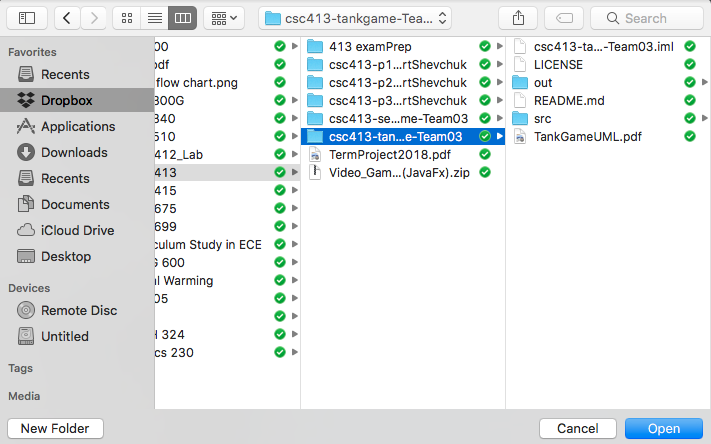
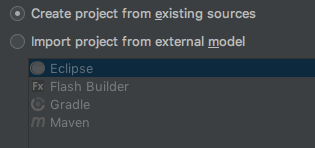
The second game came with ease. As we looked through the options that we had for the second game, we quickly were satisfied with Lazarus because we saw how similar the game we created was. With that, we knew that most of the classes would have to be erased and some basic classes like GameObject and GameEngine would stay since it could all be reused and refabricated. One interesting key component we realized from the beginning is that different boxes had different priority on which box gets destroyed when it encounters another box. This meant that there was supposed to be some kind of mechanism that was constantly checking the arrays. So right away, the gears started turning and once again, the aroma of coding filled the room.

# Development Environment

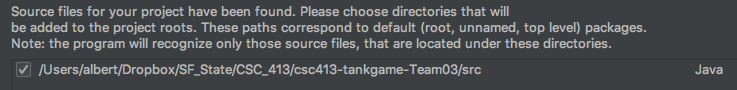
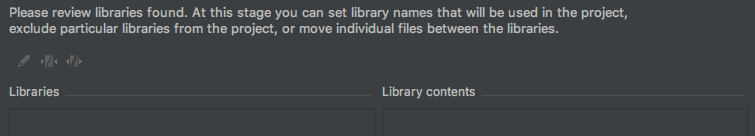
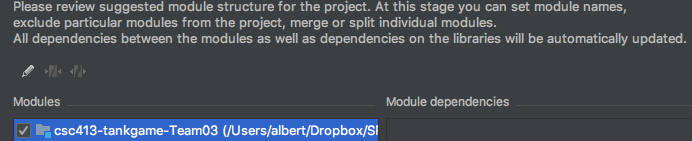
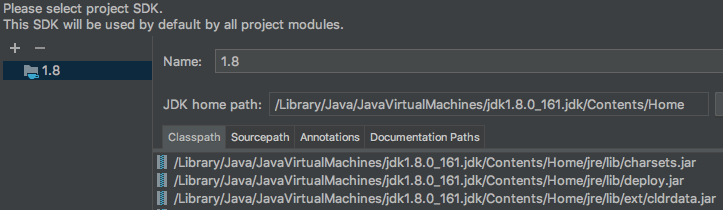
 The software platform for the creation of both games was chosen to be JavaFx, simply because of how the graphics looked. With that intact, IntelliJ IDEA 2018.1 (Ultimate Edition) was our Integrated Development Environment of choice. The Java Runtime Environment that was used: 1.8.0\_152-release-1136-b20 x86\_64 along with OpenJDK 64-Bit Server VM by JetBrains s.r.o as the Java Virtual Machine. In comparison to others, we really enjoyed working with Fx and towards the end started to like the kinks and quirks of the software.

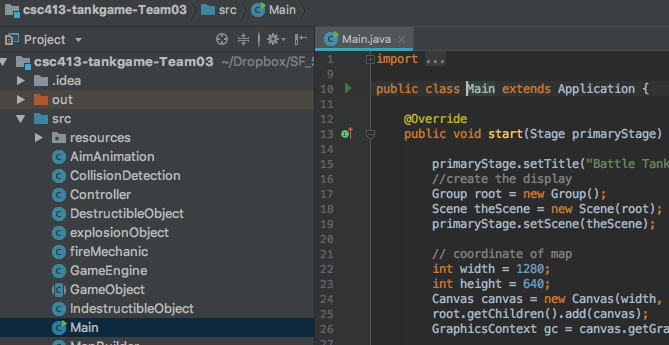
# Building and Running

**Steps:**

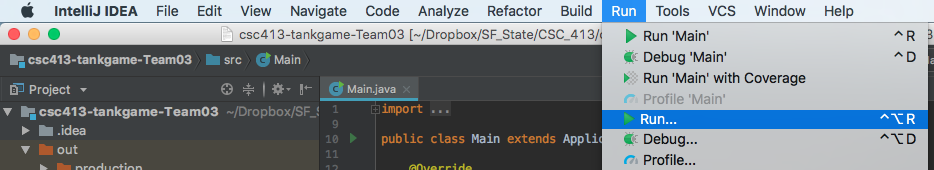
1. Download and Install the IntelliJ IDE on to your machine.
2. Launch the installed application click on “import project”.
3. Find and select your directory (example uses csc413-tankgame-Team03) and click “open” on the bottom right.
4. Then you need to make sure “create project from existing sources” is checked off and click “next” on the bottom right.
5. Feel free to rename the project under “project name:” Otherwise, double check that the project folder is correct, then continue by pressing “next” on the bottom right.

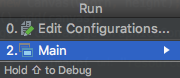


1. This is where IntelliJ will find the src folder with all of the Java files, make sure they are found and proceed by clicking “next” on the bottom right.
2. You won’t need to import any extra libraries so proceed by pressing next on the bottom right.
3. At this step, you will be asked to import your Modules. Make sure the game’s directory is selected and proceed by pressing “next” on the bottom right.
4. Make sure here that IntelliJ has identified the correct SDK version and proceed by clicking “next” on the bottom right.
5. Last but not least, there are not going to be any additional frameworks so proceed by clicking “finish” on the bottom right.
6. Since this will be the first time running this game, runtime configurations that are supposed to start from main have not been configured yet and this is one of the ways we are going to make it work. Give it some time to load, then on the left side of the program expand the “csc413-tankgame-Team03” folder and also expand the “src” folder. Double click on the file called “main.”



1. Finally, you will need to find the menu bar then navigate to “run” and click “run” that appears in the dropdown menu.



1. The configuration settings will pop up. Make sure “main” is highlighted and proceed by hitting “enter” on your keyboard. The game should compile and run now.

# Game Logistics

**Objectives & Rules:**

The primary goal of Battle Tank is to destroy the other opponent. Since this is a turn based game, the timer dictates who’s turn it is. Once the timer runs out, that players side and rocket booster moving functions are locked and what remains are the cannon controls with the firing of the projectile. The explosion of whose projectile fired finishes the turn at any point of the game.

The primary goal of Lazarus is to get to stop the boxes from falling. This is achieved by stacking the boxes in a special pattern to get to the Red Stop Switch. Movement can only occur if the boxes are stacked on an even plain or with a one box difference. Meaning if there are 2 boxes ahead of you, you cannot jump twice the box length.

**Player Controls – Battle Tank:**

Player One (Right Side on the Map)

* Left and Right keyboard button arrows control tank horizontal movement
* Up and Down keyboard button arrows control cannon vertical movement
* Forward Slash (/) keyboard button is to fire the projectile
* Period (.) keyboard button is to activate rocket booster that controls upward tank movement

Player Two (Left Side on the Map)

* A and D keyboard buttons control tank horizontal movement
* W and S keyboard buttons control cannon vertical movement
* F keyboard button is to fire the projectile
* E keyboard button is to activate rocket booster that controls upward tank movement

**Player Controls – Lazarus:**

The only player controls for this game is the left and right keyboard arrow buttons that control the horizontal movement.

# Assumptions

**Battle Tank**

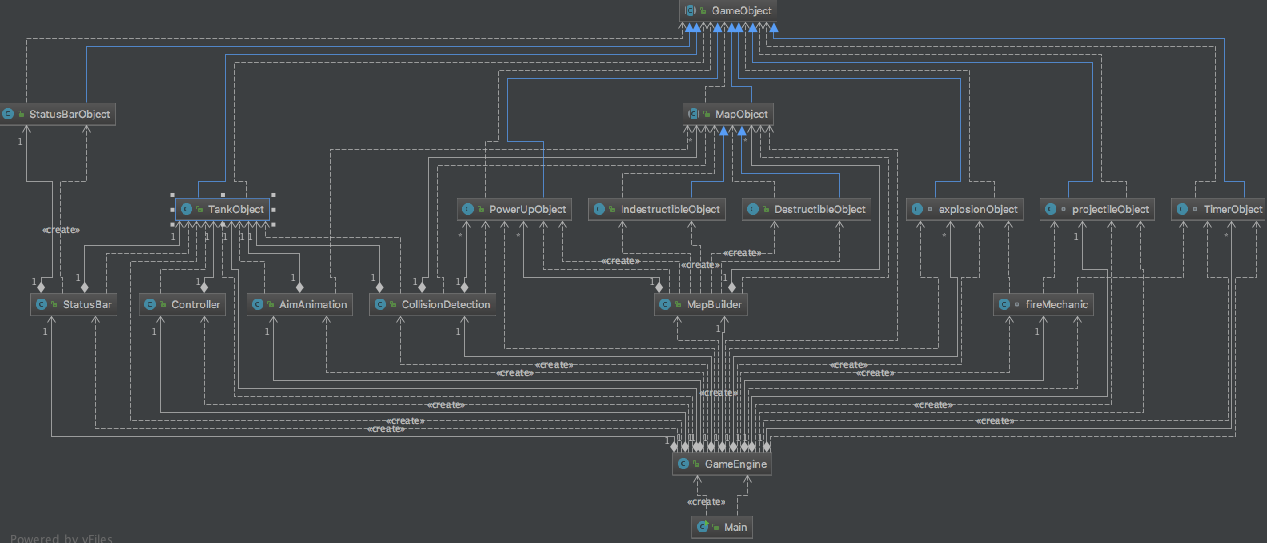
Going in to the game we understood that there needs to be a timer that the game runs on. The way the objects are placed on the screen required a derived class that held a Node; so, GameObject was one of the very first classes we created. There would have to be some kind of physics involved since we were having the projectile fly at an arc and also the gravity. The cannon had to be a separate class and the tank image had to be cannon-less. One of the biggest breakthroughs came after we chose the animation timer as the game loop and figured out how that actually works. Once we figured out that the game works in ticks, meaning that for each time the animation timer is looped one execution happens; the game started coming together really fast and the bugs crawled out. So, we got some repellent for that.

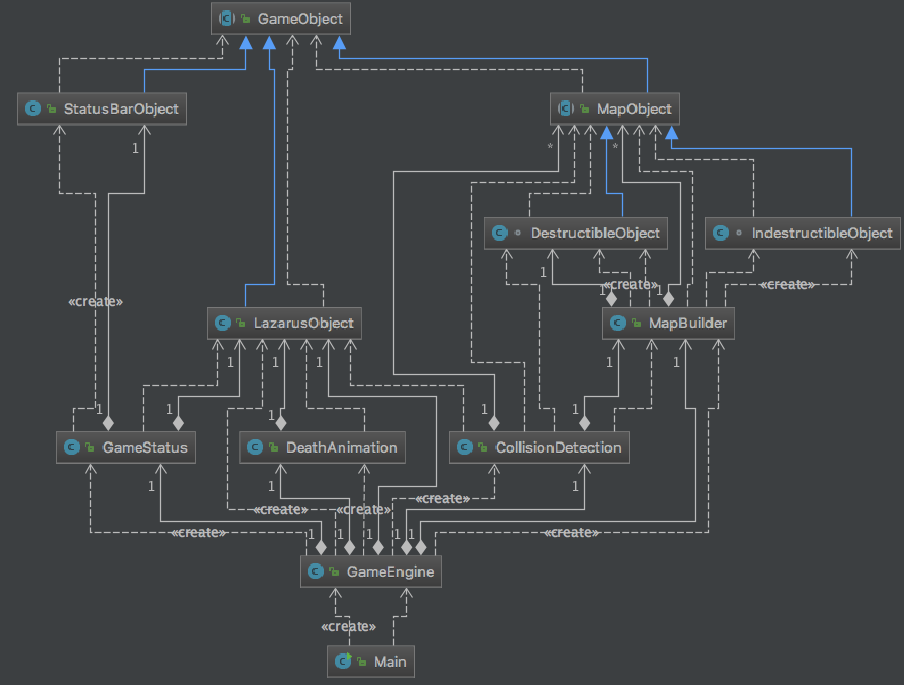
**Lazarus**

After we had struggled with the first game we realized indeed the second game would come with ease as we had a pretty good idea how this JavaFx thing was all about. We knew that almost half of the stuff we used for the first game would have to be deleted so we began with the process of cloning and stripping down Battle Tank. Once our version of the video game template was created we understood that GameEngine, GameObject, Status Bar and collision would still find its way in being useful.

# Battle Tank and

# Lazarus Class Diagrams





# Battle Tank Class Description

**Main:**

The game begins in Main where the display is created along with the canvas. The stage is shown and the GameEngine object is created. Next the gameLoop function of GameEngine class is called.

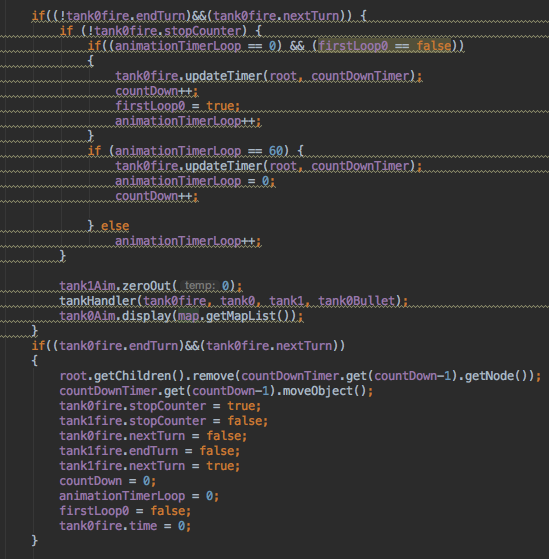
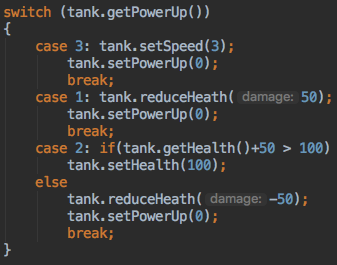
**GameEngine:**

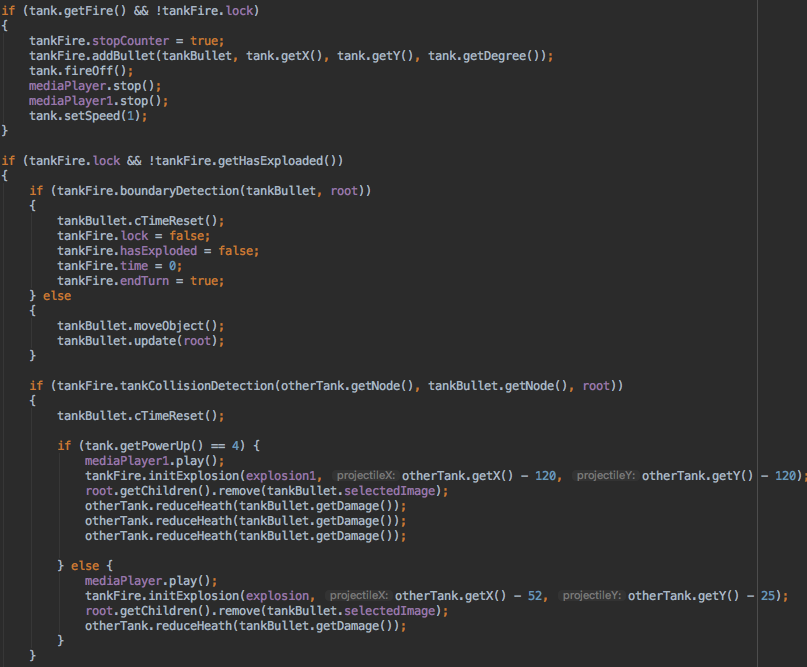
When the GameEngine Object is created in Main; Objects StatusBar, ColisionDetection, MapBuilder, fireMechanic, AimAnimation, projectileObject, TankObject, and Controller are created. Along with some more objects used for sounds, images, image arrays, flags and counters. We group all of our images and root is responsible for displaying them on the screen.

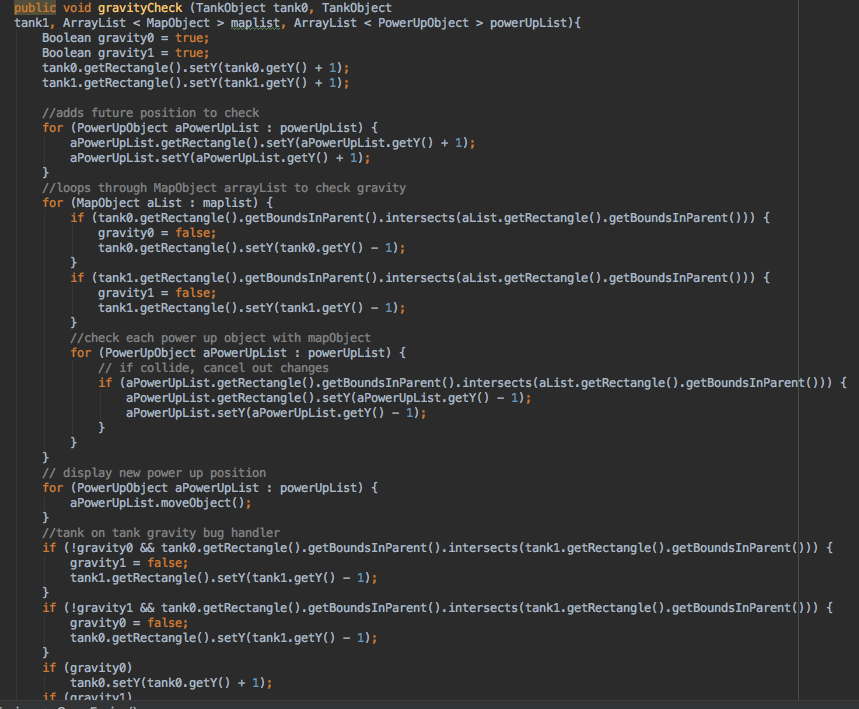
GameEngines initialization is the initialization of all the classes mentioned above. The tank, cannon and flame get placed on the screen. EventHandeler starts to listen for key strokes and the flags for player one’s turn are set.

GameLoop is the beating heart of our game. When Main calls gameLoop, and animation time is used to make changes to the game on each iteration. Right off the back the status bar, gravity check, tanks health, tanks movement, tanks collision, and tanks update functions are always running. These functions always need to be checked for each iteration.

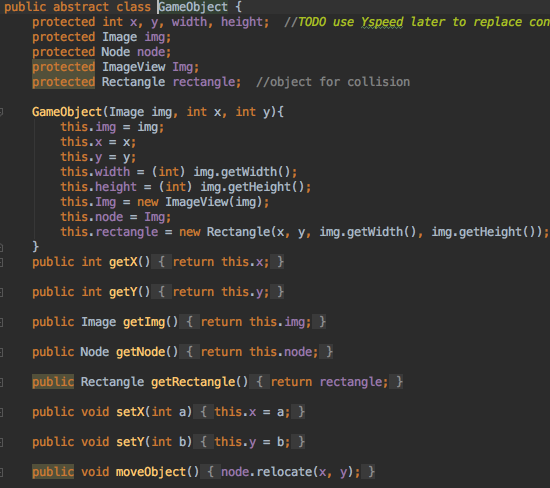
The next series of if statements is the mechanism for the timer and what makes this a turn based game. We use a series of flags to control this whole gig. The display of the timer is also controlled here along with movement locks. Towards the end a series of flags are set so player 1’s if statements don’t get accessed and player 2’s ate able to be accesses and the cycle repeats.

 In the turn base mechanism, tankHandeler is called which is responsible for all the semantics that involve the tank and what the tank does. Right off the back we have a simple switch that checks 3 of the 4 power ups that we have. Tank Speed, increase or decrease of tank health Is set here.

 Next is the different operations that can occur with the tank in our game. Once more, everything is controlled through a series of flags and counters. For instance, what we’re looking for at first is if the fire keys were pressed. That would mean the counter needs to be stopped and the bullet needs to be added to the screen. The media sounds also need to be stopped here because the way media.play() works is when it’s called, it will play until media.stop() is hit. Tank speed is reset just in case speed power up is activated. A lock flag is set off and that’s how an update function for the bullet is monitored along with some more flags that are set off. Once the bullet has been fired, tank and wall collision checks begin monitoring right away. If there is a collision, depending if the bomb power up is set, 2 styles of explosions can be initialized. Here we just need the coordinates, since the images are already pre-loaded. The explosion sound is played and health is taken away from the tank or not if it’s a wall.

The second to last function down the list is the initExplosionObject. Since we are using an array of objects that have a coordinate and an image, the images need to be preloaded to save some computing time. We do this through a series of for loops. Coincidently, we stuck the timer objects in here as well, since it’s all done in the same manner.

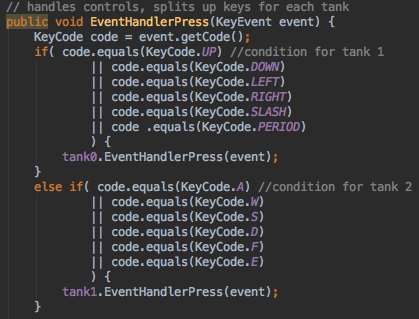
Lastly, we have a pretty neat gravity function that loops through arrays to check if map objects, tanks, power ups, have collided with another, using a rectangle. A gravity flag is set; then, in a later if statement, the Y coordinates are slowly changed.

**GameObject:**

Just like the GameEngine, GameObject is just as equivalent in importance. Most of our classes inherit from GameObject via Super.GameObject holds a few crucial components including getters and setters that fuse our game together. For example, everything in our game uses a Node to add to the Group of screen objects using root.getChildren.add(Node); This is one of the very first breakthroughs that we figured out, and it changed the whole outlook on the game. Not only did it make sense, it also made everything so much easier.

**FireMechanic**:

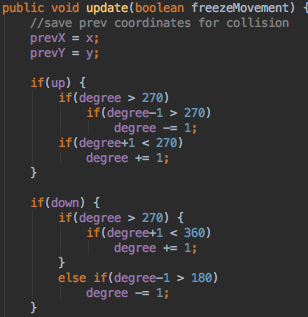
 This class is in charge of every mechanical operation that is related to the bullet. At the top, a few flags are created and set. The bullets are added here. Explosion positions and timer position are initialized. Bullets and boundary detection is here; but instead of rectangles, we use the node itself. Bullet and Tank Collision are also checked here the same way. There is an updateTimer and updateExplosion function that work very similarly side by side. For each timer loop, (tick) one image is displayed and then, the previous is removed. The ticks for the timer are slowed down to 60.

**Controller:**

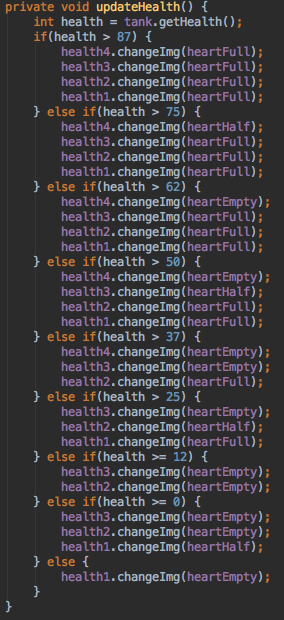
This class uses and event.getCode() to listen to all key strokes pressed and only registers the appropriate keys that are pressed and released. Since the tankObject holds all of the keys, both tanks are passed in.

**TankObject:**

This is a pretty sophisticated class that deals with all operations related to the tank. When the object is created in GameEngine, a series of flags for the movement are created. The node for the cannon is also created, since the cannon is a separate moving part that on another moving part (the tank).

 During initialization of TankObject, most of the variables are preset with default values. The moveTank function moves the tank based on the coordinates and cannon movement, depending on which way the tank is facing. We have a series of getters and setters for the tank and its operations. The way tank collisions work is that it has a previous coordinate. When it hits a collision via wall, tank, or boarder, there is also an event handler that ties in with the controller class that handles the movement.

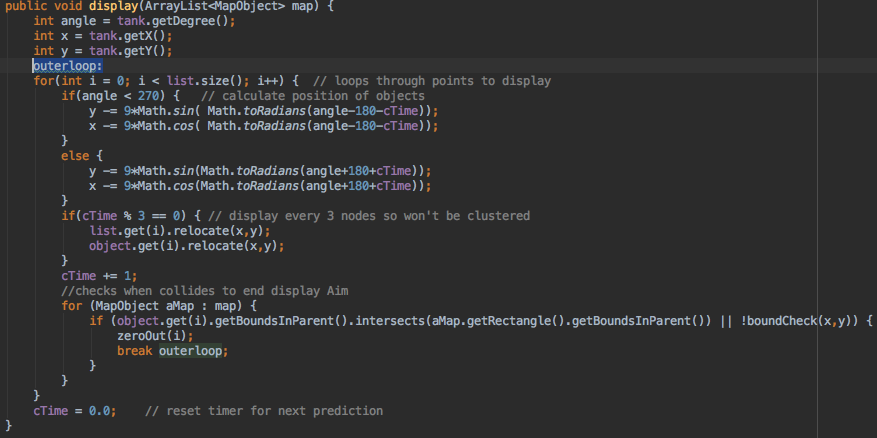
The most important function here is the update. Once the key flags have been recognized by the event handler that the controller class has filtered, it then checks the degree of the tank before changing the movement. The degree of the cannon is what dictates which direction the tank is facing. So, if it’s greater than 270 degrees, that means that the tank is looking to the right. Whereas, less then 270 means that the tank is looking in the left direction.

**StatusBar:**

 The status bar is everything that has to do with the life and health that is displayed on the top of the tank which follows around the tank. When the class is initialized, 4 individual StatusBarObjects (hearts) are initialized with the life and the tank object. Tanks coordinates are used to place the individual objects where the tank is. The display function is what updates the position of the heart. The amount of life is determined by the getLies() getter. 100 is the maximum amount of health that you can have, and the display of the hearts are changed with the updateHealth Function. updateStatusBarLocation function works in a similar way that changes the location of the x and y coordinates of the numbers that are displayed. We have 3 images for the hearts. An empty heart, half a heart and the full heart.

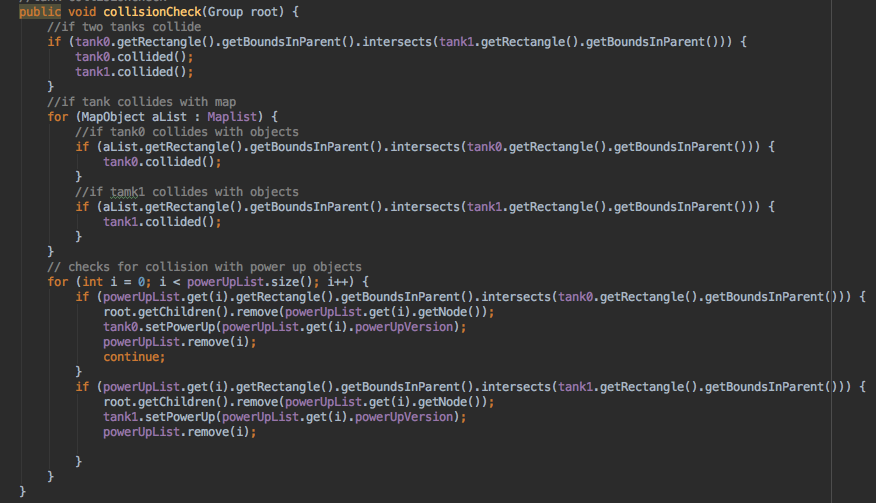
**projectileObject:**

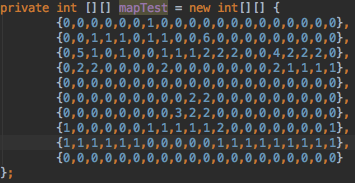
This class is mostly in charge of change of position of the bullet when fired. When the object is created in the GameEngine, an image with the coordinates are inherited form GameObject and some flags are set along with the proper variables. There is a cTimeReset function that sets the cTime to zero. cTime is a counter that is used to change the angle of the projectile.

****The rotation of the projectile is changed on every iteration by removing it from the screen, changing the angle, then adding it back to the screen. Based on the angle of the cannon, it determines which way the projectile is going to fly. We use the sin and cos curve to follow the parabolic motion that is multiplied by a constant of 9 to give it a good speed. The angle starts with the cannon angle and is changed by a factor of 1 for every tick of the game.

**AimAnimation:**

This class is what shows the trajectory of the projectile. When the object is created, we are going to need a tank object along with an image and 2 arrays, along with the cTime. The init function is in charge of looping through a constant value of 125 and adding the image of a white faded X that imitates a look of a trajectory. The display function borrows the physics from the projectileObjects update function and uses an assembly style jump statement via break to imitate a special kind of outer forloop

**CollisionDetection:**

This class is in charge of all of the collisions that occur between objects and objects, tanks and tanks, objects and tanks. Power ups, destructible, non-destructible objects are all also included. In the function CollisionCheck, we use a nested forloop to runs through the arrays and checks for collision using an intersect and a rectangle. The last function, WallsCollisionDetection, is what removes the objects if they are destructible when collided.

**MapBuilder:**

 This is a pretty cool class because it uses matrices to draw all of the objects to the screen when creating the map. When the class in created in the GameEngine, 2 vital arrays that hold the maplist and power-up list is stored. Then we have a nested forloop that maps the objects on the map based on the number that is assigned to them. Destructible Objects are = 1. Non-destructible objects are = 2. 3,4,5 and 6 are the power ups. There are 4 power ups located on the map. Red Shield Box: increases tanks health by 2 hearts. The gray box with a wrench increases the speed of the tank for the duration of their turn, by 3 multiples. The red barrel decreases the health of a tank by 2 hearts. The wooden box activates a feature where the tank is able to shoot one special projectile that makes 3 multiples the damage of the normal projectile.

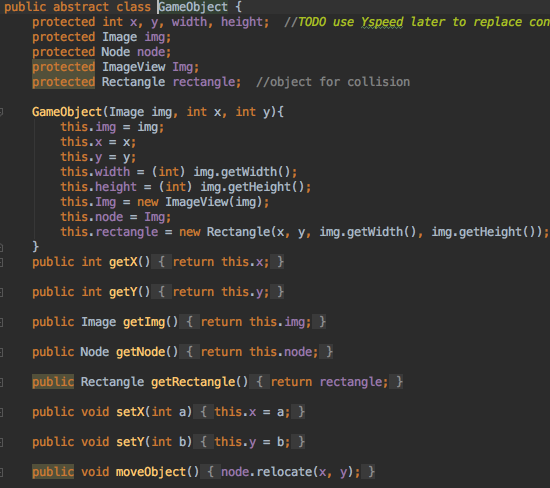
**TimerObject, StatusBarObject, PowerUpObject, MapObject,IndestructibleObject, explosionObject, DestructableObject:**

These seven classes are just a variation of X, Y coordinate and an Image. IndestructibleObject and DestructibleObject classes also have some flags as well. These classes are just basic classes that the rest of the game uses to display the images.

# Lazarus Class Description

**Main:**

The game begins in Main, where the display is created along with the canvas. The stage is shown and the GameEngine object is created. Next gameLoop of GameEngine is called.

**GameObject:**

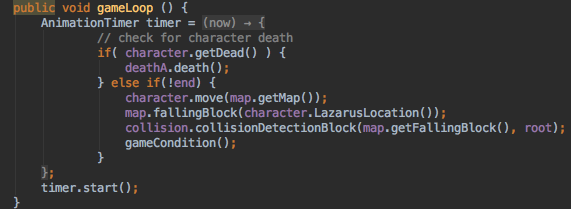
Just like the GameEngine, GameObject is just as equivalent in importance. Most of our classes inherit from GameObject via Super.GameObject holds a few crucial components including getters and setters that fuse our game together. For example, everything in our game uses a Node to add to the Group of screen objects using root.getChildren.add(Node); This is one of the very first breakthroughs that we figured out, and it changed the whole outlook on the game. Not only did it make sense, it also made everything so much easier.

**StatusBarObject, MapObject, IndestructibleObject, DestructableObject:**

These four classes are just a variation of the X, Y coordinates and an image. IndestructibleObject and DestructibleObject classes also have some flags as well, and the MapObject has a weight factor. These classes are just basic classes that the rest of the game uses to display the images.

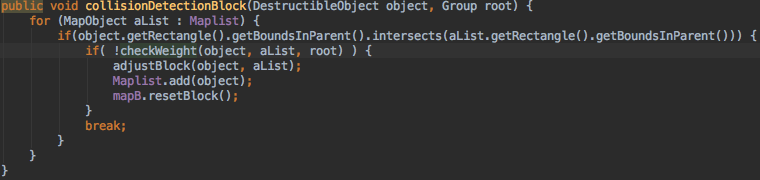
**GameEngine:**

This class creates Object necessary for the game to work such as CollisionDetection, MapBuilder, LazarusObjects, GameStatus, and DeathAnimation. When the class is initialized, the character is created and added to the screen. Similar event handler operations are borrowed from the first game. The map is populated, and the death animation is initialized.

 The restartLevel function uses the map’s reset features to clear all of the objects and reset them to the original position. This is used when the character dies and the level is restarted. There is a win() and lose() function that just draw images to the screen and set an end game flag off that freezes all operations.

The game condition class checks to see if there has been a collision with the falling object or the stop box. The game loop is what runs this game; and where the same animation timer is used to change everything on the game.

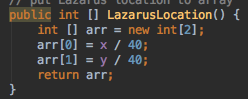
**CollisionDetection:**

This class is in charge of all of the collisions that occur with the character. When the class is created in the GameEngine, the map array is created and is imported from the mapBuilder. For the collisionDetection Block, a forloop is used to check the collision between the blocks. Similarly, the collisions between the character and the blocks are handled by the CollisionLazarusCheck function. There is a check weight function that recognizes the different boxes with another. There is also a check wining condition that sees if the character has touched the stop box along with an adjust Block that makes sure the boxes fall in alignment.

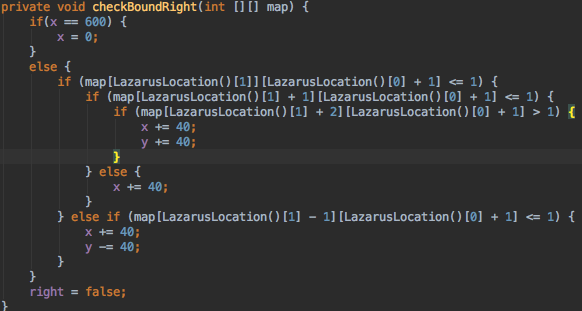
**DeathAnimation:**

This is an interesting class because it borrows the same mechanics of the explosion update functions of the first game and is placed in a separate class. The images are displayed on the map with a use of a counter; and at the end, the game is restarted and the character is set to live again.

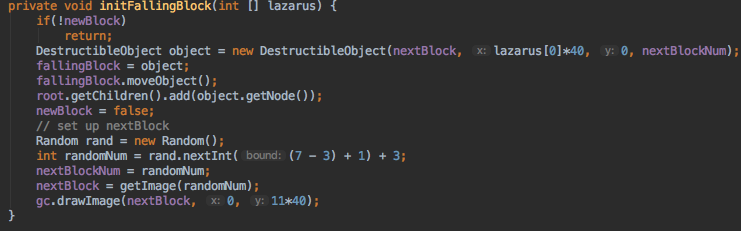
**GameStatus:**

This class also borrows the same mechanics from the first game. Instead of 4 hearts, there is only one. The only thing that changes is the life count that starts from 3. **LazarusObject:**

This class borrows a lot of the first game TankObject functions, such as the getters and setters. The key event handler is also used in the same way. We have a move function that changes the position of the character. We also have a function that updates the position of the character onto the array.

CheckBoundRight and checkBoundLeft are the main functions of this class. There are a series of if statements that check the positon of the character in comparison to the height of the boxes in the direction the key strokes that are directed to. For example, you not allowed to jump 2 boxes, so this is not permitted here. One up One Down and One straight is the only movement that is allowed. The character is also allowed to move from one side of the screen to another. Is it determined that a movement of 40 pixels is considered one box.

**MapBuilder:**

The map builder creates the map as it did in the first game via matrices.

We have a case statement that has the metalBox = 3, stoneBox = 4, rockBox = 5, woodBox = 6, cardBox = 7. The map is drawn using a function called create which traverses through the array and sets each numberedobject to the correct image. The falling of the boxes is also initialized here. The position of the character is used for the randomly generate the type of box that start to fall. The destructible boxes are claimed to be everything that fall from the sky. When the map is reset, all of those destructible objects are removed. The display function works in a reverse way, which is pretty much adding everything to the map, one tick at a time.

# Classes Shared

Between the two games, the classes that were mostly reused were CollisionDetection, DestructableObject, IndestructableObject, GameEngine, GameObject, Main, MapBuilder, MapObject and StatudBarObject. Of course, lots of modifications were necessary in order to make the game work.

# Team Reflection

From what we can reflect on the project is that a lot of learning and corporation came to be. Working with another person coding style, understanding and synchronizing with their code became the biggest challenge of creating these games. Using git was also troublesome at first primarily because of the .idea files that IntelliJ that we were sometimes pushing. Working on the code together proved that where are more than capable for the “real life” scenario of a co-op working environment in the professional world. After understanding what it takes to create a JavaFx video game now, the possibilities seem endless.

# Summary of Work

For the tank game, Peter did the GameObject, MapObject, StatusBarObject, StatusBar, AimAnimation, Controller, DestructibleObject, TankObject, PowerUpObject, IndestructibleObject, and CollisionDetection.

For the tank game, Albert did the TimerObject, projectileObject, GameEngine, fireMechanic, explosionObject, and CollisionDetection, Main.

For the second game, Peter did the CollisionDetection, DestructibleObject, GameObject, GameStatus, IndestructibleObject, MapBuilder, and LazarusObject.

For the second game, Albert did the DeathAnimation, GameEngine, Main, StatusBarObject.

**Even though we can say some classes we understood because we worked on them for a longer period of time, in the end result we had to work with each other’s files and do minor changes and tweaking so to say someone worked with one particular file is obsolete.**

# Project Conclusion

This team Project was very different from the projects that we had done before in other classes because there were a lot of dynamics that were not introduced in other prior classes. Working with a partner added a new experience to a project because in the beginning we had different ideas on what the outcome would look like. We butted heads in the beginning since we had different thoughts on how this game should turn out, however after we figured things out and got on the same page, it became a lot easier to work with. Communicating was something that we had trouble with as well since w our work habits were different. We believe that this was the hardest thing about the project, however when we worked past our differences, it became really nice to work with our partner. Once we got into the motion and on the same wave length, everything came up a lot easier and actually pretty enjoyable. As for the code within the program itself, there were some difficulties in writing some of the logic because we had not work in javaFX before this. After figuring out how to work with javaFX, things fit into place as the logic for the game were already set up to make things easier for us to build the game on top of. This project was a really good learning experience as it showed us how to work in a group and how to communicate to get a working product out with a certain deadline that we had to hit. Both games turned out great as the came out as it was intended to run without any bugs that we had ran into.