

CSC 413 Term Project

Fall 2017

Team 04:

Ian Dennis & Arnold Yu

Table of Contents

I. Introduction:	3
II. Project Information:.....	4
III. Scope of Work:	6
IV. Background:	7
V. Assumptions:	8
VI. Tank Game Class Diagram:	9
VII. Rainbow Reef Class Diagram:	10
VIII. Shared Classes:	11
IX. CollidableObj:	13
X. GameEvents:	15
XI. GameObject:	17
XII. KeyControl:	19
XIII. MoveableObj:	20
XIV. Player:	23
XV. SoundPlayer:	30
XVI. Wall:.....	32
XVII. Tank Game Specific Classes:	35
XVIII. Shell:	36
XIX. Tank:	40
XX. TankGame.....	45
XXI. Rainbow Reef Specific Classes:	46
XXII. Blocks:	47
XXIII. Enemy:	49
XXIV. RainbowReefLevel:	51
XXV. RainbowReefWorld:	
XXVI. Slider:	
XXVII. Star:	
XXVIII. Self-Reflection: Software Design and Development:	
XXIX. Conclusion:	

I. Introduction:

This document is intended to serve as the documentation for the Fall 2017 Team 04 Term Project for CSC 413. This document will cover the background, scope, and planning that went into the execution of this project. It will also go into specifics about implementation decisions and design of the classes for each of the two games developed. It will move from the classes reused between the two games, to the classes unique to each game. The documentation will end with a sections dedicated to self-reflection on this exercise in teamwork and software engineering and an ultimate conclusion of the experience and knowledge gained through the effort invested into it. We hope you enjoy the games.

- Ian Dennis & Arnold Yu

II. Project Information

Tank Game:

Github Link for Tank Game:

<https://github.com/CSC-413-SFSU-02/csc413-tankgame-team04>

Instructions for Compiling & Running:

From the cloned git directory run the following command line commands:

Compile:

```
javac RainbowReef/rainbowreef/*
```

Run:

```
java RainbowReef/rainbowreef/RainbowReefWorld
```

Development Environment:

Arnold: Netbeans IDE using JDK 1.8 on a MacBook Air running macOS Sierra

Ian: IntelliJ IDE using JDK 1.8 Update 131 on a MacBook Pro running OSX 10.11.16

Controls:

Player 1

W: Move Forward

A : Rotate Counterclockwise

S : Move Backward

D : Rotate Clockwise

Q : Fire

Player 2

I: Move Forward

J: Rotate Counterclockwise

K: Move Backward

L: Rotate Clockwise

O: Fire

Goal: Destroy the enemy tank before it destroys you!

Rainbow Reef Game:

Github Link for Rainbow Reef Game:

<https://github.com/CSC-413-SFSU-02/csc413-secondgame-team04>

Instructions for Compiling & Running:

From the cloned git directory run the following command line commands:

Compile:

```
javac tankgame/*
```

Run:

```
java tankgame/TankGame
```

Development Environment:

Arnold: Netbeans IDE using JDK 1.8 on a MacBook Air running macOS Sierra

Ian: IntelliJ IDE using JDK 1.8 Update 131 on a MacBook Pro running OSX 10.11.16

Controls:

A : Move Right

D : Move Left

III. Scope of Work:

For this project we were tasked with developing two games. One game was to be a game involving tanks, whose requirements were given to us based on a GameMaker How-To on how to develop these games (in GameMaker, not Java). The second game was up to us to decide on. We were given a list of games (with requirements) that we could choose from, or we were given the option to develop something on our own (with instructor approval). We decided to do the Rainbow Reef game from the list of games.

For both games, we convened to discuss requirements and convert those requirements into software objects. For the Tank Game, we were tasked with creating a two-player, top-down, 2-player versus game. The objective was to destroy the other player by shooting and out-maneuvering them until they ran out of lives. The game was to be split-screen, have sound, and be controlled by the keyboard. From this, we developed a list of software objects and then prioritized their importance of what should be developed first, so that we could develop and test in an iterative fashion (instead of all at once). We also tried to determine dependencies between objects so that we could try and work on the game as asynchronously as possible (since we have very different/busy schedules).

The second game, Rainbow Reef, was developed in a similar fashion. We met up and discussed the requirements from the blurb about the game in the game list document, decided on software objects, and prioritized the order in which to complete them. The requirements for the Rainbow Reef game were to create a single player game where the player controls a bar that a ball rebounds off of to hit and break blocks. The goal was to destroy all the enemies by getting the ball to collide with them.

IV. Background and Resources:

We were given a wealth of resources to look through when we embarked on this project. From source code for other 2-D Java games, to slide decks and examples on event-handling, multithreading, and design patterns that might be especially useful in the context of making a game. We were also graciously given art and sound assets for the game so that we would not have to make our own. In all, we were well equipped to set-off on our first foray into developing games in Java.

We were given multiple implementations of the game Airstrike (a 2D plane shoot-em-up) as examples of how to design classes for the games, as well as how to manage the GUI, sound, and collisions aspects of a game. Having actual code to reference how to accomplish specific tasks was extremely helpful, although it did require some digging into Java documentation to learn exactly how and why certain tasks (such as drawing all the images on the JFrame) were done in particular ways.

The in class examples on event-handling and requirements gathering were also very useful for us to hit the ground running with this game. The idea of developing a game without much knowledge on Java GUI topics and event-handling seemed daunting until we went over some examples in class. Looking at pre-written code is one thing, but having somebody walk you through it is on another level of usefulness because of the live feedback.

The most useful resource, however, was receiving all the art and sound assets that would be necessary for the game. Initially, we were more worried about how we were going to make sprites and music than actually coding and designing of the game (though Arnold is actually good at art, while I am terrible at it). Having the art assets definitely made the task of developing two games seem less intimidating than it had when we had originally been told about the project.

V. Assumptions

This is the list of assumptions we made for each game:

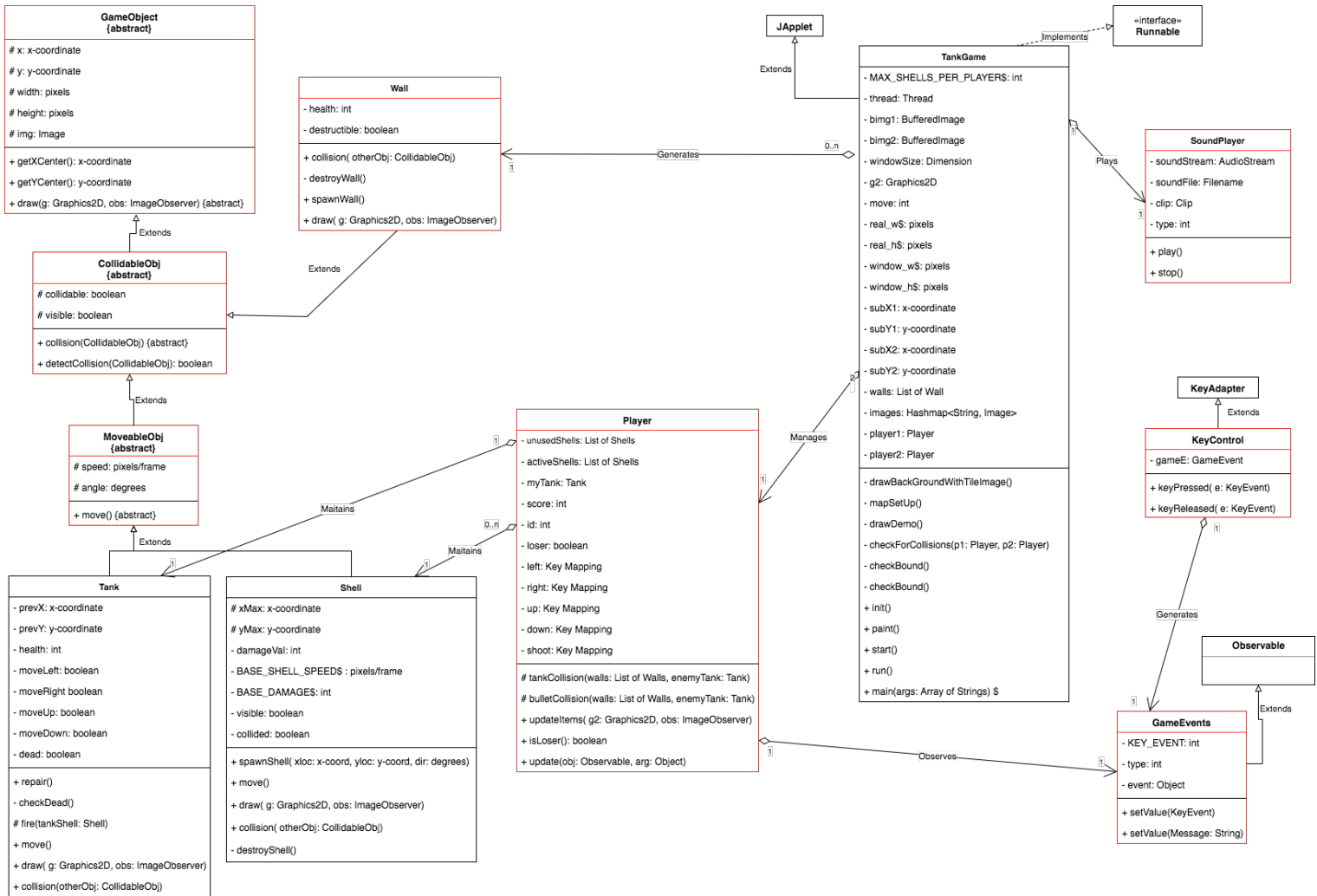
Tank Game:

- The game was for only 2 players
- The window size is fixed
- Resources for the game will always be in a specific, relative folder to the source
- Coordinate system:
 - o +y direction is down and +x is to the right
 - o Straight-upward (-y direction) is 0 degrees, and degrees increase clockwise
 - o (0, 0) is the top left corner of the window
- The tank can move forward and backward, or rotate

Rainbow Reef:

- The game is only for one player
- The window size is fixed
- Game resources are in a directory relative to the source code
- Coordinate system:
 - o +y direction is down and +x is to the right
 - o (0, 0) is the top left corner of the window
- The walls surrounding the arena will keep the ball from going out of bounds in any other direction other than straight down
- The Slider will only ever collide with Walls and the Star
 - o Since it can only move along the x-axis

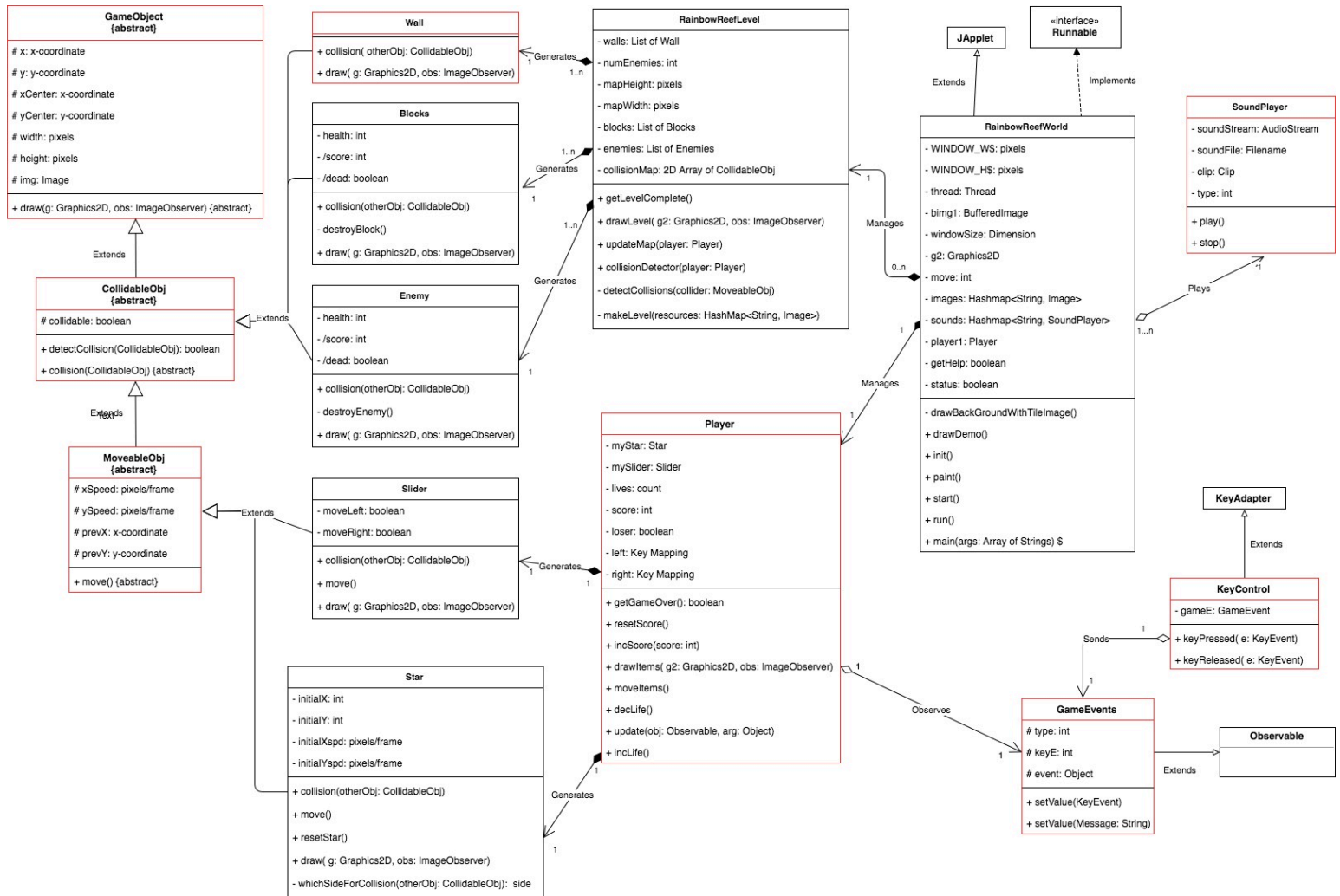
VI. Tank Game Class Diagram



Online Link for the Diagram (Google Drive):

https://drive.google.com/file/d/12lAt-4gLyrswnD5Qs0UeEtk0_5eLV04/view?usp=sharing

VII. Rainbow Reef Class Diagram



The diagram is quite large, so for easier viewing please view it from this link:

<https://drive.google.com/file/d/19osEs0EpkCIm1UaUxp2WQoANeCU-bYmn/view?usp=sharing>

One may notice that nearly half of the classes for Rainbow Reef were reused from the Tank Game, and even those that weren't reused had a strong basis on the existing classes from the Tank Game.

VIII. Shared Classes

There were quite a few shared classes between the two games, which was expected given that games in general share similar logical features. The list of shared classes is:

1. CollidableObj Class – Abstract base class for objects that can be collided with
 - extends GameObject
2. GameEvents Class– Class for sending out notifications to observers about game events
 - extends Observable
3. GameObject Class – Abstract base class of any object that needs to be drawn
4. KeyControl Class – Class for monitoring Key Events
 - extends KeyAdapter
5. MoveableObj Class – Abstract base class of any object that can move
 - extends CollidableObj
6. Player Class – Class responsible for managing all player-controlled Game Objects
7. SoundPlayer Class – Class responsible for playing sound files
8. Wall Class – Class for walls in both games
 - extends CollidableObj
 - Different between both games, because walls in Rainbow Reef aren't destructable

It should be noted that some of these classes were modified to better fit their role in a particular game, so their contents were not always identical. The responsibilities and the overall function of these classes did carry between games, however.

From here, we will delve into a more detailed explanation of each of these classes and their implementations.

IX. CollidableObj Class

Rainbow Reef

CollidableObj {abstract}
collidable: boolean
+ getCollidable() : boolean
+ setCollidable(collidable)
+ detectCollision(CollidableObj): boolean
+ collision(CollidableObj) {abstract}

Tank Game

CollidableObj {abstract}
collidable: boolean
visible: boolean
+ collision(CollidableObj) {abstract}
+ detectCollision(CollidableObj): boolean

Class Reasoning:

Objects need to be able to collide or it will be near impossible to figure out if a shot bullet hit the opposing players tank, or anything for that matter. Since the only objects that needed this information was a subset of the objects that needed to draw themselves on screen, but didn't necessarily move, it was decided that this would be a subclass of the drawable objects, but a superclass of the objects that moved.

Similar Field and Method Implementation:

protected boolean collidable

- Created so that if a game object was destroyed, it didn't have to be removed from the map.
- Protected access because the collidableObj itself handles collision detection, so objects outside of it's inheritance tree do not need to know if the object can collide with things because they won't be handling collision calculations, and are expected to call on the detectCollision() method.

public abstract void collision(CollidableObj)

- Each CollidableObj should handle their own behavior when they collide with another CollidableObj. How this is handled is up to the subclass to decide.
- Requires the object this class collided with because there may be different behavior depending on what the exact type of CollidableObj it is
- Public access so that outside classes can notify the object that it collided

public void detectCollision(CollidableObj)

- A general collision detection algorithm that uses the intersection of rectangles that represents the CollidableObjs in question to see if they collided.
- Requires another collidableObj to check against.
- Public access to allow outside objects to manage what to compare for collisions.

```
public boolean detectCollision(CollidableObj otherObj) {
    boolean collision = false;

    if (this.collidable && otherObj.collidable) {
        Rectangle my_area = new Rectangle(this.getX(), this.getY(), this.getWidth(), this.getHeight());
        Rectangle otherObj_area = new Rectangle(otherObj.getX(), otherObj.getY(), otherObj.getWidth(), otherObj.getHeight());

        collision = my_area.intersects(otherObj_area);
    }
    // Return true if there is a collision
    return collision;
}
```

Algorithm for detecting collision with another CollidableObj

Tank Game Specific Functionality:

protected boolean visible

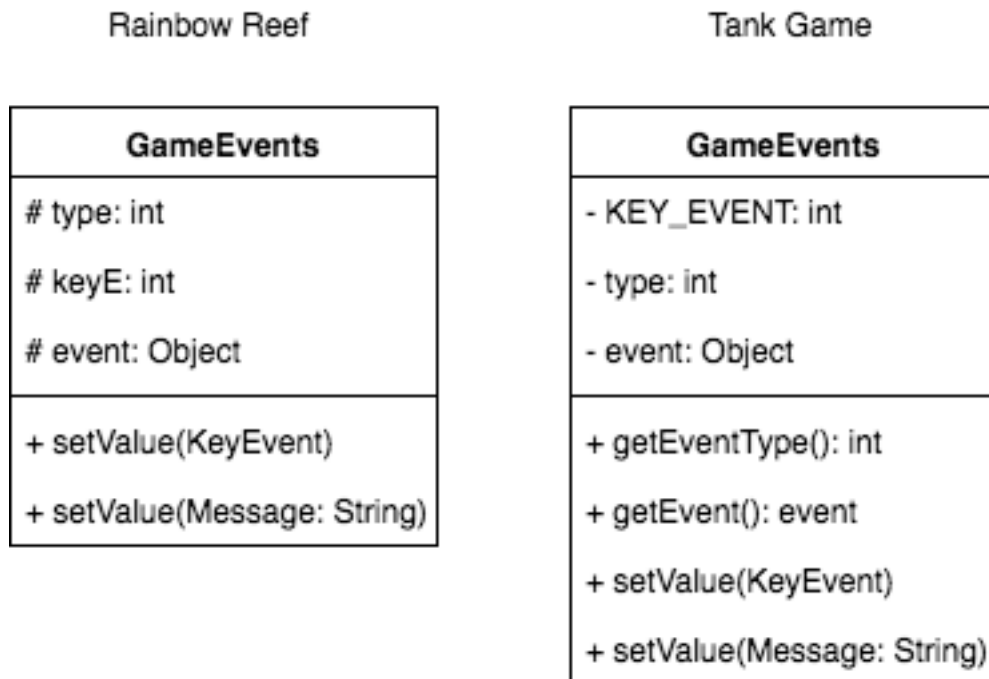
- Used so that when a CollidableObj is destroyed, it doesn't necessarily have to be discarded
- Saves from overhead of creating new CollidableObjs every time they are destroyed, instead they can be hidden from the player

Rainbow Reef Specific Functionality:

get/set for collidable

- Deemed necessary because of a more complex collision calculation in this game.

X. GameEvents Class



Class Reasoning:

We needed an asynchronous way to notify particular objects about changes in game state (like when a key was pressed). Since checking for a key press at a fixed interval will not always catch whether a key will be pressed (tedious to check all over one game loop), an Observable Object was decided on.

Similar Field and Method Implementation:

protected int type

- Used to distinguish the type of game event between a key event and everything else.
- The other event types never got used in the long run

private final static KEY_EVENT/ protected keyE

- Constant for identifying KeyEvents
- Access changed between games because it's nice to have access to this value if this class gets subclassed

protected/private Object event

- Used as a container for a particular event received. Set as an Object for flexibility of type.

public setValue(KeyEvent) / public setValue(String)

- KeyEvent variation ended up the only one being used between both games
- Public access so that Listener Objects could make use of the GameEvents
- Allows for there to be one GameEvent that keeps getting sent out

Deprecated between Games:

public int getEventType() and public int getEvent():

- More tedious to use when Observers are the only ones who really ever see these values, so protected access on these values seemed good enough.

XI. GameObject Class

Rainbow Reef

GameObject {abstract}
x: x-coordinate
y: y-coordinate
xCenter: x-coordinate
yCenter: y-coordinate
width: pixels
height: pixels
img: Image
+ draw(g: Graphics2D, obs: ImageObserver) {abstract}

Tank Game

GameObject {abstract}
x: x-coordinate
y: y-coordinate
width: pixels
height: pixels
img: Image
+ getXCenter(): x-coordinate
+ getYCenter(): y-coordinate
+ draw(g: Graphics2D, obs: ImageObserver) {abstract}

Class Reasoning:

Since these are graphics based games, some objects needed to be able to draw themselves on the screen. In order to be flexible about how these objects draw themselves, yet enforce that they do, an abstract base class was decided on.

Similar Field and Method Implementation:

protected int x

- x-coordinate of the top left corner of where the image for the object will be drawn
- Since GameObjects are responsible for drawing themselves, and collidableObjs and MoveableObjs subclass from this, protected access allows for easy access for subclasses and makes other classes use the get/sets.

protected int y

- y-coordinate of the top left corner of where the image for the object will be drawn
- Same reason as x for protected access

protected int width

- Width of the image representing the object in pixels
- Necessary for drawing and collision
- Protected access for same reason listed as x

protected int height

- Height of the image representing the object
- Reasoning for existence the same as width

protected int img

- Image representing the GameObject
- Needed to have a GUI
- protected for the same reason as everything else in this section

public abstract draw(Graphics2D, ImageObserver)

- Forces subclasses to have to determine how to draw themselves.
- Since this can vary between subclasses, the method is made abstract.

XII. KeyControl Class

Rainbow Reef

KeyControl
- gameE: GameEvent
+ keyPressed(e: KeyEvent)
+ keyReleased(e: KeyEvent)

Tank Game

KeyControl
- gameE: GameEvent
+ keyPressed(e: KeyEvent)
+ keyReleased(e: KeyEvent)

Class Reasoning:

We needed a class that could monitor for key presses (especially those that pertain to the controls of the game), yet send out notifications in a context that our game objects could understand. So we decided to subclass off of KeyAdapter to check for key presses, and use our GameEvents class to handle notification.

Similar Field and Method Implementation:

```
private GameEvent gameE
```

- Needed to notify other GameObjects about key presses asynchronously
- Private access because no other class should need access to this GameEvent

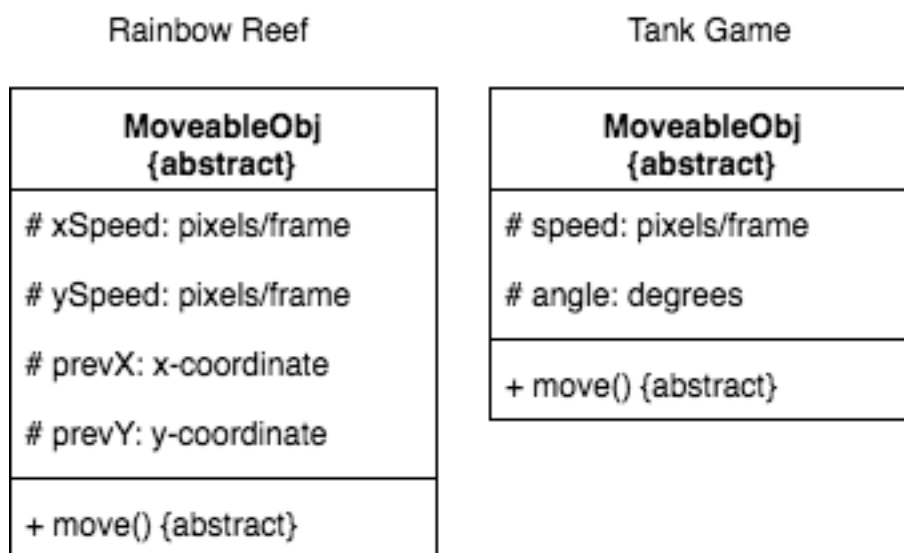
```
public void keyPressed(KeyEvent)
```

- We needed to be able to notify specific objects when a key related to the controls was pressed to start movement/action

public void keyReleased(KeyEvent)

- We needed to be able to notify specific objects when a key related to the controls was released to stop movement/action

XIII. MoveableObj Class



Class Reasoning:

Some of the objects in the game needed to be able to move around, such as the tank and shells. We noticed that these objects were a subset of those that needed to be able to draw themselves and collide with other objects, so we made this a subclass of CollidableObj and part of the GameObject hierarchy of classes. The class was decided to be abstract because different moving objects had different movement requirements, so it was better to delegate and enforce that moveable objects control how they move.

Similar Field and Method Implementation:

public abstract void move()

- MoveableObjs are responsible for defining how they move, since movement can vary; so it was decided to make this method abstract
- Made public so that outside objects with more knowledge of the game state can inform the MoveableObj when to move

Tank Game Specific Functionality:

protected int speed

- Used to determine how fast the a MoveableObj is moving in the direction it is facing
- Needed to figure out how far to displace objects every frame

protected int angle

- Used in conjunction with speed to displace moving objects
- Angle was decided on because of the tank rotating based on angles

Rainbow Reef Specific Functionality:

protected int xSpeed

- Speed for Rainbow Reef eschewed away from the use of angles, and settled for the easier adding of components
- Protected access because speed was only used for calculations done by the subclasses of this class

protected int ySpeed

- Similar reasoning as xSpeed, except this is the y-component

protected int prevX

- Used to store the previous x-location of a moveable object, this was important for keeping track of it in the collision matrix used to find collisions in Rainbow Reef
- Protected for easy access for subclasses, but making other classes that used this value for collisions go through getters

protected int prevY

- Exists for the same reasoning as prevX

XIV. Player Class

Rainbow Reef	Tank Game
Player	Player
<ul style="list-style-type: none"> - myStar: Star - mySlider: Slider - lives: count - score: int - loser: boolean - left: Key Mapping - right: Key Mapping 	<ul style="list-style-type: none"> - unusedShells: List of Shells - activeShells: List of Shells - myTank: Tank - score: int - id: int - loser: boolean - left: Key Mapping - right: Key Mapping - up: Key Mapping - down: Key Mapping - shoot: Key Mapping
<ul style="list-style-type: none"> + getGameOver(): boolean + resetScore() + incScore(score: int) + drawItems(g2: Graphics2D, obs: ImageObserver) + moveItems() + update(obj: Observable, arg: Object) + incLife() + decLife() 	<ul style="list-style-type: none"> # tankCollision(walls: List of Walls, enemyTank: Tank) # bulletCollision(walls: List of Walls, enemyTank: Tank) + updateItems(g2: Graphics2D, obs: ImageObserver) + isLoser(): boolean + update(obj: Observable, arg: Object)

Class Reasoning:

The reasoning for this class is a little more involved than those covered thus far. The idea for the player class came when we were thinking of classes to make for the Tank Game, but initially we were unsure of how to incorporate it in other than to have it keep track of score and lives. As we continued to work on the tank game, we noticed that the main class was beginning to bloat with responsibilities for taking care of virtually all of the GameObjects. As a result, we decided to shift the responsibility of player owned items (like shells and tank) away from the main class and into the Player class. The Player class was

then given the responsibility to maintain the state of these objects, which also helped to decouple them away from the main class.

This class was further refined in the Rainbow Reef game, where the responsibility of calculating collisions was moved to yet another class to try and slim down this class. This had to do with further reorganization of the main class in order to try and cut down its duties to just having to ultimately decide when to render everything in the game loop.

Similar Field and Method Implementation:

private int score

- The score of the player, added to enhance user experience
- Private so that it can only be modification to it is restricted

private boolean loser

- Used as a flag to determine the end of the game
- Private so that nothing modification is restricted, given that this flag can make a particular player lose

Directional Key Mappings (private int left/right/up/down)

- These values are the key mappings for movement
- RainbowReef only uses left and right because the bar can only move left or right
- Private so that other objects cannot change these key bindings once set
- Decouples what keys move the player controlled objects from those objects, especially since key controls can vary between players

public update (Observable obj, Object arg)

- Since the Player object now handles taking notifications of key events, it needs update in order to observe the GameEvents being sent by KeyControl.

Tank Game Specific Functionality:

private Tank myTank

- The Tank object that belongs to the player
- Since the user (player) controls the tank with keyboard inputs, it seemed to make sense to roll it into the Player class, and have the Player object control the movement of the Tank object

private ArrayList<Shell> unusedShells

- Shells that have not been shot yet
- We capped the number of shells to be fired at once to 20, because letting the player shoot infinite shells could lead to hiccups in the game as it calculates collision for each one of the shells
- Since we have a hard limit, we can then reuse the shells by pushing and popping them between two lists without ever having to worry about not having enough shells for the player to shoot
 - o This reduces overhead for memory allocation of shells as well

private ArrayList<Shell> activeShells

- Shells that have been fired by the player, but have not collided or gone “out of bounds” yet
- Used to create a subset of shells that need to have their collision tracked every frame to try and optimize our collision algorithm for the Tank Game

private int id

- An id number to differentiate the players, mainly used for end game printouts
- Private, so that the id number cannot be changed once set by other objects

private int shoot

- The key id of the button for shooting shells
- Needed because the tank shoots on a particular button press
- Made private so that this key can not be changed by outside objects once set

protected void tankCollision(ArrayList<Wall> walls, enemyTank Tank)

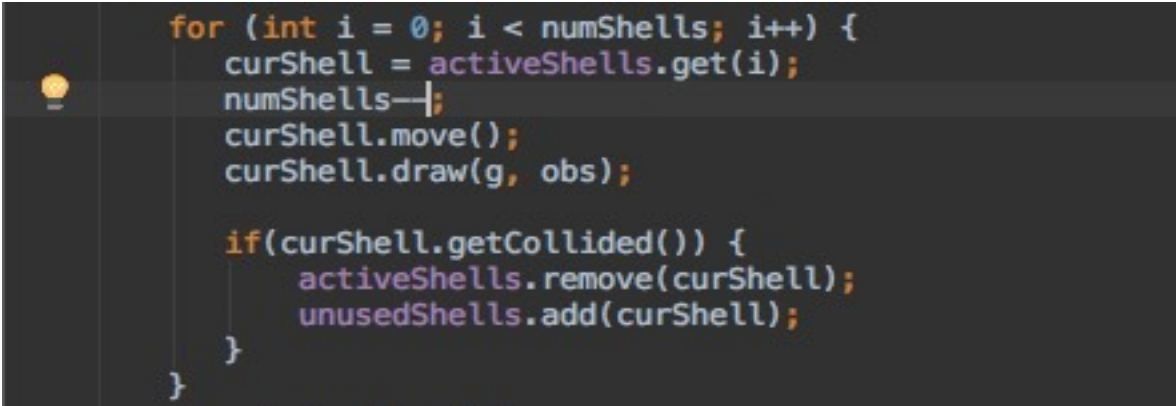
- Simple algorithm to check for checking if the tank collided with anything
- Checks the tank against every wall and the enemyTank using CollidableObj's detectCollision() method
- Calls each CollidableObj's collision function if they collided
 - o Polymorphism at work!
- Does not check against enemy bullets, because there is a separate function that checks for bullet collisions, and we didn't want to do the same case twice
- This algorithm is not scalable for a large number of CollidableObjs!
 - o Since the number of CollidableObjs in the arena is somewhat fixed, it was okay

protected void bulletCollision(ArrayList<Wall> walls, enemyTank Tank)

- Similar to the tankCollision() method
 - o Accept this checks all Shells in the activeShell List instead of just a single Tank
 - o Easy to see how this can get out of hand with large numbers of active shells
- Could not find a way to make this and the tankCollision more generic to reduce code repetition

public updateItems (Graphics2D g2, ImageObserver obs)

- Wrapper function for handling Player object movement and drawing of its Tank and Shells
- Also handles the transfer of active shells to unused shells once they despawn



```

for (int i = 0; i < numShells; i++) {
    curShell = activeShells.get(i);
    numShells--;
    curShell.move();
    curShell.draw(g, obs);

    if (curShell.getCollided()) {
        activeShells.remove(curShell);
        unusedShells.add(curShell);
    }
}

```

Loop to handle the transfer of collided shells between ArrayLists

public boolean isLoser()

- Used to determine if a given player has lost, based on the health value of their Tank
- Necessary for determining the end state of the game
- Public so that the main class can query for the game state in the main game loop

Rainbow Reef Specific Functionality:

private Star myStar

- The star that the player bounces around
- Coupled with a player since the state of the star affects the state of the player
 - o Scoring/Losing Lives
- Private so that access to the Star can be decided by the Player

private Slider mySlider

- The Slider that the Player directly controls
- Coupled to the Player since the user is controlling the Slider, and the Player class is an encapsulation of all values/Objects that are user-related

private int lives

- The number of lives the Player has remaining
- When this goes to zero, the Player loses the game
- Private since this directly affects game state, and so access to it should be heavily protected

public boolean getGameOver()

- Checks for if the Player lost due to running out of lives
- Allows for checking the number of lives without giving other objects access to that value

public void resetScore()

- Used to reset the Player's score to zero
- Currently unused
- Was planned to be used if we were to have each individual level have a high score table

public void incScore(int value)

- Increases the Player's score by a given value
- Allows for a very controlled access point to modify the Player's score

public void drawItems()

- Draws the Slider and Star that the Player owns
- Allows for a nice wrapper function to draw these items, without having another object have to get the entire Object

`public void moveItems()`

- Moves the Slider and the Star that the user owns
- Another convenience function without having to return both Objects to move them
 - o Especially useful for decoupling what wants to move Player items from needing to know what exactly those items are
- Additionally, this is the only way to check if the Star has gone out of bounds, and reset it

`public void incLife()`

- Increases the Player's life by one
- Meant to be used in conjunction with bonus blocks
- Currently Unused

`public void decLife()`

- Decrements the Player's life by one
- Meant to be used in conjunction with other planned obstacles that would cause the player to lose life

XV. SoundPlayer

SoundPlayer	SoundPlayer
- soundStream: AudioStream	- soundStream: AudioStream
- soundFile: Filename	- soundFile: Filename
- clip: Clip	- clip: Clip
- type: int	- type: int
+ play()	+ play()
+ stop()	+ stop()

Class Reasoning:

To enhance user experience, we needed a class that could play sound, but decided that this is a responsibility in and of itself so spun it off into a class. This class loads sound clips that can then be played continuously (like background music) or just once.

Similar Field and Method Implementation:

private AudioStream soundStream

- The AudioStream generated from the song filename
- Private because this should not be changed once loaded

private String soundFile

- The name of the file to get the sound from
- Private because these should not be modified

private Clip clip

- The actual Clip responsible for playing the song
- Should not be modified

private int type

- An int flag to determine how the sound should be played
- 1: Plays the sound continuously in a loop
- 2: Plays the sound once

public void play()

- Wrapper function to have the sound clip play without giving direct access to the Clip Object

public void stop()

- Wrapper object to stop playing a sound clip without giving direct access to the Clip Object

XVI. Wall Class

Rainbow Reef	Tank Game
Wall	Wall
+ draw(g: Graphics2D, obs: ImageObserver) + collision(otherObj: CollidableObj)	- health: int - destructible: boolean + collision(otherObj: CollidableObj) - destroyWall() + spawnWall() + draw(g: Graphics2D, obs: ImageObserver)

Class Reasoning:

This was a requirement of both the games, where the levels for both need to be fringed by and interspersed with walls. For that reason, it was an easy class to include. The Wall itself is mainly responsible for drawing itself, although in Tank Game the Wall has more responsibilities because it can be destroyed. In Rainbow Reef, the walls just serve as an indestructible surface for the Star to bounce off of.

Similar Field and Method Implementation:

```
public void draw (Graphics2D g, ImageObserver obs)
```

- Inherited and overridden from GameObject
- Method invoked to draw the Wall on the Graphics2D object
- Public so that other Objects can call on the Wall to draw itself

collision(CollidableObj otherObj)

- Method for resolving how the Wall behaves when something collides with it
- Inherited and overridden from CollidableObj
- In the Tank Game
 - o If the wall is destructible, being hit by a Shell will destroy it
 - o Else, nothing happens to the wall
- In Rainbow Reef
 - o This function is left empty because the Walls can't be destroyed, so they would not change state from being collided with

Tank Game Specific Functionality:

private int health

- The health value of the wall, set to 1
- This field was meant to be used in a feature to make durable walls (that took more than one shot to destroy)
- Private, so that other Objects couldn't try and destroy the wall through other means

private boolean destructible

- Flag to determine if the Wall can be destroyed
- Requirement from the Tank Game specifications

private void destroyWall()

- Method used to despawn the wall if it is destructible
 - o Just sets the Wall to not visible and not collidable
 - o Makes respawning the wall easier
- Private so that this can only be invoked through the collision method

`public void spawnWall()`

- Makes an invisible wall visible, and collidable
- Planned to use to respawn destroyed Walls based on a timer
- The feature was never fully implemented

XVII. Tank Game Specific Classes

The following classes are unique to the Tank Game. These classes are strongly bound to the specifications of the Tank Game, which is why they did not get completely reused in Rainbow Reef. Some of the logic from these classes did get reused though, even if the whole class could not be. These classes are:

1. Shell Class – Class responsible for the behavior of Shells shot by the Tank
 - Extends MoveableObj
 - Is actually relatively simple because Shells can't change direction
 - Trigonometry is used to determine the path of the Shell
2. Tank Class – Class responsible for the behavior of the Tank
 - Extends MoveableObj
 - Has a unique, but simple collision algorithm
 - This was due to issues with getting other collision algorithm's to work 100% of the time
3. TankGame Class – The main class responsible for the game loop, and drawing the graphics
 - Tries to delegate tasks to other Objects
 - Has more responsibilities than it really should
 - This led to refactoring of the main class in Rainbow Reef to try and distribute the excess

XVIII. Shell Class

Shell
<pre># xMax: x-coordinate # yMax: y-coordinate - damageVal: int - BASE_SHELL_SPEED\$: pixels/frame - BASE_DAMAGE\$: int - visible: boolean - collided: boolean</pre>
<pre>+ spawnShell(xloc: x-coord, yloc: y-coord, dir: degrees) + move() + draw(g: Graphics2D, obs: ImageObserver) + collision(otherObj: CollidableObj) - destroyShell()</pre>

Class Reasoning:

This class was specifically mentioned in the specifications of the Tank Game; some sort of Object needed to be created and able to be shot and inflict damage upon another Tank. The choice to have a fixed number of Shells, and a destroy/spawn shell method that recycles the shells, was so to reduce memory and computation overhead. Memory overhead from having to constantly produce Shells every time the user decided to shoot, and computational overhead from having to calculate the collision on a large number of traveling Shells. The constants for this class were created for ease of use when fine-tuning the user experience.

Fields:

protected int xMax

- The maximum value in the x-direction that the shell can travel
- Can be used to bound the distance a shell can travel
- Currently used to destroy the shell if it goes off-screen for any reason
 - o Since we give the Player limited shells to have active at once, it is important to try and ensure they are always accounted for

protected int yMax

- Similar to xMax, except this is the y-bound for the shell

private int damageVal

- The amount of damage a shell does
- Should not be modified from the outside, hence the private access modifier
- Created separately of the BASE_DAMAGE because of plans for damage multiplying power-ups
 - o This feature was never realized

private static final int BASE_SHELL_SPEED

- The default value for Shell speed
- Makes it easy to change the Shell's speed when fine-tuning the user experience
- Made a constant so that it cannot be tampered with by any Object

private static final int BASE_DAMAGE

- The default damage a Shell inflicts upon collision
- Makes it easy to change the Shell's damage values when fine-tuning the user experience
- Made a constant so that it cannot be tampered with by any Object

private boolean visible

- Boolean flag to determine if the Shell is visible (and will be drawn)
- Used in conjunction with recycling shells between active and used as mention in the Player class documentation
- All the unused Shells reside at location (0,0) when not active
- Private, and only set in the constructor and destroyShell()
 - o Ensures Objects can't find a way to make invisible Shells fireable

private boolean collided

- Boolean flag used to determine if a Shell should be put in the unused Shells list in the Player class
- Set when the Shell collides

Methods:

public void spawnShell(int xloc, int yloc, int dir)

- Spawns a Shell at the location specified by (xloc, yloc) with a given direction at its base speed
- Used when a Shell is fired
- Resets the shell to an active state
 - o Visible, Collidable, not Collided

public void move()

- Method used to move a visible Shell
- Inherited and overrides MoveableObj's abstract method
- Uses trigonometry to get the displacement in the x and y plane from the speed and angle of the Shell
- Also despawns the Shell if it has gone outside the bounds of the arena

```
// Assumes that non-visible shells should not be moving
if (this.visible)
{
    // Update shell position
    this.x += (int)(this.speed * Math.sin(Math.toRadians(this.angle)));
    this.y -= (int)(this.speed * Math.cos(Math.toRadians(this.angle)));

    // Check if it's still in the window
    if (this.x < 0 || (this.x > this.xMax) || this.y < 0 || (this.y > this.yMax))
        destroyShell();
}
```

Algorithm to calculate Shell displacement.

public void draw (Graphics2D g, ImageObserver obs)

- Inherited from GameObject and overridden
- Draws the image representing the Shell onto the Graphics2D Object
- Uses image rotation to make sure the image is pointing in the direction it is traveling

```
if(visible) {
    AffineTransform old = g.getTransform();
    g.rotate(Math.toRadians(this.angle), x: this.x + (this.width) / 2, y: this.y + (this.height) / 2);
    g.drawImage(this.img, this.x, this.y, obs);
    g.setTransform(old);
}
```

Calculation to find how the Shell image should be rotated

public void collision(CollidableObj otherObj)

- Inherited from CollidableObj, and overrides
- Extremely simple, if a Shell collides, it is destroyed

private void destroyShell()

- Private access modifier so that other Objects cannot arbitrarily destroy Shells
 - o Other than through calling it's collision() method
- Resets the Shell to it's position before being spawned

XIX. Tank Class

Tank
- prevX: x-coordinate - prevY: y-coordinate - health: int - moveLeft: boolean - moveRight: boolean - moveUp: boolean - moveDown: boolean - dead: boolean
+ repair() - checkDead() # fire(tankShell: Shell) + move() + draw(g: Graphics2D, obs: ImageObserver) + collision(otherObj: CollidableObj)

Class Reasoning:

It's not a Tank Game without a Tank, so including this class was a no-brainer. The decision to decouple the Tank from needing to know where the Shell it was firing was made so that we didn't have to go through the Tank to check for collisions on the active Shell. Decoupling it from the direct KeyEvents to move it was done with the idea in mind that the Player class should control the class, similar to how the user is controlling the game. So we

delegated that responsibility to the Player class instead. For collision, the Tank simply returns to its position prior to collision upon collision with a Wall or another Tank. This inflicts no damage, because being able to ram the opponent and shoot them for damage at the same time seemed to unbalanced, and would lead to users just rushing at each other while firing until one person lost.

Fields:

private int prevX

- The last x-coordinate of the Tank before it moved
- Used to handle tank collision
- Set to private because only the Tank should be changing this value

private int prevY

- Similar to prevX, but the y-coordinate instead

private int health

- The current health of the Tank
- Set to private so that this value can only be modified upon collision with a Shell
- When this is zero, the Tank dies, and it's owner loses the game

private boolean moveLeft

- Boolean used to indicate to the Tank that it should rotate to the left
- Private access to restrict variable access to setter methods

private boolean moveRight

- Boolean used to indicate to the Tank that it should rotate to the right
- Private access to restrict variable access to setter methods

private boolean moveUp

- Boolean used to indicate to the Tank that it should move forward in the angle it is facing
- Private access to restrict variable access to setter methods

private boolean moveDown

- Boolean used to indicate to the Tank that it should move backward, opposite the angle it is facing
- Private access to restrict variable access to setter methods
 - o The Player uses the setter to notify the Tank to move upon the appropriate KeyEvent

private boolean dead

- Flag used to determine if the Tank has run out of health
- Will be checked to determine the loser of a Tank Game match

Methods:

public void repair(int heal)

- Increases the Tanks health
- Used in conjunction with picking up the health power-up
 - o Not Implemented

private void checkDead()

- Method that sets the dead flag based on the current health of the Tank
- Made private to control the pathways to setting the dead flag to true
 - o Since this flag signals the end of the game

public boolean isDead()

- Public function to check the dead flag
- Acts as a wrapper for the checkDead() function for outside Objects

protected void fire(Shell tankShell)

- Spawns a the Shell passed to the Tank, and sets it to active
- Decouples the Tank from needing to know or having to keep track of the Shells it fires

public void move()

- Inherited and overridden from MoveableObj
- Moves the Tank if it is not dead
 - o Based on the boolean movement flags that are set
- Rotation is in 6 degree increments
- Forward and backward movement are dependent on its current angle
 - o Its previous location is also recorded in case of a collision from moving

public void draw(Graphics2D g, ImageObserver obs)

- Draws the Tank and rotates its image based on the angle it should be facing
 - o Rotation calculation is the same as Shell
- Inherited from and overrides method from MoveableObj

`public void collision(CollidableObj otherObj)`

- Inherited and overrides the method from `CollidableObj`
- Two cases
 - 1. It collides with a Shell
 - Health is reduced and check if the Tank is dead
 - 2. Collides with a Wall or Tank
 - Reset this Tank to its location prior to collision
 - Tried a different collision algorithm that caused the Tank to rebound off of Walls it collided with
 - The Tank would collide through Walls at specific angles
 - Abandoned that method for a simpler one

XX. TankGame

TankGame
<ul style="list-style-type: none"> - MAX_SHELLS_PER_PLAYERS: int - thread: Thread - bimg1: BufferedImage - bimg2: BufferedImage - windowSize: Dimension - g2: Graphics2D - move: int - real_w\$: pixels - real_h\$: pixels - window_w\$: pixels - window_h\$: pixels - subX1: x-coordinate - subY1: y-coordinate - subX2: x-coordinate - subY2: y-coordinate - walls: List of Wall - images: Hashmap<String, Image> - player1: Player - player2: Player
<ul style="list-style-type: none"> - drawBackGroundWithTileImage() - mapSetUp() - drawDemo() - checkForCollisions(p1: Player, p2: Player) - checkBound() - checkBound() + init() + paint() + start() + run() + main(args: Array of Strings) \$

Class Reasoning:

This class is the main class that controls the flow of the game. It renders the graphics, runs the main game loop, initializes the arena, and loads the media assets. It honestly has more responsibility than we intended it to have, but we did not end up finding the time to refactor it and delegate those responsibilities to new or existing classes.

XXI. Rainbow Reef Specific Classes

The following classes are unique to Rainbow Reef Game. These classes are strongly bound to the specifications of Rainbow Reef, and differ from the Tank Game because of design lessons learned in the process of working on the Tank Game.

These classes are:

1. Blocks Class – Class that represents the destructible blocks on the map
 - Extends CollidableObj
 - Decorates the map, and gets in the Players way of destroying enemies
2. Enemy Class – Represents the enemies to be destroyed on the map
 - Extends CollidableObj
 - These Enemy Objects must be cleared from the map to progress to the next level
3. RainbowReefLevel Class – A class to encapsulate the state of the GameObjects within a level
 - In charge of calculating the collision caused by the Slider and Star
 - Also responsible for drawing the GameObjects on the map
4. RainbowReefWorld Class – The main class for RainbowReefWorld
 - Responsible for running the main game loop
 - Responsible for drawing the background and other GUI elements
5. Slider Class – The class that represents the user controlled bar
 - Player controlled, but only moves parallel to the x-axis
6. Star Class – The class that represents the Object that bounces around the map
 - Collides with/breaks blocks and enemies

XXII. Blocks Class

Blocks
- health: int - /score: int - /dead: boolean
+ collision(otherObj: CollidableObj) - destroyBlock() + draw(g: Graphics2D, obs: ImageObserver)

Class Reasoning:

As with most of the other unique-to-one-game classes, this class came straight out of the requirements for Rainbow Reef. The decision to put in a point value, and have varying health values was also suggested while we were in class. The class itself is pretty simple, and has little behavior other than handling being collided with, and destroying itself when it has been collided with enough times.

Fields:

private int health

- The current health of the block, this is passed in as a parameter
- Private access so that health can only be reduced through specific controlled circumstances
 - o Collision with the Star in this case

private int score

- The score the block adds to the Player's upon being destroyed
- Set to 10 * the starting health of the block
- Private access so that once set, the score cannot be modified externally

private boolean dead

- A flag used to signal if the block is out of health and should be destroyed
- Set to private so that it cannot be tampered with externally
- This flag is also used to signal that this Block should be removed from the list of blocks in the level

Methods:

public void collision(CollidableObj otherObj)

- Overrides the abstract method from CollidableObj
- Handles collision with the Star
 - o There are only two MoveableObjs that can collide with other CollidableObjs
 - o The Slider only moves parallel to the x-axis, and the blocks are assumed above it
 - o Therefore the only thing that should collide with Blocks is the Star

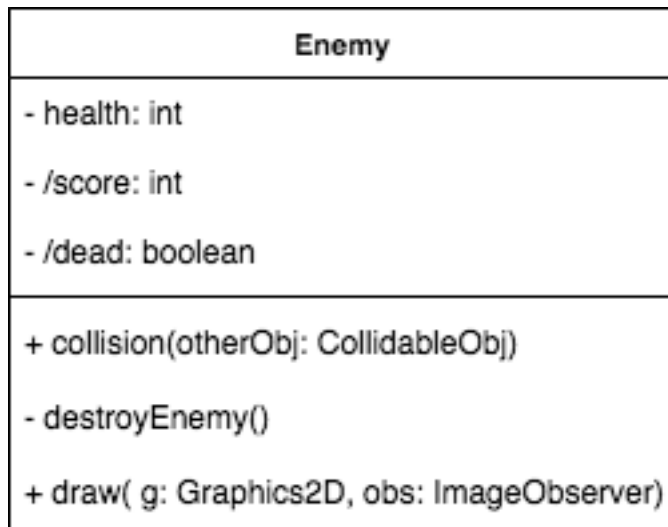
private void destroyBlock()

- Destroys a given block by setting it to not be visible and not collidable
- Private, so that only this Object has the ability to do this operation

public void draw(Graphics2D g, ImageObserver obs)

- Draws the block at its current location, with its image
- Overrides the abstract method from GameObject
- Public access so that another Object can tell the Block to draw itself

XXIII. Enemy Class



Class Reasoning:

Looking at the class diagram for this, I am astounded by how similarly this class looks to the Blocks class. If anything, we should have made a Destructable class that those two classes could subclass off of, but I digress. This class is similarly simple in behavior, though has room to do more. For instance, if we had more time we could have made a feature where enemies temporarily disappear or change to a state where they cannot be collided with since defeating every enemy is the main goal of each level in Rainbow Reef.

Fields:

private int health

- The health value of the enemy, when it becomes zero, the squid is destroyed
 - o Defaults to 1

private int score

- The score added to the Player's upon destroying this Enemy
 - o defaults to 100 * health

private boolean dead

- Flag to indicate that an Enemy is ready to be removed from the RainbowReefLevel's enemy list
 - o When the number of enemies in the list is zero, the level is complete

Methods:

public void collision(CollidableObj otherObj)

- Handles collision with this Object
 - o Only case considered is with the Star, for reasons mentioned in the Blocks class documentation
- Is the only method that invokes the destroyEnemy() method

private void destroyEnemy()

- Sets this Enemy to non-visible, non-collidable, and marks it for removal from the List of Enemies maintained by the level

public void draw(Graphics2D g, ImageObserver obs)

- Overrides the abstract method from GameObject
- Responsible for drawing the Enemy in the game

XXIV. RainbowReefLevel Class

RainbowReefLevel
- numEnemies: int - mapHeight: pixels - mapWidth: pixels - walls: List of Wall - blocks: List of Blocks - enemies: List of Enemies - collisionMap: 2D Array of CollidableObj
+ getLevelComplete() + drawLevel(g2: Graphics2D, obs: ImageObserver) + updateMap(player: Player) + collisionDetector(player: Player) - detectCollisions(collider: MoveableObj) - makeLevel(resources: HashMap<String, Image>)

Class Reasoning:

These class was designed to alleviate and distribute some of the responsibilities of the main class, as was originally planned to be done in the Tank Game.

Fields:

private int numEnemies

private int mapHeight

private int mapWidth

private ArrayList<Wall> walls

```
private ArrayList<Blocks> blocks
```

```
private ArrayList<Enemy> enemies
```

```
private CollidableObj[][] collisionMap
```

Methods:

```
public boolean getLevelComplete()
```

```
public void drawLevel(Graphics2D g2, ImageObserver obs)
```

```
public void updateMap(Player player)
```

```
public void collisionDetector(Player player)
```

```
private void detectCollisions(MoveableObj collider)
```

```
private void makeLevel(HashMap<String, Image> resources)
```

XXV. RainbowReefWorld Class

RainbowReefWorld
<ul style="list-style-type: none"> - WINDOW_W\$: pixels - WINDOW_H\$: pixels - thread: Thread - bimg1: BufferedImage - windowSize: Dimension - g2: Graphics2D - move: int - images: Hashmap<String, Image> - sounds: Hashmap<String, SoundPlayer> - player1: Player - getHelp: boolean - status: boolean
<ul style="list-style-type: none"> - drawBackGroundWithTileImage() + drawDemo() + init() + paint() + start() + run() + main(args: Array of Strings) \$

Class Reasoning:

Similar to TankWorld, this class was the main class that controlled the game loop. Having learned some lessons from Tank Game, we delegated the responsibility of maintaining the GameObjects on the map away from this main class and gave it to RainbowReefLevel. This class still draws the background and other GUI elements, which could possibly have been delegated to another class as well, enabling this class to just worry about running the game loop.

XXVI. Slider Class

Slider
- moveLeft: boolean - moveRight: boolean
+ collision(otherObj: CollidableObj) + move() + draw(g: Graphics2D, obs: ImageObserver)

Class Reasoning:

This class was also taken straight out of the requirements for the Rainbow Reef game. It is the bar that the Star rebounds off of to hit the blocks and enemies, and it controlled by the key inputs from the user, albeit indirectly. We went with a similar design style for handling user movement as we did with the Tank, where it's the Player Object that actually notifies the Slider when to move, instead of the KeyEvents directly.

Fields:

private boolean moveLeft

- Flag to indicate to the Slider that it should move left
- Should only be set by the Player

private boolean moveRight

- Flag to indicate to the Slider that it should move right

Methods:

public void collision(CollidableObj otherObj)

- Overrides the abstract method from CollidableObj
- Only handles the case of colliding with the Wall
 - It's assumed the Blocks will always be above the Slider
 - The Slider has no behavior when the Star rebounds off of it
- Rebounds off the Wall if it collides with it

```

if (otherObj instanceof Wall) {
    if (moveLeft) {
        setX(x + (2 * xSpeed));
    }

    if (moveRight) {
        setX(x - (2 * xSpeed));
    }
}

```

Formula used to determine how far to rebound off the Wall

public void move()

- Overrides the abstract method from MoveableObj
- Moves the Slider either left or right
- Also records the Slider's previous position

public void draw(Graphics2D g, ImageObserver obs)

- Overrides the abstract method from GameObject
- Draws itself based on its current coordinates

XXVII. Star Class

Star
- initialX: int - initialY: int - initialXspd: pixels/frame - initialYspd: pixels/frame
+ collision(otherObj: CollidableObj) + move() + resetStar() + draw(g: Graphics2D, obs: ImageObserver) - whichSideForCollision(otherObj: CollidableObj): side

Class Reasoning:

This class was also pulled straight out of the requirements for the Rainbow Reef game, which required a block (star) that would be used to hit the Blocks for points, and destroy enemies to progress in the game. Initial values given to the Star upon construction are recorded so that when the Star falls below the map, it can be respawned at its original location. The collision algorithm for the Star is more complex than any of the other MoveableObj between both games, because the side of the block it strikes and the direction the Star is traveling, will affect how the Star rebounds. For that reason, the side of collision needs to be found to determine the Star's behavior on collision.

Fields:

private int initialX

- The initial x value passed into the Star's constructor
- Recorded for when the Star has to be reset
- Private and has no setters/getters, because this should not be modified

private int initialY

- The initial y value passed into the Star's constructor
- Recorded for when the Star has to be reset
- Private and has no setters/getters, because this should not be modified

private int initialXspd

- The initial x component of the speed passed into the Star's constructor
- Recorded for when the Star has to be reset
- Private and has no setters/getters, because this should not be modified

private int initialYspd

- The initial y component of the speed passed into the Star's constructor
- Recorded for when the Star has to be reset
- Private and has no setters/getters, because this should not be modified

Methods:

```
public void collision(CollidableObj otherObj)
```

- Overrides the abstract method from CollidableObj
- Checks what side of the CollidableObj the Star collided with
 - This will determine how it rebounds
 - Switches the direction of the speed component related to the side of the Object the Star collided with, and moves it to it's pre-collision position
 - Special case for collisions with the Slider
 - If the Slider was moving when the Star collided, a portion of the Slider's speed is added to the Star's x component speed

```
public void move()
```

- Overrides the abstract method from GameObject
- Moves the Star based on its x and y speed components
- Also records its previous position

```
public void resetStar()
```

- Resets the star back to its original position
- Made public so that the Player can reset the Star once it notices it has fallen below the map

```
public void draw(Graphics2D g, ImageObserver obs)
```

- Overrides the method from GameObject
- Draws the Star in the game

private int whichSideForCollision(CollidableObj, otherObj)

- Determines the side of the CollidableObj that the Star hit and returns it
- Uses rectangle intersection to determine this
 - o Draw thin rectangles on each side, and checks each one for collision iteratively
 - Currently bugged, X-axis collisions don't work properly
 - Could be because the top and bottom sides are checked first?
 - I think the order of checking them needs to be refined to not be biased for which sides are checked first
 - Or possibly, the rectangles are too thick

```

/*
 * Check which side of the other collidable object the star collided with
 * done by checking coordinates of the star relative to the object it collided with
 *
 * Side translation:
 * 0 - bottom
 * 1 - top
 * 2 - right side
 * 3 - left side
 *
 * Using 1/2 of the width, height was chosen somewhat arbitrarily
 *
 * Assumptions:
 * 1. The star collided with the other object
 * 2. The object can be well represented by a rectangle
 */
private int whichSideForCollision(CollidableObj otherObj) {

    boolean collisionFound = false;
    int side = -1;
    int collidableX, collidableY, collidableWidth, collidableHeight;
    Rectangle starArea = null;
    Rectangle sideCollisionBox = null;

    starArea = new Rectangle(this.getX(), this.getY(), this.getWidth(), this.getHeight());
    collidableX = otherObj.getX();
    collidableY = otherObj.getY();
    collidableWidth = otherObj.getWidth();
    collidableHeight = otherObj.getHeight();

    // loop until a collision is found
    for (int i = 0; i < 4 && !collisionFound; i++) {

        // Check each side of the other collidable object for collisions until one it is found
        switch (i) {
            case 0: sideCollisionBox = new Rectangle(collidableX, y: collidableY + collidableHeight - (collidableHeight / 2),
                collidableWidth, height: collidableHeight / 2);
                break;
            case 1: sideCollisionBox = new Rectangle(collidableX, collidableY, collidableWidth, height: collidableHeight / 2);
                break;
            case 2: sideCollisionBox = new Rectangle(x: collidableX + collidableWidth - (collidableWidth / 2), collidableY,
                width: collidableWidth / 2, collidableHeight);
                break;
            case 3: sideCollisionBox = new Rectangle(collidableX, collidableY, width: collidableWidth / 2, collidableHeight);
                break;
        }

        if (starArea.intersects(sideCollisionBox)) {
            side = i;
            collisionFound = true;
        }
    }

    return side;
}

```

XXVIII. Self-Reflection: Software Design and Development

Arnold Yu:

In both projects, I participated on class designs with my partner. I specifically focused on setting up GUI, map design, key control, and sound player. I felt I did well on both GUI and key control, and understand how java library will help with varies graphic design. I also felt I could do better on map design to make the second game more interesting by doing slightly different implementations. I developed a better software design skill and strengthened understanding of object-oriented-programming. I realized a better design can make coding easier and faster.

Challenges:

1. First challenge was the key control associated with images. When I was testing whether the TankWorld works properly and Tank object appeared on the screen. I specifically set two Tank images differently. Both of them shared same angle parameter for Tank object, I also passed different angle to each object and they moved differently. I thought it is the angle and key control method did not work properly. After spent some time, I found out both images need to have same initial angle such as degree zero, then after I passed different angle parameters, it moved correctly.
2. Second challenge I encountered was the split screens and mini map. In first version, my mini map displayed the split screens, which replaced the left upper corner of the map. I overcome the challenge by switching orders of the code. I found out every frame, it kept covering the old images. After creating new object and replaced in the right order. It worked fine.

Ian Dennis:

My role in the project was to work on the calculations necessary for collision and movement, and analyzing our design to see if we could better encapsulate Objects and delegate tasks such that there was not unnecessary coupling between classes. My biggest takeaway from this project was knowledge on Java GUI, which was a topic that has always seemed daunting to me because it seems hard to pick a point to start learning it (given how massive the libraries for it are). I have found software design to be the most interesting part of programming for a while now, and this project just affirmed my enjoyment with it. To me its more fun to come up with the design than it is to implement it, but if you never implement it you never learn the flaws in the design. All in all, I enjoyed working through some of the more frustrating moments our team had with this project. Nothing feels quite as good as debugging a (seemingly) difficult bug, although I wish I had spent more time on this project to figure out the collision bug we were having in the Rainbow Reef Game. I guess I'll have to come to the bottom of it in my free time.

XXIX. Conclusion

This was a fruitful experience for our team, as we delved into areas of Java we had never before used, as well as got to exercise the object oriented design theory that we were learning in class. Reusability really was the key with this project, and a good design went a long way into incorporating features without having to modify a large swath of classes (due to unnecessary dependencies). We were both challenged by this project, both by our unfamiliarity with many of the libraries that ended up being the bread-and-butter for our project, as well as maintaining coordination to produce as few merge conflicts as possible when working on features.

Communication also went a long way with this project. The conversations we had over our game designs were fun, and it turns out that diagramming goes a long way in bridging the gap between understanding what somebody is talking about. It was nice to have another mind to bounce ideas off of, or to be shown different perspectives that we would not otherwise notice.

I think we both enjoyed working together on this project, and look forward to collaborating on future software projects with others given the skills we gleaned from this experience.