Richard Robinson
Lorraine Goveas

Assignment 4: Tank Game, Rainbow Reef

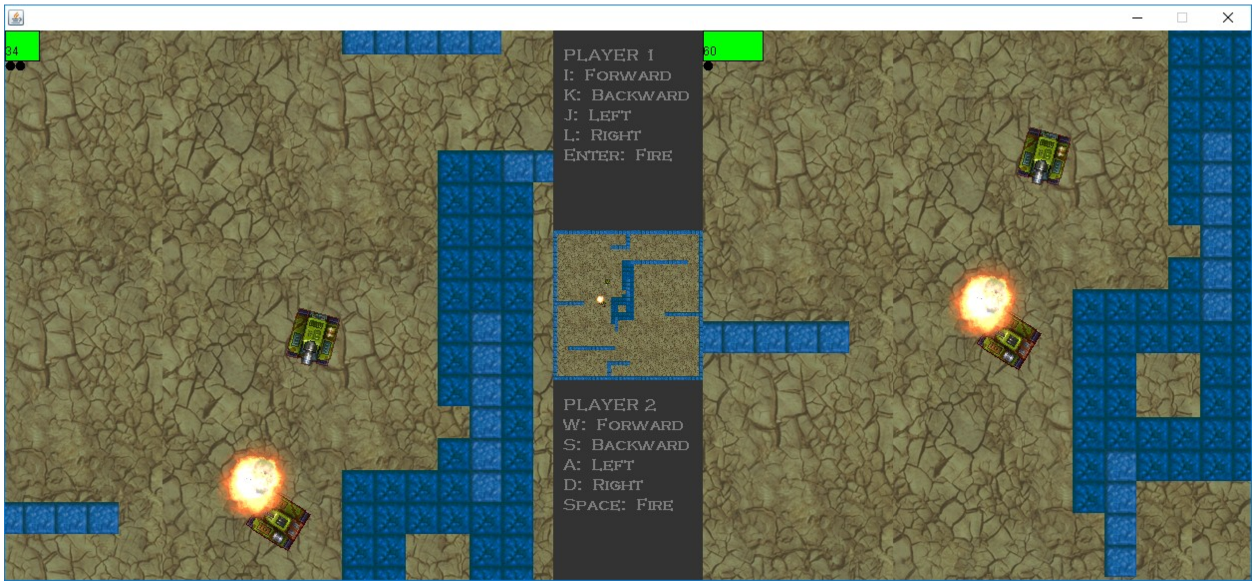
https://github.com/SFSU-CSC-413/assignment-4-richard-and-lorraine

I asked JRob about pushing resources to aid command line running, don't take off 25%

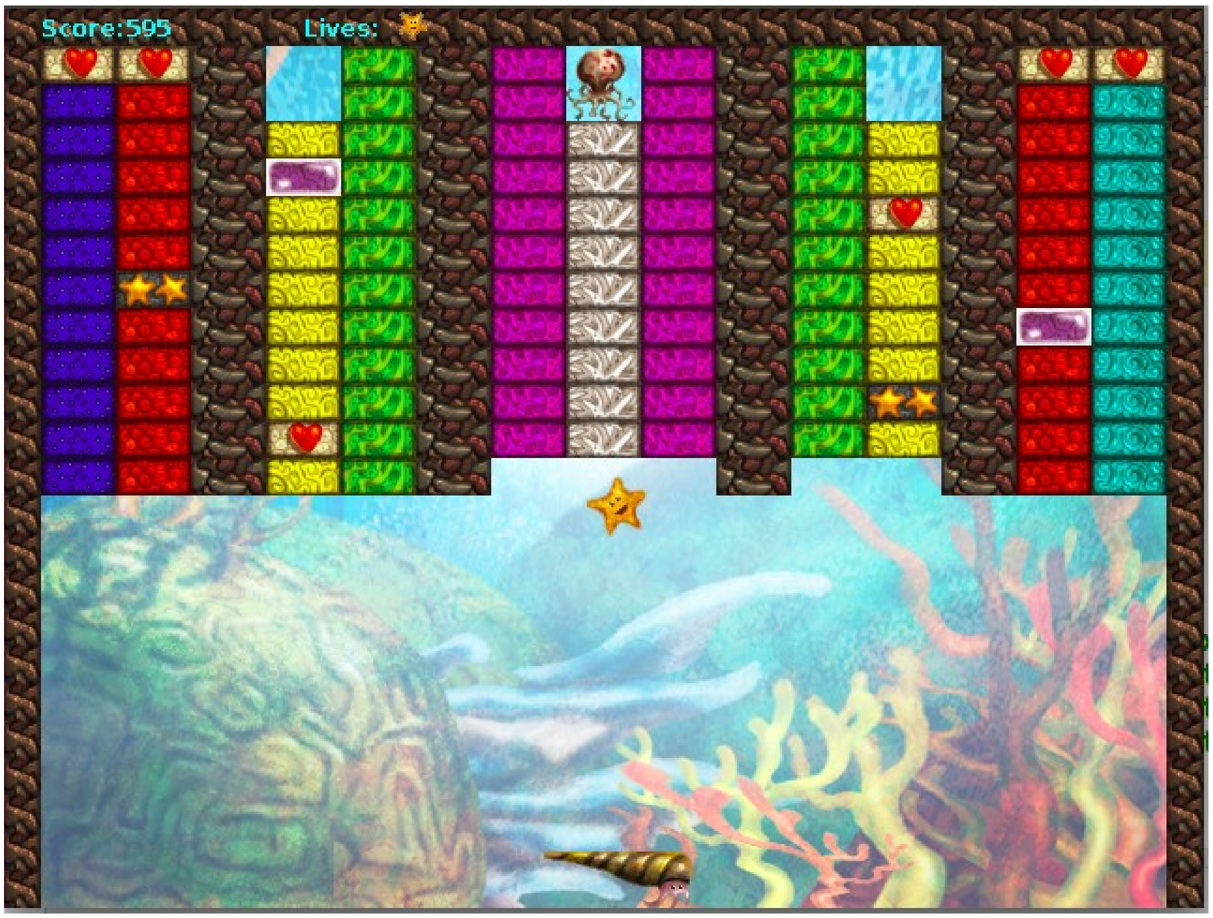
Introduction:

A tank game and a block-breaking game, Rainbow Reef, were created for this assignment. The goal of this project was to approach the development in an object-oriented fashion. Because of this it was possible to reuse many of the classes from the tank game for the second game.

The tank game is a split-screen top-down action game where the goal is to subjugate the opposition with explosive shells! Each player gets two lives and in the current implementation each has their own distinctive tank (There are six tanks total, two colors and light, medium, and heavy variants. Tank selection was not fully implemented for the deadline but the tanks are creatable.) with one player controlling the light tank and the other the heavy. There are breakable and unbreakable walls, an ending screen, sound effects, and the best open source soundtrack.



In rainbow reef the octopus overlords look down upon your mighty star, but with the power of Katch (a shell) and Pop (the star, like popstar) the barriers and fortifications stand no chance. This lovely, adorable, little star breaks through the protective blocks and defeats the octopus, presumably with friendship. There are four action-packed levels, a catchy song, and sounds for this game.



At this point we would like to highlight some things that may make this project distinct from the other projects. For the tank game we used the first sprite frame for projectiles and tanks. We also used affine transforms alongside some simple math for direction, rotation, and general positioning. Our Maps file can be passed a map number and generate a relevant map from it's list. To add a map one can simply put the resource path of a new map.. These games have sound, music, possess no game-breaking bugs that we know of, and are fully playable. We have six different tanks, and three different projectiles; given a few more days we would also have had the menu completed and allowed for easy tank selection as well as two or more maps with different background themes. The Rainbow Reef game has multiple levels and special effect blocks. It is possible to win both of these games, but the program needs to be reset to run again. Our sprite class can parse through sprites with more than one row.

We would also like to mention that there are two typos in the code, we apologize for that. It was not until documentation that this was realized but alas we cannot change code after the submission date. We hope you understand.

Execution and Development Environment:

We used intellij to develop our code with JDK version 1.8. To execute we ran the code.

Our code will run following the below steps. We could submit a version that will run without copying the src folder but it involves making a push after the due date, it is not worth the risk of a 0 for some convenience. The IDE used accessed the resources slightly differently, this is why the map texts and visual and audio resources need to be copied over.

TANK GAME

TO JAR:

1) Change to "src" directory
2) Compile the java files with javac Running/RunTankGame.java
3) Create a text document or mf file in the "src" folder, name it manifest.

This manifest will have this line in it:

    Main-Class: Running.RunTankGame

4) Type "jar cvfm AwesomeTankGame.jar manifest.txt ."
5) Run with java -jar AwesomeTankGame.jar

TO RUN WITH JAVA COMMAND:

After step 3 type java Running/RunTankGame

Rainbow Reef

TO JAR:

1) Change to "src" directory
2) Compile the java files with javac Running/RunRainbowReef.java
3) Create a text document or mf file in the "src" folder, name it manifest.

This manifest will have this line in it:

    Main-Class: Running.RunRainbowReef

4) Type "jar cvfm RainbowReef.jar manifest.txt ."
5) Run with java -jar RainbowReef.jar

TO RUN WITH JAVA COMMAND:

After step 3 type java Running/RunRainbowReef

Scope of Work:

Tank game:

Game Objects:
Distinct game objects were created for tanks, walls, projectiles, and backgrounds.
-Some walls are destructible
-Each distinct class splits off into child classes when relevant : six tanks, three projectiles, and breakable and unbreakable walls

Game Clock:
-responsible for update intervals, is able to be run as a thread and observed

Maps class:
-can add new maps through a map number and by adding the resource path to an array

Keylistener :
- Uses a boolean array, relevant index is set to true on press and false on release

Controllers:
-A tank controller that assists tank movement and handles collisions
-A projectile controller that handles projectile movement, collisions, damage, and explosion animations

The Resources package contains all of the images and audio used for the game.

GameWorld is responsible for holding the current game state and displaying something relevant to the players.

Rainbow Reef:

We were able to reuse much from the tank game for this game, making it significantly easier. These components stayed relatively the same: Maps, controllers, keyboard input, sprites, and walls.

Game objects had to be changed a bit, but the majority of the work on it was already completed.

The objects in this game are:
-Blocks
-Octopus
-Star
-Shell

A notable difference here is that the octopus, shell and star are all animated sprites while the tank game only had explosions as animated sprites.

There are special effect blocks such as the double star block which makes another lovely star.
Controllers:
Shell Controller
-Responsible for shell movement

Star Controller
-Responsible for star movement and collisions

Rainbow Reef game world holds the game state such as player score, lives, and the game objects.

Multiple levels were implemented in this game and the player wins upon completing all levels or loses upon losing all lives.

Music was incorporated through the use of a thread and sound effects are present. This was interesting as not only did Rainbow Reef use classes originally intended for the tank game, but the tank game took the music class from Rainbow Reef.

Assumptions:

The player will have a great time. The player understands how to double click a JAR file or run the code through another means. It is also assumed that the player owns a computer with Java installed. It is also assumed that the player presses the appropriate keys to move the tanks and shell.

Implementation Decisions and Code Organization:

We decided to attempt to organize classes in a meaningful and legible way. Each class was made so that they would be responsible for their part and nothing more. This allows easy access to particular components, for example if one wants to look at a specific tank they can easily find it, if someone wanted to find the resources for images it is likewise easy to identify. This approach limits cluttering and makes the code easier to work with and to read.

The GameClock class is responsible for creating updates at regular intervals. This is implemented as an observable and can be run on a thread. Other classes may observe this class in order to synchronize and update at the defined interval. It was absolutely necessary that the clock was observable to be able to link fluidly with other classes.

The GameFrame within the Running folder initializes a frame and adds the GameWorld Panel to it.

The GameWorldPanel is responsible for holding the current game state and displaying it in a meaningful way to the player. This holds the game objects and pieces together the components to make a meaningful whole. This also displays all the visual elements; the player screens, minimap, controls, and game end screen.

The RunTankGame class does exactly what it says and runs the game.

Game objects in an abstract class which holds components that all game objects should possess such as x and y coordinates, resource paths, images, degrees(for which way the image faces), and the potential for hitboxes. There are different subclasses which extend game objects such as tanks, projectiles, and walls. The wonderful thing about setting up components this way is that new items can be created easily or old things can be modified with little or no effect on other classes.

The abstract Tank class contains health, velocity, safe zones (for holding last non-colliding position), a firing delay, and projectile type. It also contains methods for movement, firing, and repainting. All tanks are capable of doing these things so they are included here. The movement uses sine and cosine to move the tank forwards and backwards and uses affine transforms for rotation. The firing delay controls the rate of fire for the tanks and it checks the system time to see if it is ready to fire. Safe zones are used to handle collisions. From here the tank class is extended into specific tanks each with their own health, resource path, projectile type, firing delay, and velocity.

The abstract Wall class contains the total number of walls, a wall ID, health, and means for taking damage. The wall ID was used because we implemented a hash table for storage and access to walls. When each wall is created it is assigned an unique ID and the number of walls is incremented. The walls are capable of taking damage and have a health just in case a wall should take a certain amount of damage before being destroyed. This class extends into indestructible and destructible walls. The difference between these two is the image path and that the destructible walls override the takeDamage method so that they can take damage.

Projectiles contains the abstract Projectile class. This contains a name identifier, damage amount, velocity, and a source (what tank fired the shell). Projectiles have the ability to move forward only and are capable of painting in a certain direction due to the use of affine transforms. This class branches into heavy, medium, and light projectiles. The projectiles differ in damage, velocity, name, and image path. The reason for having a source on these projectiles is to prevent friendly fire collisions from occurring when firing.

The last game object is Background which simply has an image path and sets its image.

The Graphics package contains the Animations and Sprite Classes alongside the Explosions class. The Sprite class is capable of processing multi-rowed sprites by being passed an image path and the number of X frames and the number of Y frames the sprite contains. This will generate an array of buffered images that can be navigated in order to grab a specific sub-image. Animation initializes a sprite and walks through the frames sequentially and allows looping. The Explosion class creates a sprite and an animation in order to create an explosion effect. This class branches into the different explosion types, LargeExplosion, MediumExplosion, SmallExplosion, and TankExplosion. These classes can pass their constructors X and Y coordinates and generate an explosion at that location. The explosions were set up this way to make the generation of explosions easy. Tthe wanted explosion type simply has to be called and the explosion occurs.

The Maps class reads the text file and extracts meaningful information from it. It builds a hash map of walls for later use, finds starting tank positions, and holds resource paths for the available maps. These maps can be read by passing through a map number which correlates to a text document which contain the maps. The Maps class was made this way so that new maps could be added very easily and so that it could be reused for other programs.

The classes within Movement are related to movement. KeyboardInput is an observable keylistener containing a boolean array which represents keystrokes. The boolean values are set to true when a corresponding key is pressed and false when released. This allows for other classes to check what keys are currently being pressed and also allows for multiple inputs at once. ProjectileController is responsible for handling the creation and deletion of projectiles, projectile collisions, damage, and initializes explosion effects for the projectiles. TankController uses the boolean array from the KeyboardInput in order to move the tanks and to make them fire. The movement behaviors are defined within the tank classes themselves, but the TankController is what makes movement possible.

The Music class is runnable and contains a resource path to the songs of interest. It is meant to be established as a thread so that it may constantly play music. It sleeps for the song duration then plays it again in order to assure that there is no awkward time without music.

Each class is well-defined, easy to use, and expandable. The classes were not done perfectly and we can identify areas for improvement, but they are organized in a meaningful and accessible fashion.

The game world for super rainbow reef contains the game state and displays it to the player. It initializes the current map by passing a map number to the Maps class. A hash map of the objects is created through parsing the text files.

The game objects for this game include a star, octopus, shell, and blocks. The star, octopus, and shell are all sprites. These game objects all have a health, score values, object IDs, hitboxes, degrees, resource paths, images, and a static count of the total number of objects.

The star is the main movable object for this game. It has separate functions for moving horizontally and vertically in order to better handle direction changes from collisions. It contains a boolean to check if it is fired, a method to fire, and methods to alter X and Y movement directions (when it collides with a block it should bounce off of the block in an appropriate direction).

The shell class has methods to move left and right.

The next level is reached when the star hits the octopus.

There are many types of blocks each containing a different score value and some of them have special effects. The special effect blocks include: split blocks which makes another star, life blocks which adds an extra life, and double blocks which can take two hits before dying. These effects are handled through an effect value present within the blocks. Wall blocks and solid blocks are not destructible. If the star collides with a life block then the player gets a new new life. If it collides with a split block a new star object is created. If it collides with the octopus then the map number is increased and the next text file is loaded.

The Movement package includes the ShellMovement and StarMovement classes. The ShellMovement class is responsible for moving the shell left and right and firing.   The StarMovement class is responsible for star movement and collisions. This class also changes the direction of the star, handles the blocks damage and removal of blocks, and adds sound effects caused by collisions.

The Maps, GameClock, Music, KeyboardInput, and Sprites classes are pretty much identical to those of the tank game and were reused. The majority of other content were simply altered versions of tank game classes.

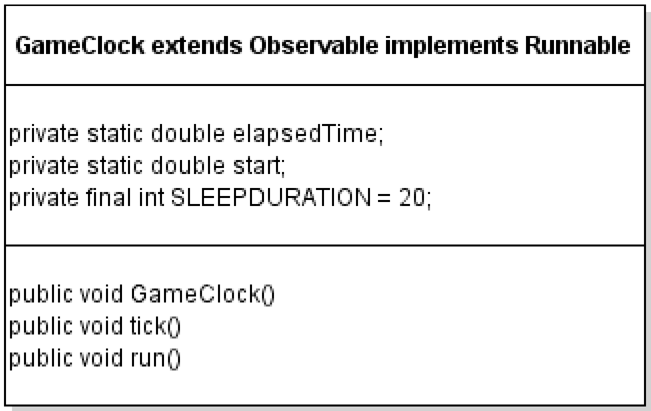
Results and Conclusions:

This project was excruciatingly frustrating, but also massively rewarding. We learned a lot from this project and are immensely glad that we were assigned it. The time-frame present to work on this project was rather short for the content desired and the outcome would have been cleaner and more refined had more time been devoted to this project. The most interesting part of this project was how easy it was to make a second game by reusing the resources from the first game.

Class Diagrams:

TankGame:

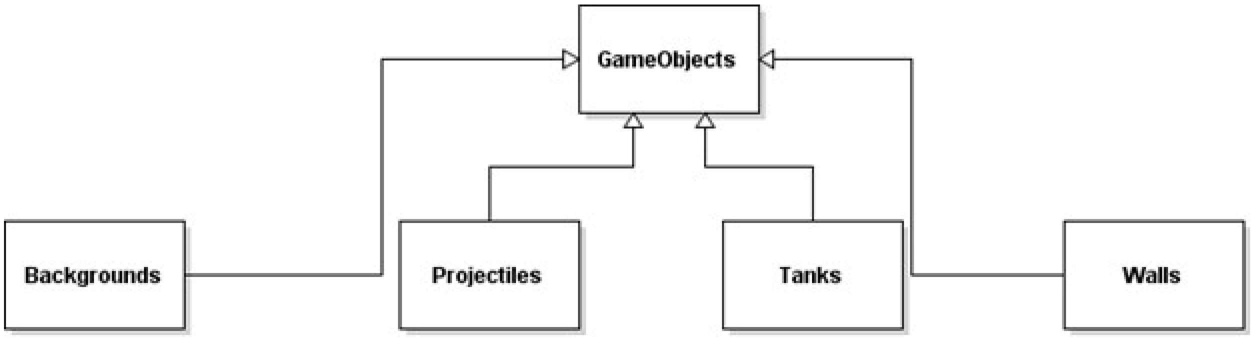
Clock:

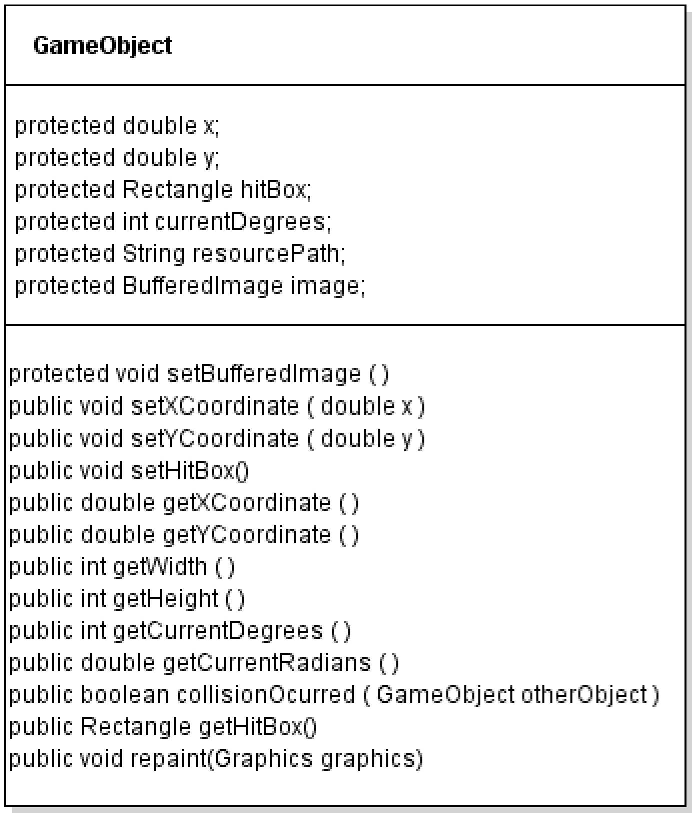


```
GameClock extends Observable implements Runnable

private static double elapsedTime;
private static double start;
private final int SLEEPDURATION = 20;

public void GameClock()
public void tick()
public void run()
```

The Game Clock class is an observable that notifies its observers every time the clock ticks.

Game Objects:

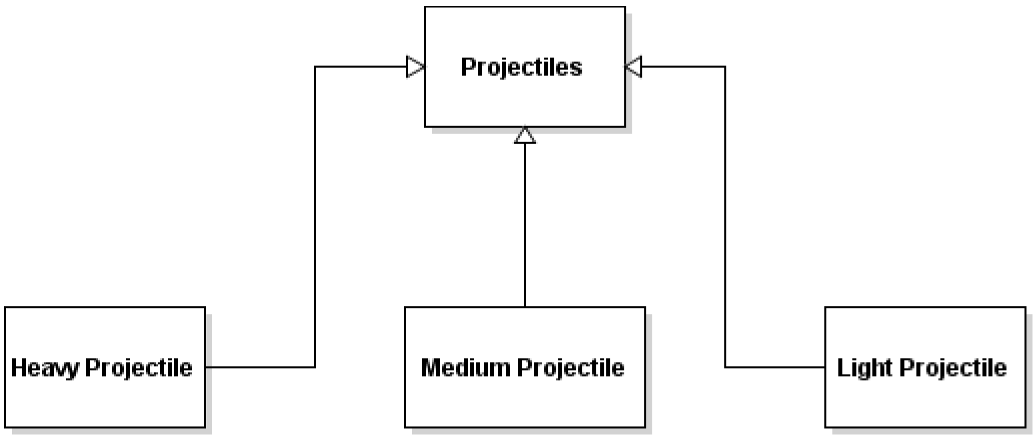




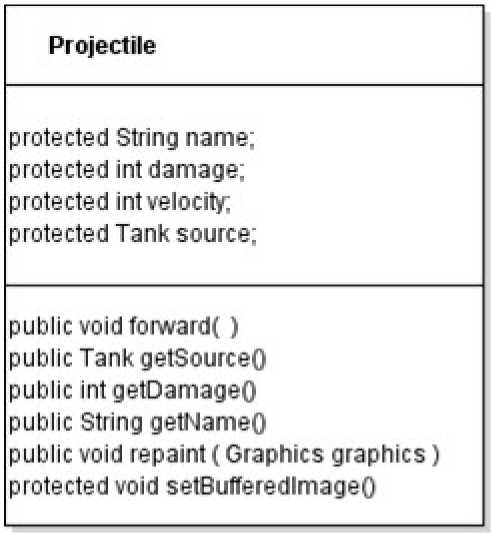
```
GameObject

protected double x;
protected double y;
protected Rectangle hitBox;
protected int currentDegrees;
protected String resourcePath;
protected BufferedImage image;

protected void setBufferedImage ( )
public void setXCoordinate ( double x )
public void setYCoordinate ( double y )
public void setHitBox()
public double getXCoordinate ( )
public double getYCoordinate ( )
public int getWidth ( )
public int getHeight ( )
public int getCurrentDegrees ( )
public double getCurrentRadians ( )
public boolean collisionOcurred ( GameObject otherObject )
public Rectangle getHitBox()
public void repaint(Graphics graphics)
```

Game Object:
There are four categories of game objects walls, backgrounds, tanks, and projectiles that extend the GameObjectS class. Each game object has an x and y coordinate, a hitbox, an image, and a number of degrees to indicate which direction the image is facing. Collisions can be checked for any game object using the collisionOcurred method.

Projectiles:



Projectile:



Projectile

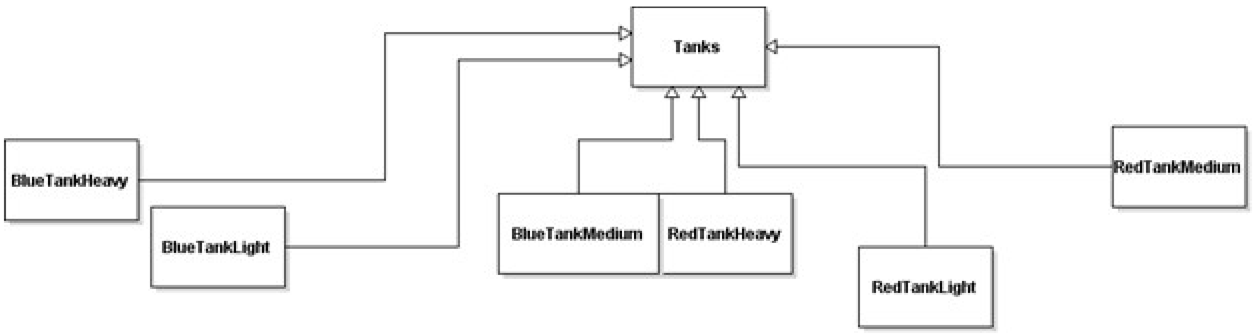
```
protected String name;
protected int damage;
protected int velocity;
protected Tank source;
```

```
public void forward( )
public Tank getSource()
public int getDamage()
public String getName()
public void repaint ( Graphics graphics )
protected void setBufferedImage()
```

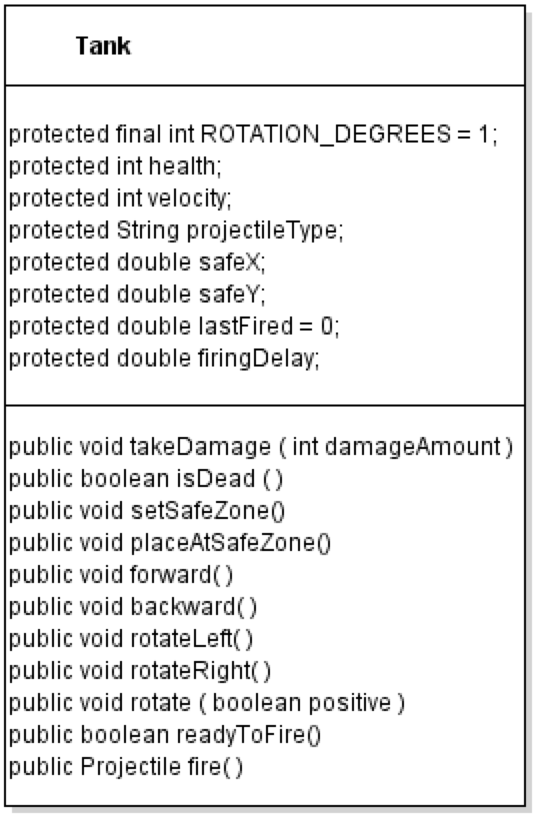
There are three different types of projectiles that extend the projectile class. Each projectile has a name, amount of damage that it can inflict on another object, and a velocity.
forward(): moves the projectile forward based on velocity and current angle
getSource(): indicates the tank that owns the projectile to prevent friendly fire damage
getDamage(): returns damage amount for projectile
repaint(graphics): draws the projectile and adjusts the projectile image direction to the correct angle using affine transformations
setBufferedImage(): takes the image from the directory and sets the image to the correct size

Each extending tank class has a constructor to initialize the protected values and another constructor that takes a tank object to set an owner and position.

Tanks:



Tank:

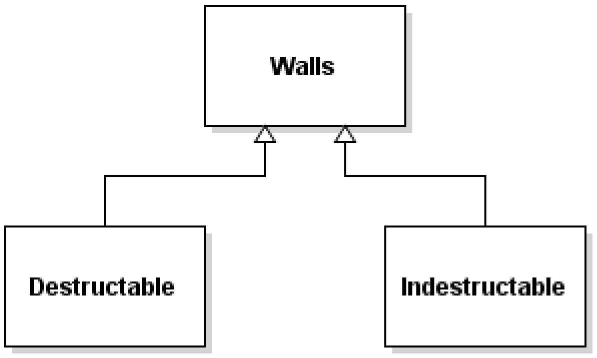


```
Tank

protected final int ROTATION_DEGREES = 1;
protected int health;
protected int velocity;
protected String projectileType;
protected double safeX;
protected double safeY;
protected double lastFired = 0;
protected double firingDelay;

public void takeDamage ( int damageAmount )
public boolean isDead ( )
public void setSafeZone()
public void placeAtSafeZone()
public void forward( )
public void backward( )
public void rotateLeft( )
public void rotateRight( )
public void rotate ( boolean positive )
public boolean readyToFire()
public Projectile fire( )
```

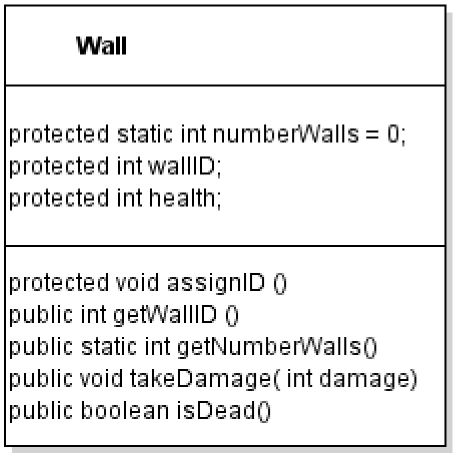
The tank class contains components necessary for tank objects. There are six different tank objects that extend the tank class. Each tank has a health, velocity, projectile type, firing delay, and the time at which it last fired. The class also stores the last non-colliding X and Y coordinates and has methods directing how the tank behaves.

takeDamage(int): decrements the health of the tank by the damage amount
isDead(): returns true if the tank has 0 or less health
setSafeZone(): sets the non-colliding X and Y coordinate
getSafeZone(): retrieves non-colliding X and Y coordinate and sets tank X and Y to these
forward(): moves tank forward using velocity and current angle
backward(): moves tank backward using velocity and current angle
rotate(Boolean): rotates the tank by ROTATION_DEGREES, passed true by rotateRight() and false by rotateLeft()
readyToFire(): returns true if the firingDelay time has passed
fire(): returns the projectile type corresponding to the tank's projectile type
setBufferedImage(): sets the image to the correct size
repaint(graphics): draws the tank facing the correct direction through the use of affine transformations
The tanks extending the Tank class contain constructors that can place the tanks at a particular X and Y coordinate and a constructor to initialize the protected components.

Walls:



Wall:

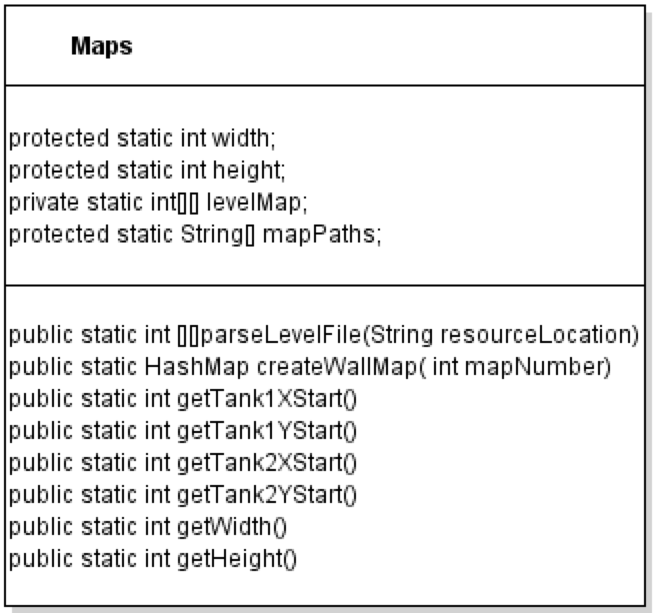


```
protected static int numberWalls = 0;
protected int wallID;
protected int health;

protected void assignID ()
public int getWallID ()
public static int getNumberWalls()
public void takeDamage( int damage)
public boolean isDead()
```

The Wall class contains a component to track the number of walls. Each wall has a wall ID number and health. Walls are given the ability to take damage and there is a method to check if the walls are dead.
assignID(): assigns the wall an integer ID number equal to the number of walls then increments this number by one. This is called in the wall constructors

getWallID(): returns the wall ID
getNumberWalls(): returns number of walls
takeDamage(int): by default this does nothing, the children of this class will define this if needed. This is here because projectiles check walls for damage, so all walls need a method named takeDamage.
isDead(): returns true if health is equal to or less than zero; used for later removal of walls
The destructible walls take damage method allows the wall to take damage. The indestructible wall's does not. They have constructors to place the wall at a particular X and Y coordinate and to initialize the protected values.

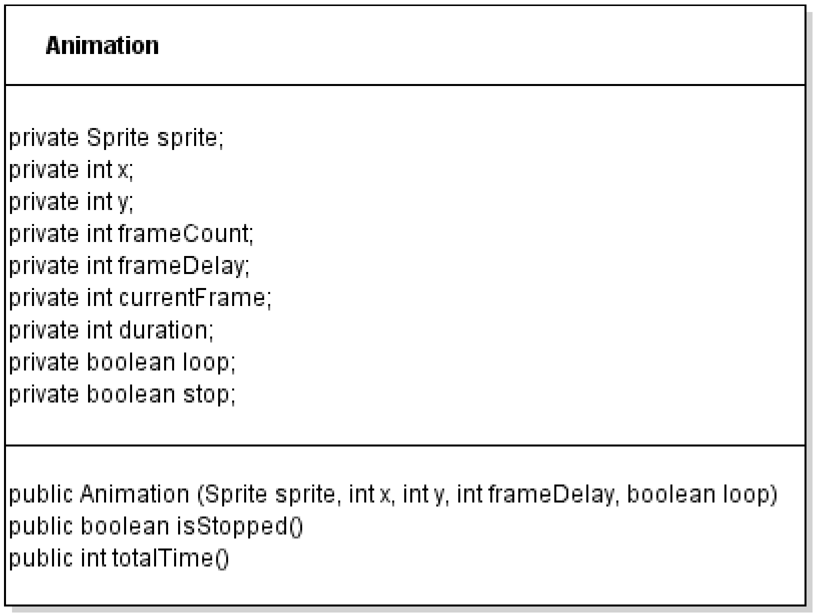
Maps:



```
Maps

protected static int width;
protected static int height;
private static int[][] levelMap;
protected static String[] mapPaths;

public static int [][]parseLevelFile(String resourceLocation)
public static HashMap createWallMap( int mapNumber)
public static int getTank1XStart()
public static int getTank1YStart()
public static int getTank2XStart()
public static int getTank2YStart()
public static int getWidth()
public static int getHeight()
```

The maps class parses text-based map files into a 2D array and extracts tank and wall coordinates. More maps can be added by adding another text file and adding the resource path to the string array map paths.
getMap(int): returns the resource path for the indicated map
createWallMap(int): takes the map number, creates wall objects and returns a hashmap of the walls added. This hashmap is used in GameWorldPanel.
parseLevelFile(String): takes the map file's resource location and creates the 2D array

Animation:

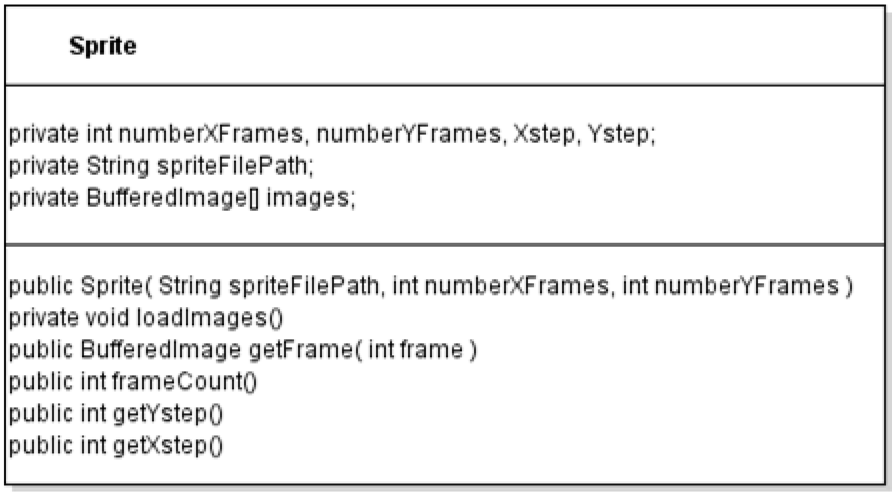


```
Animation

private Sprite sprite;
private int x;
private int y;
private int frameCount;
private int frameDelay;
private int currentFrame;
private int duration;
private boolean loop;
private boolean stop;

public Animation (Sprite sprite, int x, int y, int frameDelay, boolean loop)
public boolean isStopped()
public int totalTime()
```

The Animation class contains an X and Y coordinate, count of frames, delay between frames, current frame, duration of the animation, ability to loop, and a Boolean to check if it is stopped.

The animation takes place within the repaint method which simply parses through the sprite frames according to the parameters given.

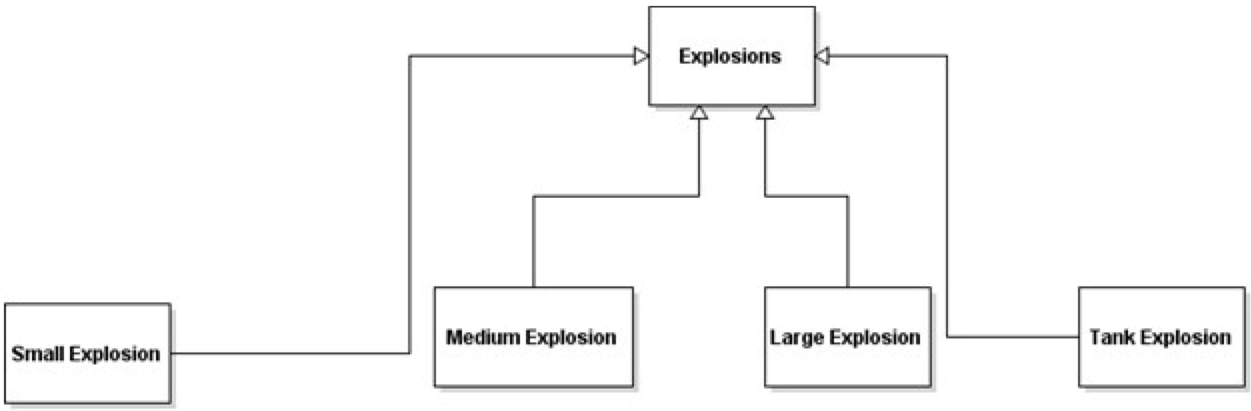
Sprite:



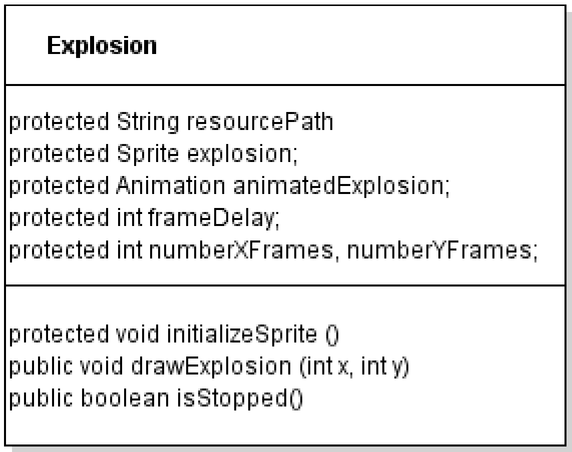
```
Sprite

private int numberXFrames, numberYFrames, Xstep, Ystep;
private String spriteFilePath;
private BufferedImage[] images;

public Sprite( String spriteFilePath, int numberXFrames, int numberYFrames )
private void loadImages()
public BufferedImage getFrame( int frame )
public int frameCount()
public int getYstep()
public int getXstep()
```

The sprite class breaks images up into frames and is capable of retrieving requested frames. It requires a certain number of X frames and Y frames. The Xstep and Ystep is generated by the image size and the number of frames.

Explosions:



Explosion:

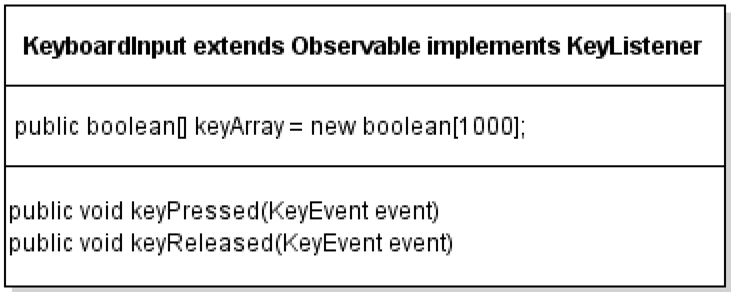

Explosion

protected String resourcePath
protected Sprite explosion;
protected Animation animatedExplosion;
protected int frameDelay;
protected int numberXFrames, numberYFrames;

protected void initializeSprite ()
public void drawExplosion (int x, int y)
public boolean isStopped()

There are four different types of explosions. The small, medium, and large explosions are for when a projectile hits a wall or a tank. The tank explosion is for when the tank dies. The explosion class creates the explosion animations.
Explosions contain a resource path, Sprite, Animation, frameDelay, and a certain number of X frames and Y frames.
initializeSprite(): creates a sprite based on the particular explosion image
drawExplosion(int, int): creates an Animation based on the sprite created with the passed in X and Y coordinates
The constructors within the child classes define the resource path, frame delay, and number of X and Y frames. They also take an X and Y coordinate in order to place the explosion animation correctly.

Keyboard Input:


KeyboardInput extends Observable implements KeyListener

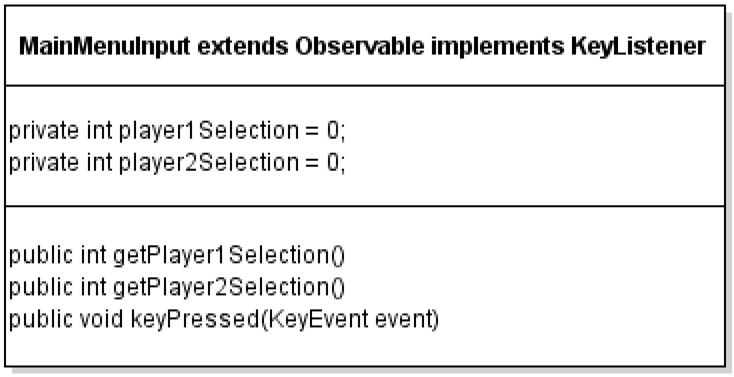
public boolean[] keyArray = new boolean[1000];

public void keyPressed(KeyEvent event)
public void keyReleased(KeyEvent event)

The keyboard input class contains a Boolean array indicating the keys currently pressed.
keyPressed (KeyEvent): sets the Boolean array indexed by the key code from the key stroke to true.
keyReleased(KeyEvent): sets the Boolean array indexed by the key code from the key stroke to false.

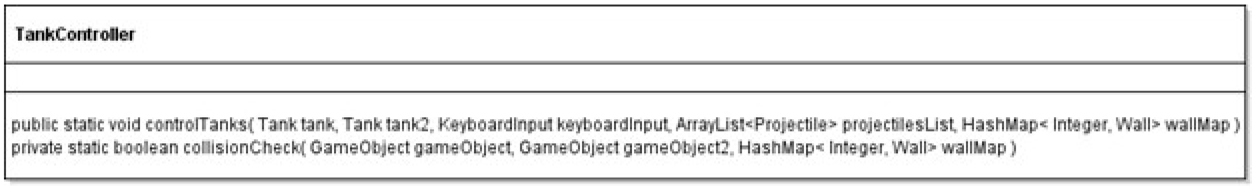
Main Menu Input (not finished):

The main menu input class takes in key presses from the user so they can choose what tank to use. This is not currently fully implemented.



```
MainMenuInput extends Observable implements KeyListener

private int player1Selection = 0;
private int player2Selection = 0;

public int getPlayer1Selection()
public int getPlayer2Selection()
public void keyPressed(KeyEvent event)
```

Tank Controller:

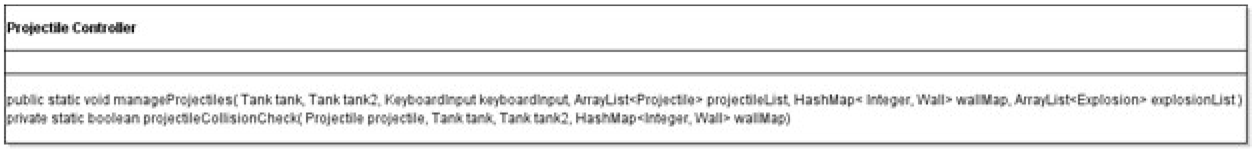


```
TankController

public static void controlTanks( Tank tank, Tank tank2, KeyboardInput keyboardInput, ArrayList<Projectile> projectilesList, HashMap< Integer, Wall> wallMap )
private static boolean collisionCheck( GameObject gameObject, GameObject gameObject2, HashMap< Integer, Wall> wallMap )
```

The tank controller class checks the Boolean key array from keyboard input and performs different actions based on what keys are currently being pressed.

Collision checks are done upon tank movement. The tank is moved to the last non-colliding coordinates if it collides upon movement. If no collision occurs the safe zone is updated.

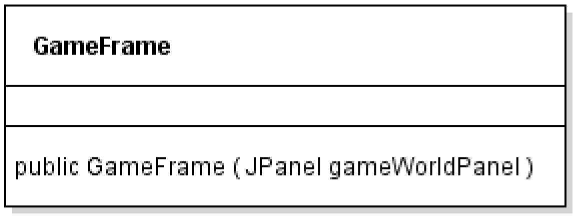
Projectile Controller:



```
Projectile Controller

public static void manageProjectiles( Tank tank, Tank tank2, KeyboardInput keyboardInput, ArrayList<Projectile> projectileList, HashMap< Integer, Wall> wallMap, ArrayList<Explosion> explosionList )
private static boolean projectileCollisionCheck( Projectile projectile, Tank tank, Tank tank2, HashMap<Integer, Wall> wallMap)
```

The projectile controller class checks if a collision occurred between the projectiles and any of the other game objects. It also makes objects take damage and adds an explosion animation if a collision occurred.

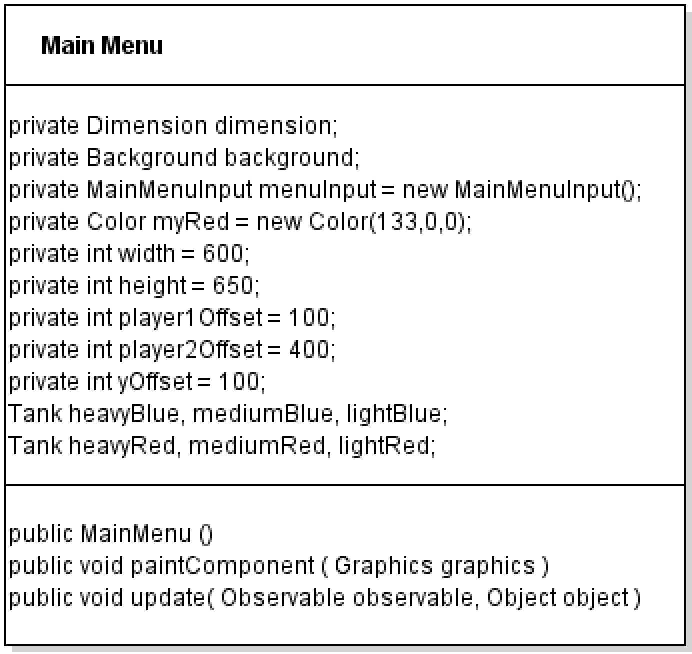
GameFrame:



```
GameFrame

public GameFrame ( JPanel gameWorldPanel )
```

The game frame is the primary game frame.

Main Menu (not finished):



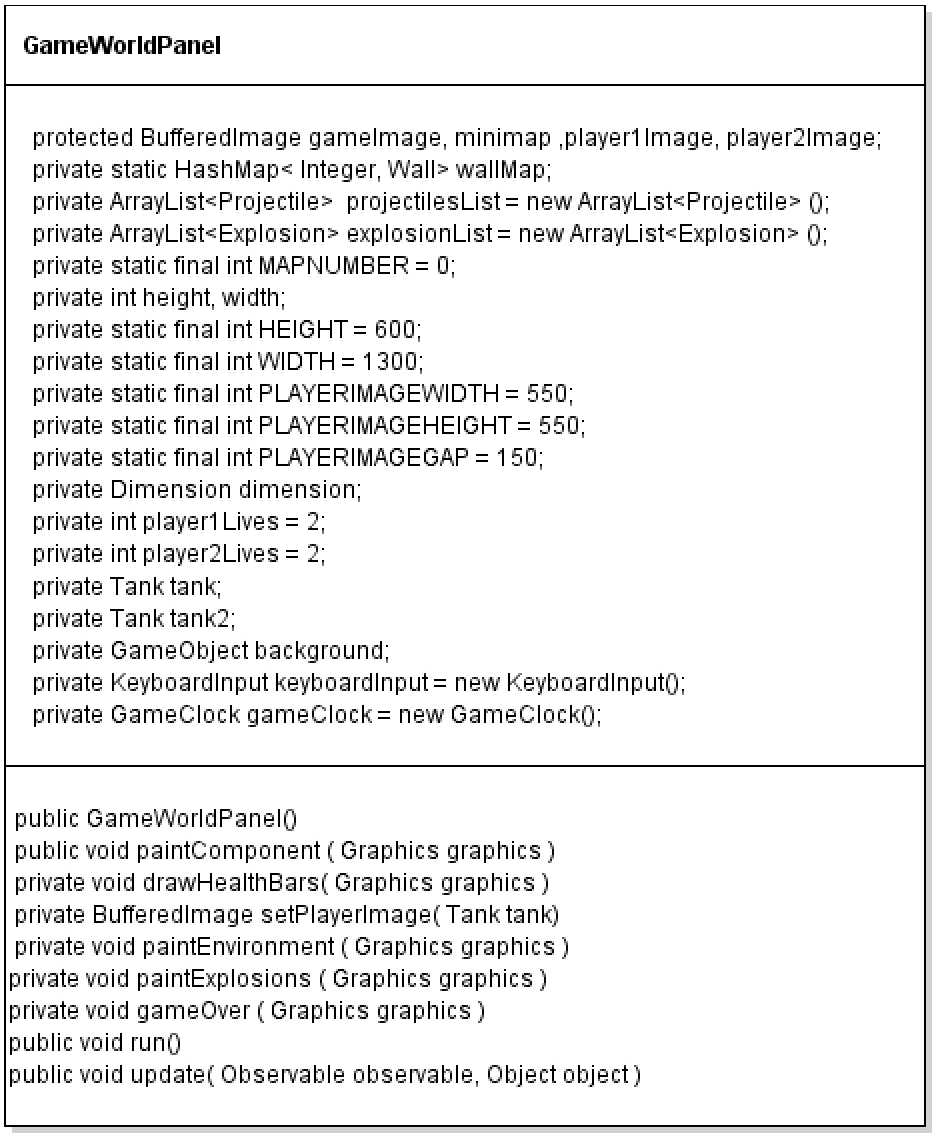
Main Menu

```
private Dimension dimension;
private Background background;
private MainMenuInput menuInput = new MainMenuInput();
private Color myRed = new Color(133,0,0);
private int width = 600;
private int height = 650;
private int player1Offset = 100;
private int player2Offset = 400;
private int yOffset = 100;
Tank heavyBlue, mediumBlue, lightBlue;
Tank heavyRed, mediumRed, lightRed;
```

```
public MainMenu ()
public void paintComponent ( Graphics graphics )
public void update( Observable observable, Object object )
```

The main menu class draws out the main menu for the selection of tanks.

Game World Panel:



GameWorldPanel

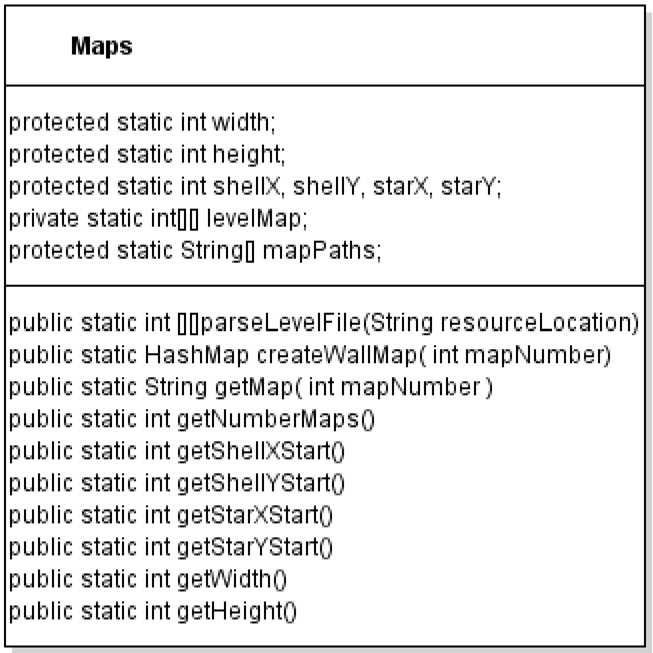
```
protected BufferedImage gameImage, minimap ,player1Image, player2Image;
private static HashMap< Integer, Wall> wallMap;
private ArrayList<Projectile>  projectilesList = new ArrayList<Projectile> ();
private ArrayList<Explosion> explosionList = new ArrayList<Explosion> ();
private static final int MAPNUMBER = 0;
private int height, width;
private static final int HEIGHT = 600;
private static final int WIDTH = 1300;
private static final int PLAYERIMAGEWIDTH = 550;
private static final int PLAYERIMAGEHEIGHT = 550;
private static final int PLAYERIMAGEGAP = 150;
private Dimension dimension;
private int player1Lives = 2;
private int player2Lives = 2;
private Tank tank;
private Tank tank2;
private GameObject background;
private KeyboardInput keyboardInput = new KeyboardInput();
private GameClock gameClock = new GameClock();
```

```
public GameWorldPanel()
public void paintComponent ( Graphics graphics )
private void drawHealthBars( Graphics graphics )
private BufferedImage setPlayerImage( Tank tank)
private void paintEnvironment ( Graphics graphics )
private void paintExplosions ( Graphics graphics )
private void gameOver ( Graphics graphics )
public void run()
public void update( Observable observable, Object object )
```

The purpose of the game world panel is to manage and draw out everything that is happening in the game.
This panel observes the clock and the keyboard input and updates every time they change.
The update method utilizes the tankController and the projectileController to handle movement, collisions, damage, and animations. The lives of each player decrement here if a tank dies.
The current map state is saved onto an image. This image is scaled and clipped to create the player images and the mini map.

Super Rainbow Reef:

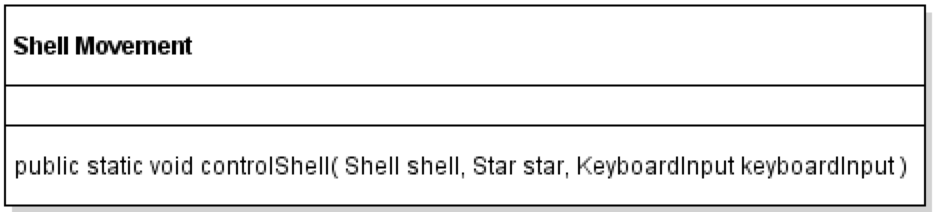
Maps:



```
Maps

protected static int width;
protected static int height;
protected static int shellX, shellY, starX, starY;
private static int[][] levelMap;
protected static String[] mapPaths;

public static int [][]parseLevelFile(String resourceLocation)
public static HashMap createWallMap( int mapNumber)
public static String getMap( int mapNumber )
public static int getNumberMaps()
public static int getShellXStart()
public static int getShellYStart()
public static int getStarXStart()
public static int getStarYStart()
public static int getWidth()
public static int getHeight()
```

The maps class parses text-based map files into a 2D array and extracts the shell, octopus, and star start position and block coordinates.
More maps can be added by adding another text file and adding the resource path to the string array map paths.
getMap(int): returns the resource path for the indicated map
createWallMap(int): takes the map number, creates block objects and returns a hashmap of the walls added. This hashmap is used in gameWorldPanel.
parseLevelFile(String): takes the map file's resource location and creates the 2D array

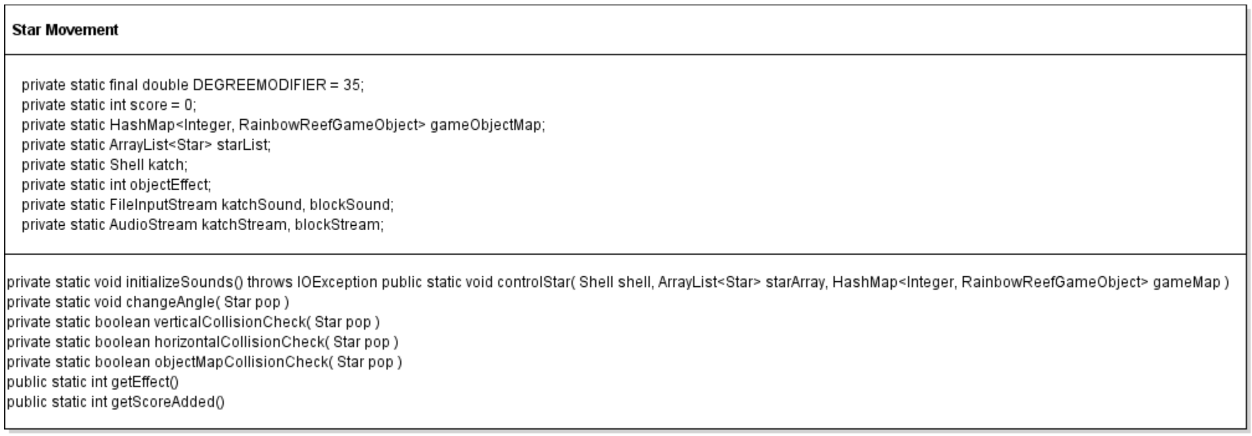
Shell Movement:



```
Shell Movement


public static void controlShell( Shell shell, Star star, KeyboardInput keyboardInput )
```

The shell movement class checks the Boolean key array from keyboard input and moves the shell left or right based on what keys are currently being pressed.   It also fires the star if the enter key is pressed

Star Movement:



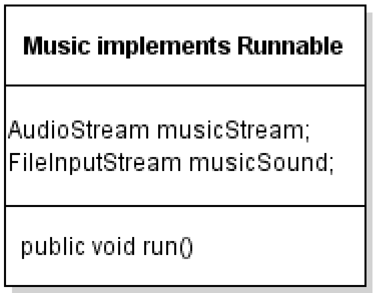
Star Movement

```
private static final double DEGREEMODIFIER = 35;
private static int score = 0;
private static HashMap<Integer, RainbowReefGameObject> gameObjectMap;
private static ArrayList<Star> starList;
private static Shell katch;
private static int objectEffect;
private static FileInputStream katchSound, blockSound;
private static AudioStream katchStream, blockStream;
```

```
private static void initializeSounds() throws IOException public static void controlStar( Shell shell, ArrayList<Star> starArray, HashMap<Integer, RainbowReefGameObject> gameMap )
private static void changeAngle( Star pop )
private static boolean verticalCollisionCheck( Star pop )
private static boolean horizontalCollisionCheck( Star pop )
private static boolean objectMapCollisionCheck( Star pop )
public static int getEffect()
public static int getScoreAdded()
```

The star movement class controls the movement of the star. It also checks if the star collides with any of the blocks, the shell or the octopus.
public static void controlStar(Shell shell, ArrayList<Star> starArray, HashMap<Integer, RainbowReefGameObject> gameMap): moves the star based on where it colllides
private static void changeAngle(Star pop): changes the angle of the star based on where on the shell it hits.
private static boolean verticalCollisionCheck(Star pop): checks if the star collides vertically with any of the game objects and places it in a safe zone if it does.
private static boolean horizontalCollisionCheck(Star pop): checks if the star collides horizontally with any of the game objects and places it in a safe zone if it does
private static boolean objectMapCollisionCheck(Star pop): checks if the star collides with any of the game objects and removes them from the hashmap. It also plays audio when the star collides with an object.
public static int getEffect(): returns the effect of each object
public static int getScoreAdded(): returns the score of the game object

Music:



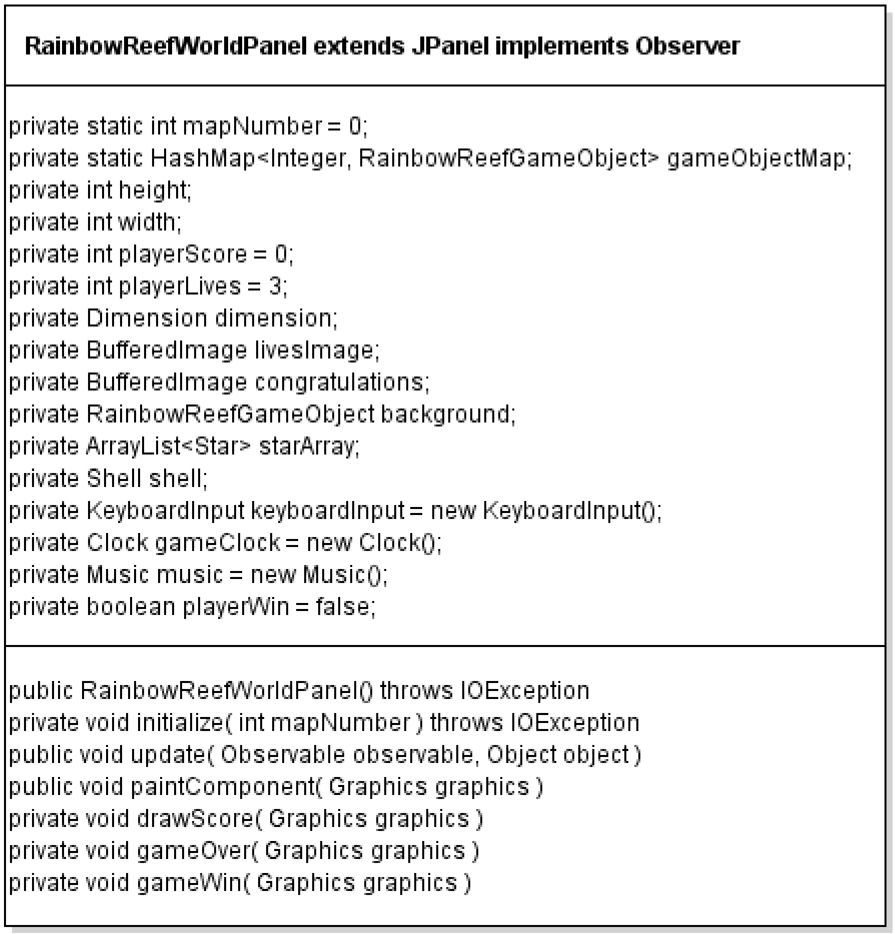
Music implements Runnable

```
AudioStream musicStream;
FileInputStream musicSound;
```

```
public void run()
```

The music class plays background music on a thread.

Game World:



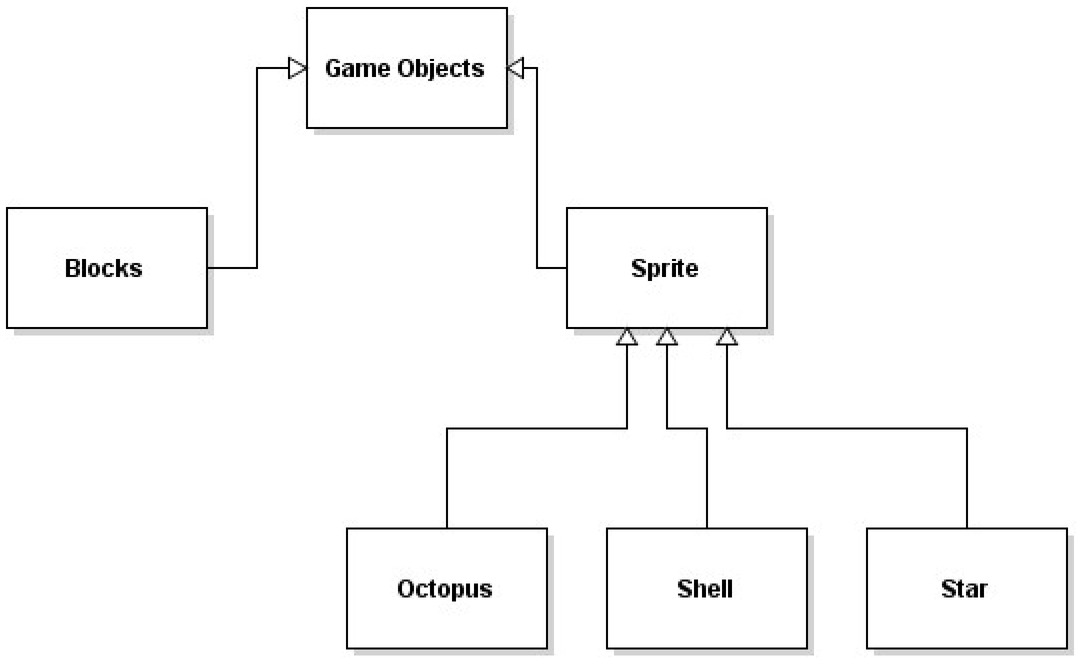
RainbowReefWorldPanel extends JPanel implements Observer

```
private static int mapNumber = 0;
private static HashMap<Integer, RainbowReefGameObject> gameObjectMap;
private int height;
private int width;
private int playerScore = 0;
private int playerLives = 3;
private Dimension dimension;
private BufferedImage livesImage;
private BufferedImage congratulations;
private RainbowReefGameObject background;
private ArrayList<Star> starArray;
private Shell shell;
private KeyboardInput keyboardInput = new KeyboardInput();
private Clock gameClock = new Clock();
private Music music = new Music();
private boolean playerWin = false;
```

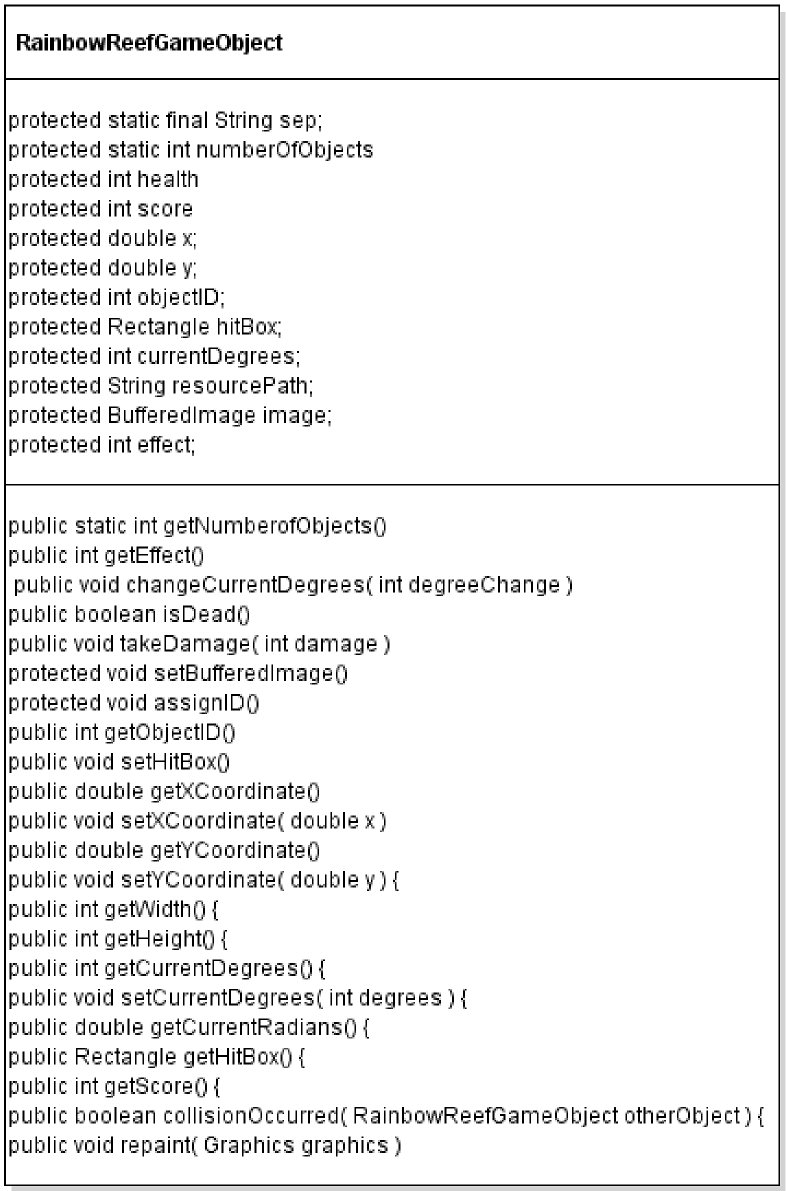
```
public RainbowReefWorldPanel() throws IOException
private void initialize( int mapNumber ) throws IOException
public void update( Observable observable, Object object )
public void paintComponent( Graphics graphics )
private void drawScore( Graphics graphics )
private void gameOver( Graphics graphics )
private void gameWin( Graphics graphics )
```

The rainbow reef world panel draws out everything that is happening in the game and instantiates all of the objects.
private void initialize(int mapNumber): creates a new star object, shell object and background, calls the createwallmap method for the map number that is passed through
public void update(Observable observable, Object object): updates the star and shell movement every time the clock ticks and also checks for special effects.
public void paintComponent(Graphics graphics): draws out the background, blocks, shell, gameover, and congratulation screen
private void drawScore(Graphics graphics): draws out the player score
private void gameOver(Graphics graphics): draws out the game over screen, deletes the game clocks observers, and removes the key listener
private void gameWin(Graphics graphics : draws out the congratulations screen,deletes the game clocks observers, and removes the key listener

Game Objects:



Game Object:



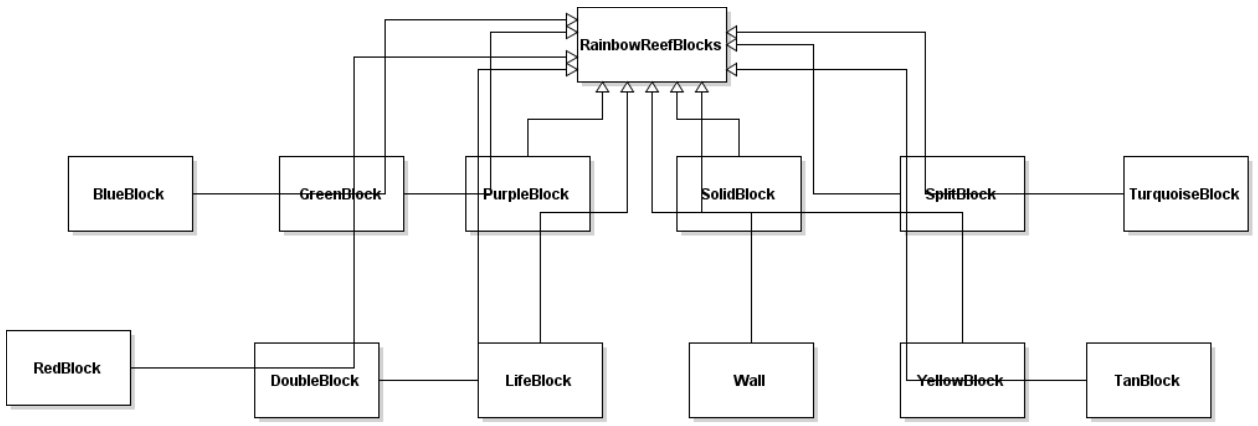
RainbowReefGameObject

```
protected static final String sep;
protected static int numberOfObjects
protected int health
protected int score
protected double x;
protected double y;
protected int objectID;
protected Rectangle hitBox;
protected int currentDegrees;
protected String resourcePath;
protected BufferedImage image;
protected int effect;
```

```
public static int getNumberofObjects()
public int getEffect()
 public void changeCurrentDegrees( int degreeChange )
public boolean isDead()
public void takeDamage( int damage )
protected void setBufferedImage()
protected void assignID()
public int getObjectID()
public void setHitBox()
public double getXCoordinate()
public void setXCoordinate( double x )
public double getYCoordinate()
public void setYCoordinate( double y ) {
public int getWidth() {
public int getHeight() {
public int getCurrentDegrees() {
public void setCurrentDegrees( int degrees ) {
public double getCurrentRadians() {
public Rectangle getHitBox() {
public int getScore() {
public boolean collisionOccurred( RainbowReefGameObject otherObject ) {
public void repaint( Graphics graphics )
```

The blocks and game object sprites extend the game object class. Each game object has a health, score, x coordinate, y corrdinate, id, hitbox, resourcepath, image, and effect.

Blocks:



There are 13 different types of blocks. The wall and the solid blocks are indestructible all of the other blocks can be destroyed.

Block:

Block
public String resourcePath public int Id public int health public int score public int effect
public Block(int xCoordinate, int yCoordinate)

Each block has a resource path, id, health, score, and effect. The x and y coordinate for each block is set.

Game Objects Sprites:

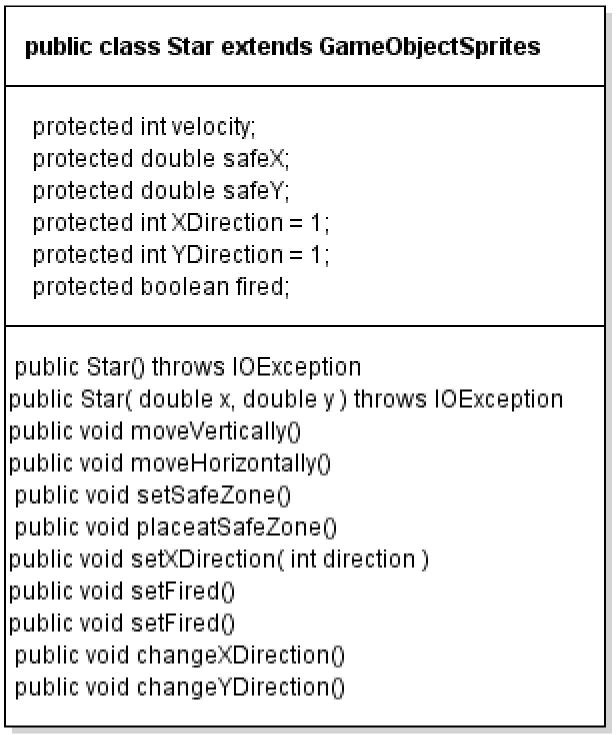
GameObjectSprites extends RainbowReefGameObject

protected int XStep, YStep, frames, XFrames, YFrames;
protected int currentFrame = 0;
private BufferedImage[] imageFrames;

protected void loadImages() throws IOException protected void setBufferedImage() throws IOException public BufferedImage getFrame(int frame) public int frameCount() public int getYstep(public int getXstep() public void repaint(Graphics graphics)

The GameObjectSprites class extends the GameObject class. The sprite class breaks images up into frames and is capable of retrieving requested frames. It requires a certain number of X frames and Y frames. The Xstep and Ystep is generated by the image size and the number of frames.

Star:



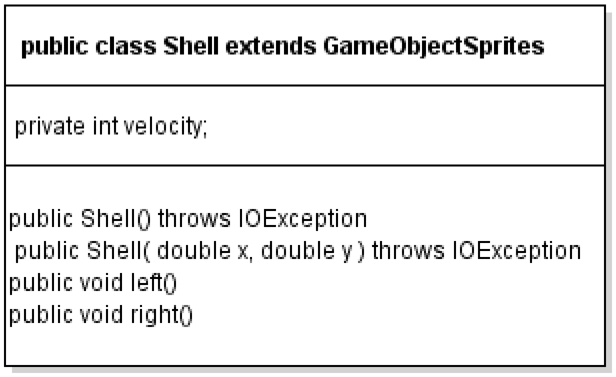
public class Star extends GameObjectSprites

```
  protected int velocity;
  protected double safeX;
  protected double safeY;
  protected int XDirection = 1;
  protected int YDirection = 1;
  protected boolean fired;
```

```
 public Star() throws IOException
public Star( double x, double y ) throws IOException
public void moveVertically()
public void moveHorizontally()
 public void setSafeZone()
 public void placeatSafeZone()
public void setXDirection( int direction )
public void setFired()
public void setFired()
 public void changeXDirection()
 public void changeYDirection()
```

The star class sets the health, velocity, xframes and y frames for the star.
public Star(double x, double y) throws IOException: sets the x and y coordinates for the star and sets a hibox for the star
public void moveVertically(): changes the y coordinate for the star using velocity and angle and sets a hitbox
public void moveHorizontally(): changes the x coordinate for the star using velocity and angle and sets a hitbox
public void setSafeZone(): sets the noncolliding x and y coordinate
public void placeAtSafeZone(): places star in noncolliding x and y coordinates
public void setXDirection(int direction): sets the x direction
public void setFired(): sets fired to true
public boolean isFired(): checks if the star is fired
public void changeXDirection(): changes the x direction by -1
public void changeYDirection() : changes the y direction by -1

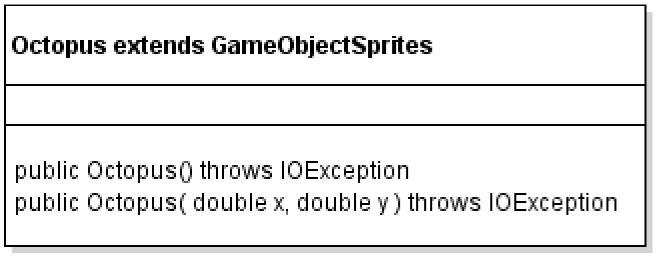
Shell:



```
public class Shell extends GameObjectSprites
-------------------------------------------
private int velocity;
-------------------------------------------
public Shell() throws IOException
 public Shell( double x, double y ) throws IOException
public void left()
public void right()
```

The shell class sets the velocity, health, number of x frames, and y frames for the shell. The left and right methods changes the x and coordinates of the shell and sets a hitbox for the shell.

Octopus:



```
Octopus extends GameObjectSprites
-------------------------------------------

-------------------------------------------
public Octopus() throws IOException
public Octopus( double x, double y ) throws IOException
```

The octopus class extends the GameObjectSprites class so that the octopus can be animated. It also sets the health, score, effect, and x and y position for the octopus.