`

# Free the Bob - Design Document

Jonathan Lucuix-André

Mugisha Kakou

# Contents

# I. Project Overview

## I.1 Story

In the world of *Free the Bob,* video games have harnessed the power of virtual reality. Gamers worldwide compete professionally in video games, rising the ranks of professional leaderboards.

Amidst this virtual reality craze, you play Bob, an avid gamer whose only aspiration is to climb the competitive ladders and compete in the professional leagues. Indeed, only then would life truly become an enthralling adventure.

On March 24, 2024, a survival game named *Survivor* is released. It becomes a success overnight, selling two million copies on its first day in retail. Among its innovative gameplay mechanics, *Survivor* claims to revolutionize the way virtual reality is utilized in the video game space.

Bob, one of the lucky owners of the game, grabs a copy on release day and makes his way home. Anxious to play the game, he puts on his virtual reality headset, inserts the disc into his gaming machine, and closes his eyes. Almost instantaneously, Bob gets sucked into an alternate reality, waking up in a post-apocalyptic forest.

As he spawns in this new world, Bob is greeted by a flying companion named *Levi*, tasked to guide Bob through the game's mechanics. *Levi* teaches him the basic survival skills he needs to survive in the forest.

Before leaving Bob, *Levi* informs him of one life-threatening caveat: he can't leave the game until he beats it. Startled by this truth, Bob demands to know how to get out of the game. *Levi* explains that the only way to escape is to build a teleportation device. However, even he doesn't know how to craft it.

With these teachings, Bob sets out to venture into the frozen recesses of an unknown world. Perfused with snow, and filled with the flesh-eating undead, the video game world is endless, spanning thousands of kilometers in all directions. Thus, using the knowledge Bob recollects from chemistry class, he must collect resources, craft consumables, and build the weapons he'll need to fend off the zombies of the forest.

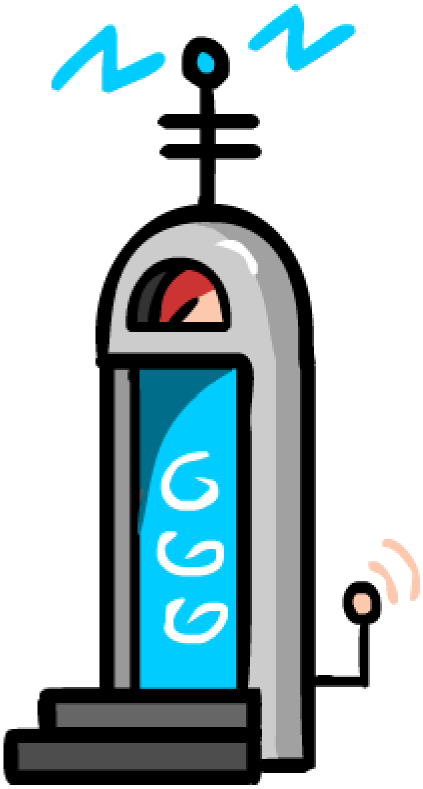


**Figure 1 : Visual representation of the teleporter**

# I.2 Project Description

## I.2.1 System Objectives

The objective of the game is to test the player’s knowledge of chemistry. Only with an adequate understanding of chemistry can the player survive the forest. Therefore, the player must be able to reason out certain chemical formulas, allowing him or her to craft basic survival necessities, along with the weapons he or she will need to fight the flesh-eating undead.

Furthermore, given that the project is a video game, the objective of the system is also to entertain the player. As a matter of fact, a great deal of work has been put to make the game as fun as possible. To do this, the project was streamlined, and effort was put into the game to simplify the controls, and make them accessible and intuitive for everyday phone users.

### I.2.2 Project Constraints & Scope

There are numerous constraints that were present when we started developing the game. For one, the game was built to be single-player. Furthermore, it wan't meant to use the Internet, as it was built to be played offline.

Another constraint present at the beginning of development was that we chose to target Android versions 2.3.4 and above. This limitation was imposed to ensure that every device would be capable of playing the game at a decent frame rate. However, for this reason, the scope of the project is rather large, as a large portion of Android phone owners run an Android version above version 2.3.4.

Furthermore, another constraint we imposed on the project was our use of the *OpenGL ES 1.0* graphics library instead of the version 2.0 and above. We chose to develop for an older version of the library in order to maintain compatibility with older devices, therefore augmenting the scope of our project. If we chose a more recent version of the library, we would alienate a great deal of potential Android users. In fact, not all phones support *OpenGL ES 2.0* and above. In addition, we set this constraint because we did not need any functionalities introduced in later versions of *OpenGL ES.*

However, from a user's point of view, the biggest constraint of the game is that the player must have knowledge about the formation of certain resources using chemical formulas. This is a constraint we decided to at the beginning of the project. In fact, if the player did not know the ingredients to make gunpowder, we wanted him to have a hard time finding the right ingredients to craft such an item. We tried to limit this constraint by introducing the Survival Guide, which gives pointers on how to combine certain items to create new ones. However, players may not feel inclined to immediately traverse the Survival Guide in search of item recipes. This allows the game to feel like a more genuine survival game, where the user needs to figure out certain mechanics of the game on his or her own.

Additionally, another constraint we imposed at the beginning of development was the fact that the game used the concept of layers, a new concept that has never been implemented in 2d games. In fact, in a traditional 2.5D game, the player usually has the ability to move up, down, left or right. However, in this game, the player can only move left or right, but can move up or down between what are called "layers". This will present a barrier of entry for players. However, we wanted to create a project with innovative mechanics that could define new types of 2d movement.

On a separate note, near the end of development, there were several constraints we had to impose on the project. These constraints were made in order to meet the project's deadline. For instance, there are a very restricted amount of items that can be crafted using chemistry. As previously explained, we needed this limitation in order to meet the deadline for the project. Drawing the art for each item, along with programming unique functionalities for each new item is a time-consuming process. Therefore, near the end of development, we chose to keep our list of available items relatively small, instead of expanding on it. This reduced the scope of our project, as it ultimately made it less feature-rich.

Near the end of development, we also had to impose another constraint on our project. In fact, we chose not to implement the optional features. This constraint was created due to the fact that we did not have enough time to implement these additional features. Furthermore, given that we wanted to play-test the game fully before handing it in, we chose to spend our final week bug-fixing. Once again this reduced the scope of our project. In fact, as with the constraint stated in the paragraph above, this ultimately made our system less content-rich.

### I.2.3 Tools & Methodologies

The game uses the Android Developer Tools in order to work on Android devices. All user interfaces and controls are designed with a touch screen in mind, allowing for a larger potential user base, and a lower barrier of entry.

Furthermore, the application is built off the *LibGDX* framework, which allows the game to utilize the graphical capabilities of the *OpenGL ES* library. The framework also uses scene2d, a 2d scene graph which allows the creation of graphical user interfaces. This scene graph gives us access to pre-defined GUI widgets. Its functionality closely resembles that of Java's *Swing* library.

In addition, given that the *LibGDX* framework is used, the application can run on the PC platform for debugging purposes. As such, an Android phone is not required to test minor changes in code, effectively making the debugging process faster, and thus more efficient.

Finally, in terms of tools, we also used the *Spine* animation engine. This is an external program which allows us to import images and hand-animate them. A run-time library inside *LibGDX* is used to import these animations and run them in the game.

On a separate note, the methodologies we adopt for the creation of our game are closely tied to our decision to build the application for Android devices. In fact, the need to optimize our algorithms to the best of our abilities is crucial. In truth, only with a great deal of optimization can our application run smoothly on low-end Android devices. This is important, as these low-end devices still exist on the market. Additionally, as detailed in later sections, most of our algorithms run efficiently, therefore allowing the procedural generation of our world and the game logic itself to run quickly.

Furthermore, one of our biggest methodologies was the need to intelligently control garbage collection. In fact, on Android devices, Java is known to activate its garbage collector often. In addition, it also does it very inefficiently. Therefore, object instantiation and destruction needs to be minimized in order to prevent garbage collection from activating when the player is traversing the world and playing the game. For this reason, our biggest optimization is our use of *pools,* which allows the program to create a great number of objects when it launches, and to hold its object references for the game's entire lifecycle. Using the same concept of *pools*, object instantiations are reused whenever an already-created object is no longer needed. Essentially, we recycle objects for as long as possible to avoid running out of memory.

### I.2.4 Critical Project Events

The most critical project event which occurred during development was our need to add certain features to our application. For instance, when showed the game to our instructor, we were given suggestions to add music and sound into the game. This occurred on week 13, two weeks before the final source code was due. However, given that most of our features were already implemented, we decided to acknowledge the suggestion and implement audio into the game. However, the task was larger than expected, as it necessitated the implementation of three new classes: the *SoundManager,* the *MusicManager*, and the *SoundListener*. The first of the three manages the sound effects, and the volume at which they plays. Similarly, the *MusicListener* does the same thing for music playback. Note that the sound and music were separated from a programmatical perspective. We chose this approach because, if we ever wanted to add an *Options Menu*, the user would want to change the volume of sound and music independently. Thus, we chose to manage sound effects and music separately. Finally, the last of the three classes listed above delegates events to the *GameScreen* whenever the application needs to play a sound.

However, implementing the audio feature imposed a constraint to our project not explained in *Section I.2.2 Project Constraints & Scope*: we had to find sound effects, but we couldn't create them. In fact, we lacked experience in creating sound and music. Therefore, we turned towards *FreeSound.org,* which allowed us to choose from a vast library of free and royalty free sounds. We imported these sounds into our game, effectively adding audio to the project.

On a different note, the second critical project event we encountered was our need to implement the *GameSelectScreen*. Again, due to the teacher's feedback, we decided to include a new screen into our game which would allow the user to either load, save or continue his most recent profile. This required a few additions in code, but allowed us to improve the user's experience significantly. In fact, in the testing phase of the project, when the users navigated the *GameSelectScreen,* users were able to traverse the game's menus very easily. Furthermore, due to this added screen, users had no difficulty in grasping the concept of profiles and worlds.

Furthermore, we added this additional feature two weeks before the project was due for submission, on week 13. Nonetheless, this additional feature did not require a large amount of extra code. In fact, whereas adding sound and music required three new classes, the *GameSelectScreen* only required one class: the *GameSelectScreen*. This new class simply allows the user to select to either load, continue or create a new profile.

Once the *GameSelectScreen* was implemented on week 14, we decided to take away the constraint which forced users to have a maximum of three profiles. In fact, we wanted to allow the player to have access to an unlimited amount of profiles, as long as the device does not run out of physical hard drive space. This change did not require many alterations of code. In fact, the profile selection list already present in the *WorldSelectMenu* was simply placed into a scroll pane, so that the user could scroll down his list of profiles. Furthermore, we also replaced the *maxProfiles:int* member variable inside the *ProfileManager* with the *numProfiles:int* variable. This new variable keepstrack of the amount of profiles held by the user. Therefore, given that the necessary modifications were small, we managed to implement them on week 14.

# II. Design

## II.1 Algorithms

Important Notes:

1. Only the most important and complex algorithms of the project are detailed in this section. Explanations of other algorithms present in the project are given in *section II.3.1 Classes.*
2. The section which requires the analysis and understanding of the system's problems is merged with the section on algorithms. This allows the problems to be defined at the same time that the algorithms our presented. In this manner, we navigate and describe the problem at the same time we introduce the solution.

### II.1.1 Procedural Terrain Generation & Code Structure

Programmatically, the forest in *Free the Bob* needed to be procedurally generated. Furthermore, it had to be infinite. To create an efficient solution to this problem, the world was built using a “terrain level”. This level consists of a fixed-size matrix, where every element consists of a “terrain layer”. Each layer is represented as either a constant, linear, or cosine function which the player can walk on. From a programmatic perspective, the player is always at the center of the terrain level. When the player moves from one layer to another, the level adjusts its matrix and moves the layers so that the player always resides in the center-most layer. Once the layers are re-positioned inside the matrix, their row and columns relative to the world also change. Using the layer's cell coordinates, along with a random number generator, the layer's terrain is built. In fact, the geometry of each layer is rebuilt according to the world seed and the layer's cell coordinate relative to the world. In fact, to calculate the seed needed to generate the layer's geometry, the layer's column is multiplied by the world seed. This seed is then placed into a Random object, which is then used to determine the geometrical properties of the layer. As mentioned above, these geometric properties simply denote a constant, linear, or cosine function. The generation of such an elementary function is defined as follows. First, a random number generator provides number between 0 and 1.0, exclusive. If the number is greater than 0.7, the layer will be a cosine function. If the number is above 0.4, the layer will be a linear function. If the number is below these two values, the layer becomes a constant function.

If the layer is a cosine function, the random number generator then produces a new number between 0 and 1.0. This random number is chosen as the amplitude of the function. Similarly, if the layer is represented as a linear function, another random number is generated. This random value is chosen to be the slope of the function. However, if the layer is a constant function, no extra computations or random numbers are required. This operation of defining the elementary function for a *TerrainLayer* takes *O(1)* time.

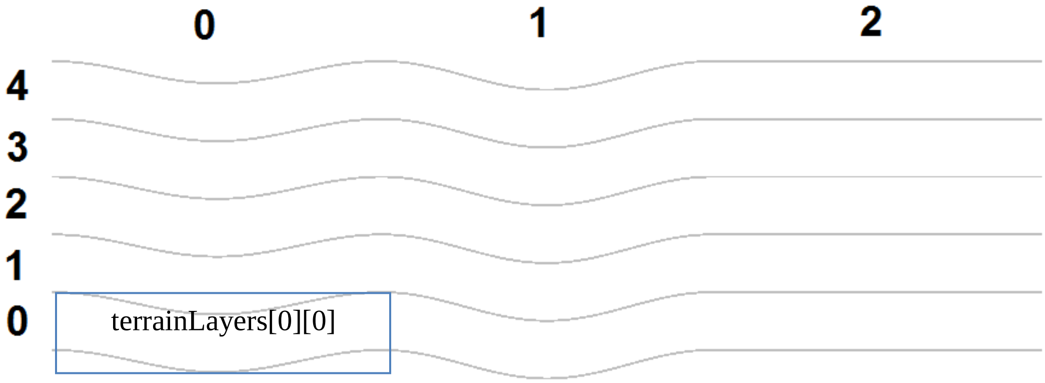


**Figure 2: Visual representation of the *TerrainLayer* matrix**

Conversely, we needed to find a method in which to populate the world with objects. Furthermore, this process had to be done efficiently. Given that the world is infinite, we needed to find an intelligent algorithm which placed objects on each *TerrainLayer* efficiently. To achieve this goal, we again use the concept of seeds and random number generators. In reality, to place objects on each layer, a random number generator seed is computed. To calculate said seed, the terrain layer's row and column are added. The world seed is then added to this result. The resulting number is called the object seed. Since every layer has unique cell coordinate in the world, each layer has a unique seed. This unique seed is used to randomly generate numbers which will, in turn, be used to create a series of objects for the layer. Thus, each layer has a different set of objects. The object-placement algorithm then iterates from the left-most x-position of the layer to the right-most x-position of the layer. Using the pre-computed seed, a math.util.Random object generates a random number. If, for instance, the randomly-generated number is below *0.5*, a tree is placed in the x-position the loop is iterating through. If the number is above *0.6*, for example, a box is placed there. This object-placement algorithm thus takes *O(n)* time, as it uses a *for* loop in order to cycle through the left-most position to the right-most position of a layer.

A *TerrainLayer* thus holds an array of *GameObjects*. This array holds the GameObjects that are rendered in the given layer. Each layer also has a row and a column in the world. The column determines the layer's geometry, while its row determines the objects placed on the layer.

On a separate note, we wanted to create an understandable relationship and hierarchy between each class in the program. In case we wanted to introduce new functionalities or features, we wanted the code to be easy to understand and to build on. Therefore, part of our solution to this problem is the *World* class, which holds a reference to the terrain level, along with the *GameObjects* contained in the game. In addition, it also controls all game logic. Thus, the class can be seen as a container or a master class for most of the game's entities and gameplay mechanics. This class allowed the program to sustain an understandable code structure.

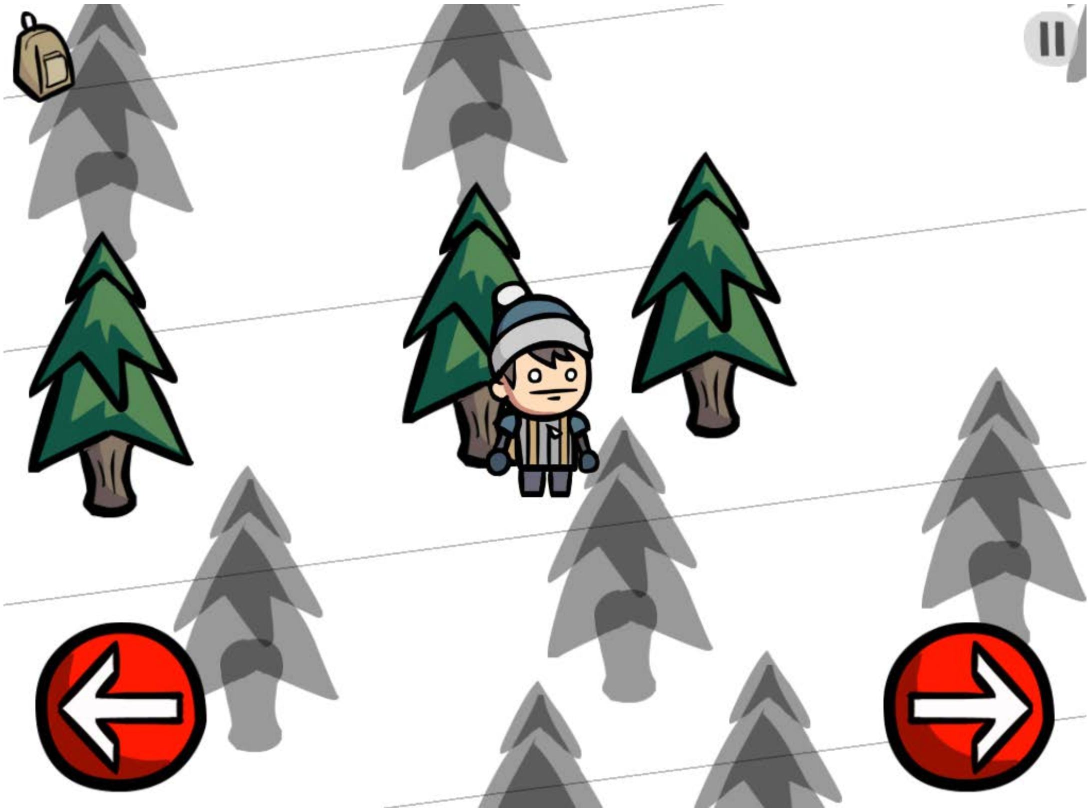


**Figure 3: The player exploring the procedurally-generated world**

### II.1.2 Input Handling & GameObject Updating

We wanted to create an efficient process to handle input. In fact, we did not want any noticeable frame rate inconsistencies to appear when the user tapped on an object. Therefore, we wanted to use an efficient method which would filter taps and detect which objects said taps touched. For this reason, we used the following algorithm in order to process input. First, when the user taps the screen, the finger tap is delegated to the *InputManager* class. Then, the *InputManager*, which holds an instance of the camera used to render the world, can convert the given tap coordinates into world coordinates. The position of the tap is then delegated to the *World*, which can subsequently decide how to process the touch.

To detect if a GameObject in the world is touched, the *World* instance first takes the *TerrainLevel* instance, and grabs all of the *TerrainLayers* in the same row as the player. Each of the *TerrainLayers' GameObject* arrays' are taken and added to a more global *GameObject* array. This array is stored inside the *TerrainLevel* class. The *World* then receives this array, cycles through it in *O(n)* time, and checks each game object collider to see if the given tap coordinates touch the game object. Thus, this algorithm presents an efficient solution to the problem of input handling.

To update the game objects and their logic, a similar procedure was used. In truth, every game tick, the aforementioned *GameObject* array is passed to the *World* so that every entity in the world can be updated. Thus, the algorithm used to update *GameObjects* works as follows. First, the *World* class iterates through each *GameObject* in the aforementioned array in *O(n)* time. Then, each *GameObject's update()* method is called.

*Note on Bounding Boxes:*

Each GameObject in the world holds a *Collider* instance. This collider is used to dictate the physical dimensions of an object, and to determine whether two objects ever collide. All of the GameObjects in the game use bounding boxes denoted by the Rectangle class. This class contains the methods intersects(Rectangle) and intersects(Vector2). These methods simply check whether or not a rectangle or a point intersects with another rectangle or another point in space. These intersection algorithm consists of simple mathematical calculation which take *O(1)* time.

II.1.3 Camera Culling

As with the previous algorithms, we wanted rendering to be optimized in our game. In fact, we knew that image drawing would be one of the most resource-heavy operations in the program. Therefore, as a solution to our problem, a technique called camera culling was implemented to optimize object rendering. To explain the algorithm, say that a tree wants to be drawn on-screen. First, the tree's collider is checked to be inside or outside the world's camera. If the object happens to be outside the camera, it is not rendered by *OpenGL*. Conversely, if it is viewable by the camera, it is drawn to the screen.

Internally, this algorithm is implemented inside the *GameObjectRenderer* class. Inside the *render()* method, each GameObject instance in the world is first tested to be inside the camera. This is done using the *Collider.insideCamera(OrthographicCamera):boolean* method. If it returns *true*, the *GameObject* is inside the camera's visible region, and is thus drawn to the screen. Programmatically, the *Collider.insideCamera(...):boolean* method takes in an

*OrthographicCamera* instance as an argument. The method which checks if the *GameObject's* rectangle collider intersects with the camera. This is tested using the camera and the collider's position and size and simple mathematical calculations.

### II.1.4 Heads-Up Displays

Furthremore, we wanted to introduce a well-structured way of creating GUIs. In fact, we knew that we would have many GUIs in our game, and thus their creation needed to be easy. To render a heads-up display for the game, subclasses are derived from the *Hud* class. This class holds a *Stage*, a *LibGDX* object used to draw 2d widgets to the screen. Each *Hud* subclass holds buttons and widgets that will be drawn to the screen using this stage. Therefore, using thsi code structure, we found a solution to our need to make the creation of in-game GUIs simple.

### II.1.5 Zombie AI

Given that our game contained zombies, we needed to control them using an efficient algorithm. In point of fact, if the algorithm was innefficient, the game would be unable to update a large number of zombies at the same time. Thus, we avoided this limitation using efficient artificial intelligence. When a zombie is walking around in the world, it simply walks back and forth between the beginning and the end of the layer to which it belongs.

```
//Pseudo code
if(zombie.x > layer.rightPoint.x)
    zombie.direction = Direction.LEFT;
else if(zombie.x < layer.leftPoint.x)
    zombie.direction = Direction.RIGHT;
```

Conversely, when the zombie is fighting against the player, can make several decisions. First, he can charge to towards the player. Once he reaches the player, he will deal damage him. Alternatively, the zombie can shoot an earthquake at the player. Lastly, the zombie can retreat back to his original position.

The zombies' decision-making algorithm is detailed in the pseudo-code below.

```
//Pseudo code
if(timeSinceLastAttack < 1 second)
        return;
int choice = Math.random();
if(choice > 0.5)        //50% chance of charging at the player
{
        zombie.state = State.CHARGE;
        zombie.direction = Direction.LEFT;  //The player is always to the left of the
                                             //zombie
}
else
{
        zombie.state = State.SMASH;
}


// If the zombie has collided with the player, deal damage to him and retreat back to
// original position
if(zombie.collider.intersects(player.collider))
{
        zombie.dealDamage (player);
        zombie.state = State.WALK;  //Walk back to original position
        zombie.direction = Direction.RIGHT;
}
```

```
// If the zombie is past the left bounds of the level, make him teleport to the other side of
// the level to walk back to his original position.

if (zombie.x < level.x)

{

        zombie.x = level.rightPoint.x;

        zombie.state = State.WALK;

        zombie.direction = Direction.LEFT;

}
```

### II.2.6 Item Crafting

The way the item crafting system works is with *Item* instances. Each *Item* instance represents a resource that the player can use to craft another item. Inside the crafting menu, say that the player mixes items *a* and *b*. These items will be placed inside a list, which is sent to the *CraftingManager* to test if the list of items corresponds to a possible combination.

```
// Pseudo code

// (CraftingManager method which checks if a and b form a crafting combination)

if(a instanceof Wood.class && a.quantity == 10 && b instanceof Iron.class &&
b.quantity == 5)

{

        //Return Axe.class, so that the user can craft an axe

}

else if(a instanceof Iron.class && a.quantity == 10

        && a instanceof Wood.class && a.quantity == 15)

{

        //Return Rifle.class, so that the user can craft an axe

}

...

...
```

The algorithm uses a brute force approach. In fact, the items that the player is trying to mix are checked with each possible combination until an appropriate one is found. This approach is efficient, as there are not many possible item combinations in the world. This algorithm has *O(1)* time complexity, and is thus very efficient.

## II.2 GUI

*Splash Screen:*

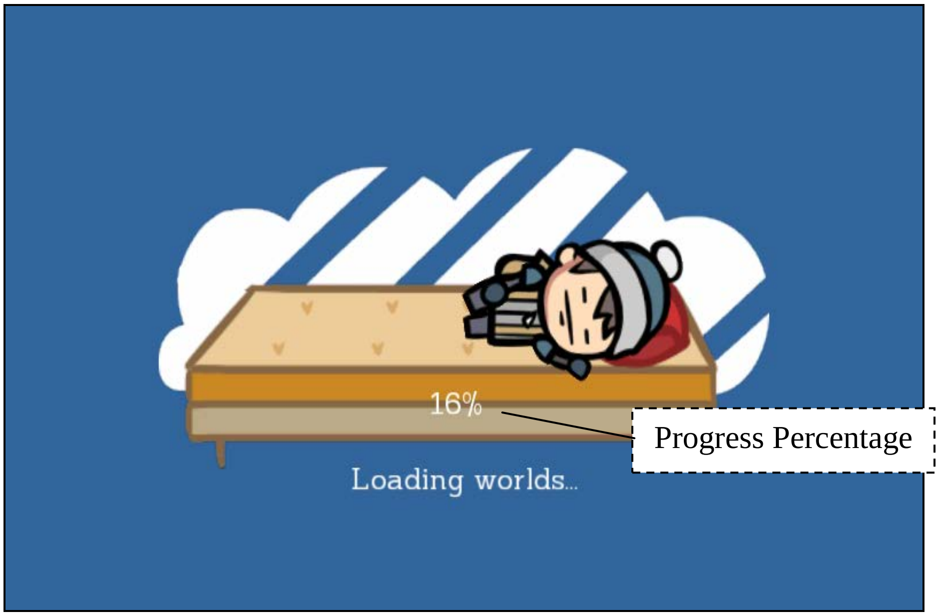
The splash screen is displayed once when the game is started. It is shown for approximately one and a half seconds to inform him about the creators of the project. It lasts only for a second and a half to avoid boring the player.

**Figure 4: Visual representation of the splash screen, presenting the creators of the game**

*Loading Screen:*

The loading screen appears after the splash screen when the game starts. It displays a progress bar, informing the player on how much time is left before the game is loaded. This prevents the user from believing that the game is frozen.

(See figure on following page)

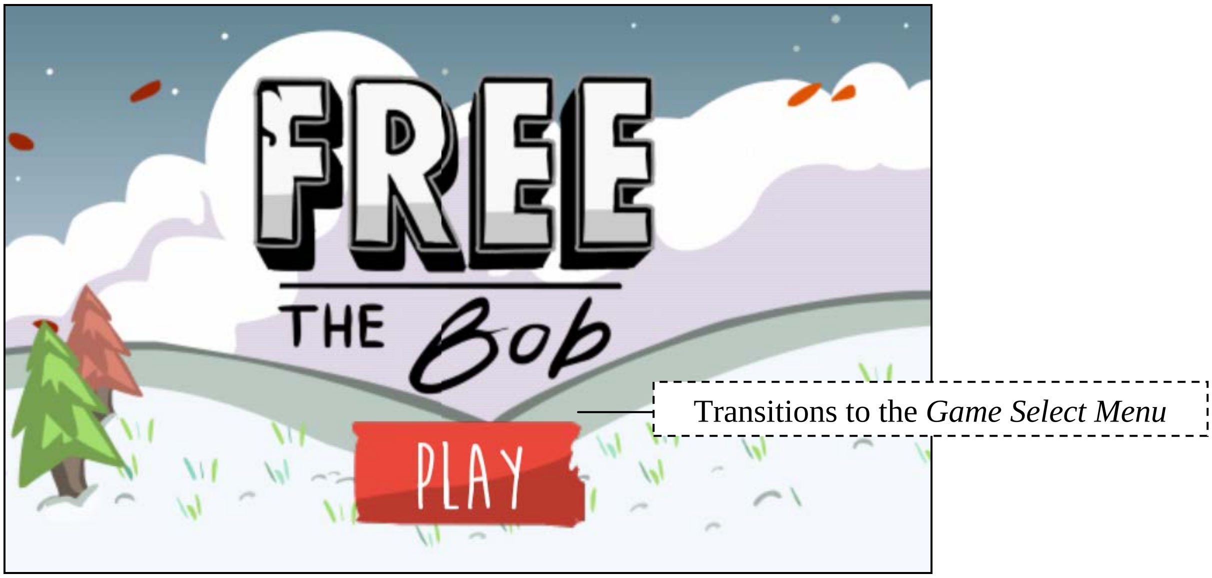


Progress Percentage

**Figure 5: Visual representation of the loading screen**

*Main Menu:*

This is the menu displayed once the game finishes loading. It prompts the user to start playing the game.



Transitions to the *Game Select Menu*

**Figure 6 : Visual representation of the main menu once the game has finished loading**

*Game Select:*

The game selection screen allows the player to either continue from his last-saved profile, create a new world, or load an old profile.

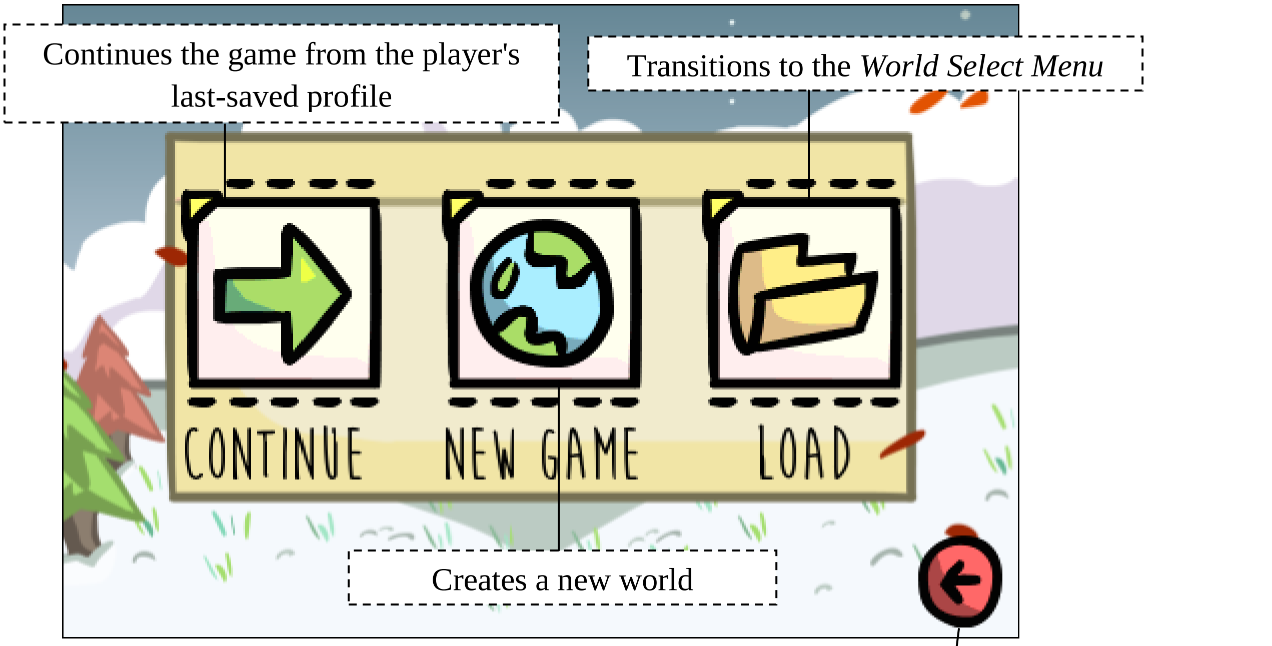
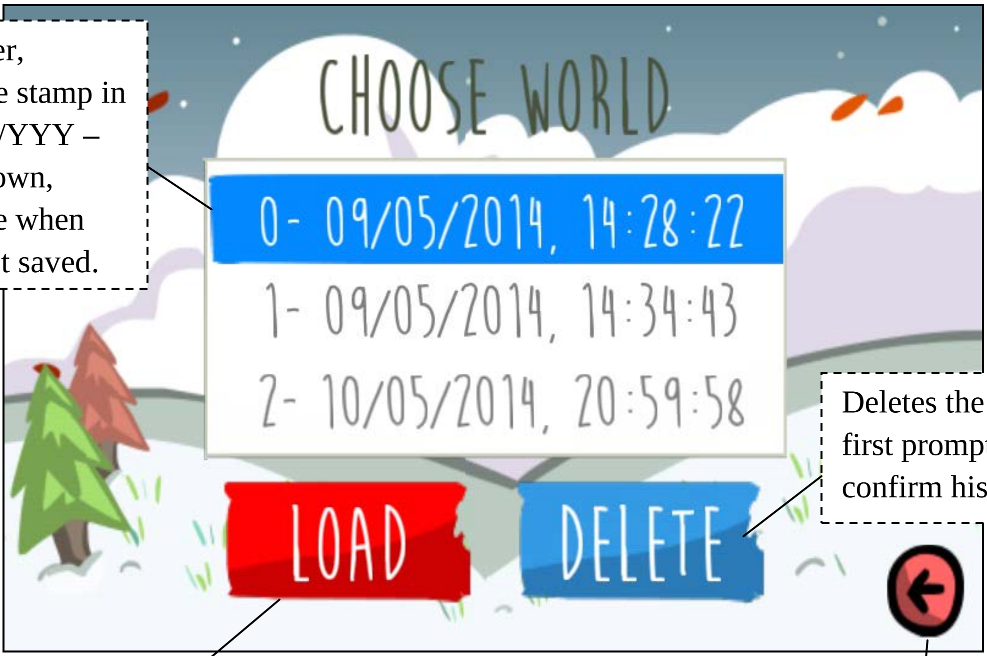

Continues the game from the player's last-saved profile
Transitions to the *World Select Menu*
CONTINUE
NEW GAME
LOAD
Creates a new world
Transitions to the Main *Menu*

**Figure 7: Visual representation of the Game Select Menu**

*World Select:*

The world selection screen displays a list of profiles that the user has saved. The player can choose to load an already-created profile.

(See next page for figure)



The profile number, followed by a time stamp in the form MM/DD/YYY – HH:MM:SS is shown, indicating the time when the profile was last saved.

Deletes the profile. A popup first prompts the user to confirm his choice.

**Figure 8: Visual representation of the World Select Menu**

Starts the game with the selected profile

Transitions to the Main *Menu*

*Exploration HUD:*

This heads-up display (HUD) is shown when the player is exploring the world and scavenging items.

(See next page for visual representation of the Exploration HUD)

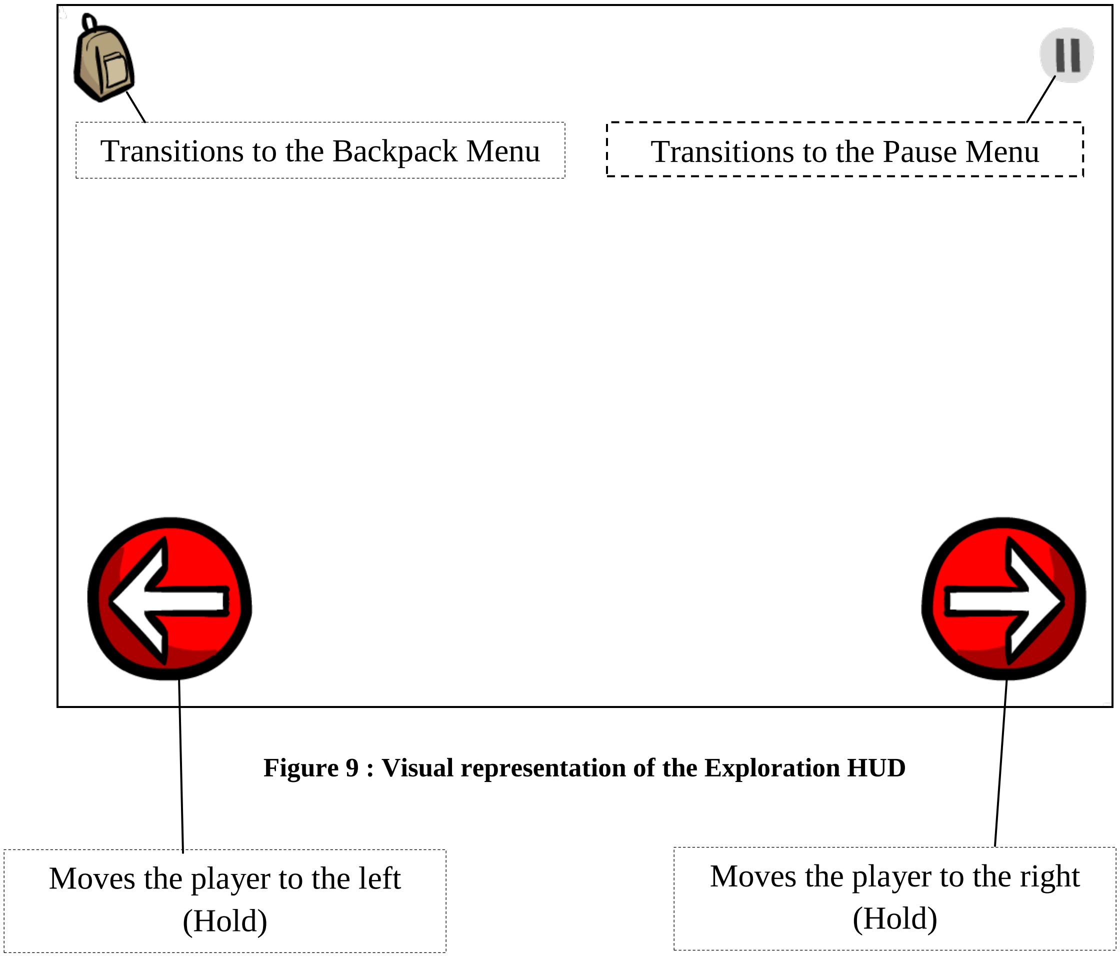

Transitions to the Backpack Menu
Transitions to the Pause Menu
Moves the player to the left (Hold)
Moves the player to the right (Hold)

**Figure 9 : Visual representation of the Exploration HUD**

Great care was put into making the buttons as noticeable as possible. Furthermore, the size of the buttons were chosen to be rather large. This avoids the user from becoming frustrated at his inability to press a button which is too small.

(See next page for Combat HUD)

*Combat HUD:*

This HUD is shown when the player is in combat mode with a zombie. The player has three different options. He can jump, perform a melee attack, or fire his ranged weapon.

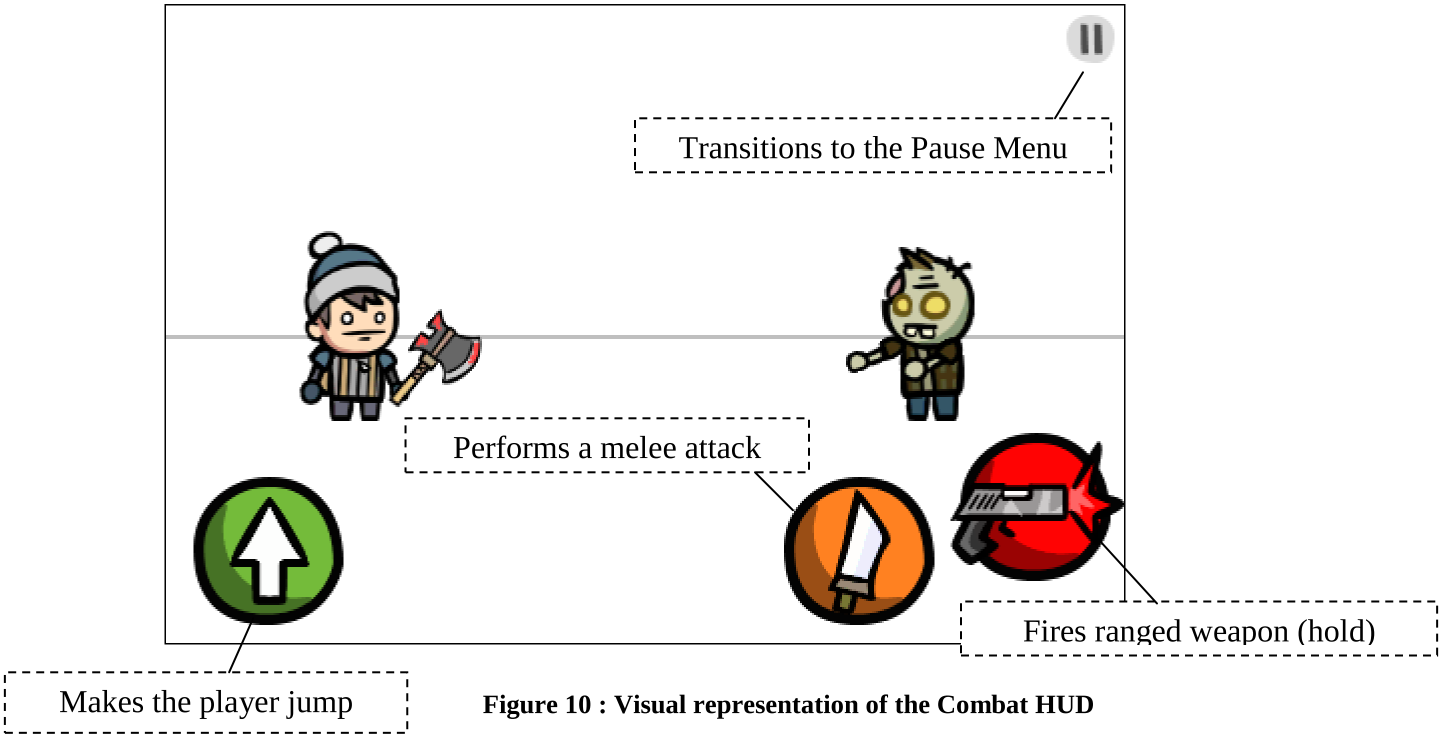


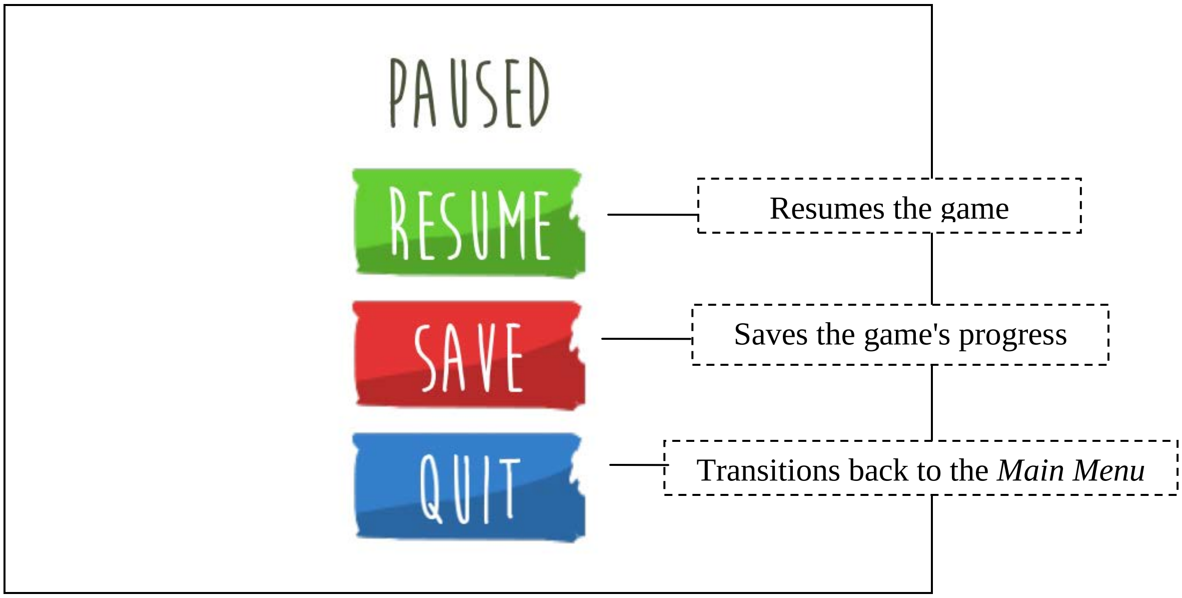
**Figure 10 : Visual representation of the Combat HUD**

Once again, the main action buttons were made to be as large as possible, without obstructing the player's view of the combat. This allowed the user to be able to easily see his options wihout needing to look too closely.

In addition, the buttons were all mapped to different colours to ensure that the player could recognize each different button quickly.

*Pause Menu:*

The pause menu is displayed when the user presses the pause button on the top right corner of the screen. When shown, the game is paused, and the player is prompted to either resume the game or transition to the main menu.



Resumes the game

Saves the game's progress

Transitions back to the *Main Menu*

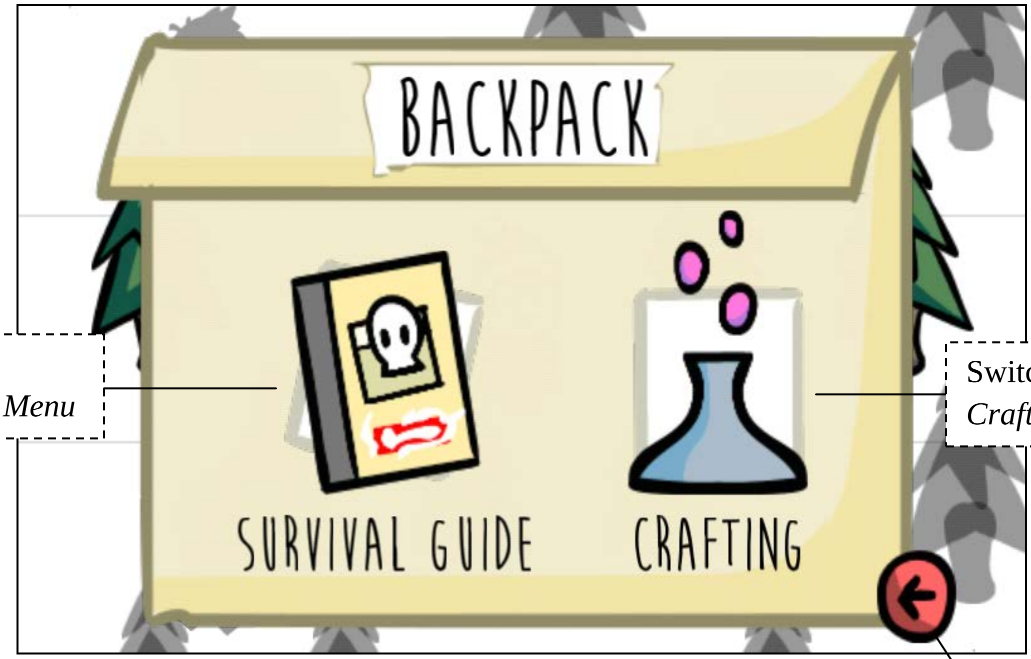
**Figure 11 : Visual representation of the pause menu**

As before, the buttons were mapped to different colours so that the user could recognize each option easily. It is also appealing to the eye, as it adds a needed sense of colour to the game.

(See next page for Backpack Menu)

*Backpack Menu:*

This menu is displayed when the user presses the backpack button. When this menu is shown, the game is paused. Here, the player has a chance to transition to the crafting menu or to transition to the survival guide.



Switches to the *Survival Guide Menu*

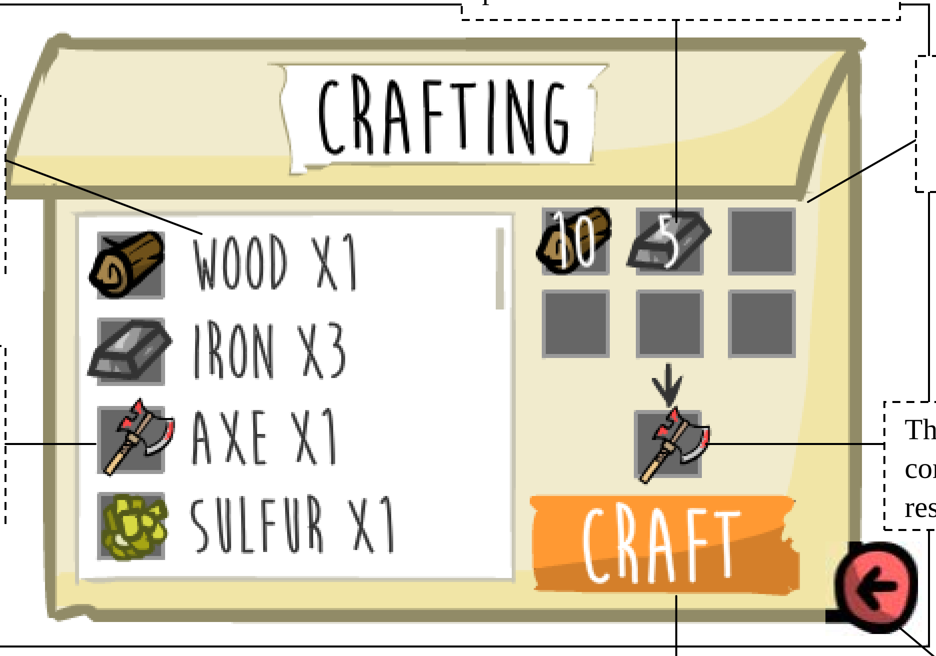
Switches to the *Crafting Menu*

Resumes the game

**Figure 12 : Visual representation of the backpack menu**

*Crafting Menu:*

The crafting menu is accessed from the backpack. It allows the user to craft new items using the objects he has collected in the world. The list on the left shows all of the items that belong to the player. It is called the inventory list. The right-hand boxes called the crafting list, which contains the items that are combined to create a new item. Note that, in order to transfer an item from the item list to the crafting list, the user simply has to click on an item in the item list. In order to transfer an item from the crafting list back to the item list, the user has to click on an item in the crafting list.



The number on top of the item indicates the quantity of items placed in the list.

The name of each item is displayed, followed by the amount available in the inventory.

A total of six items can be added to the crafting list

When an item is pressed, one instance of that item is transferred to the crafting list.

The item crafted by combining the above resources

Transitions to the backpack menu.

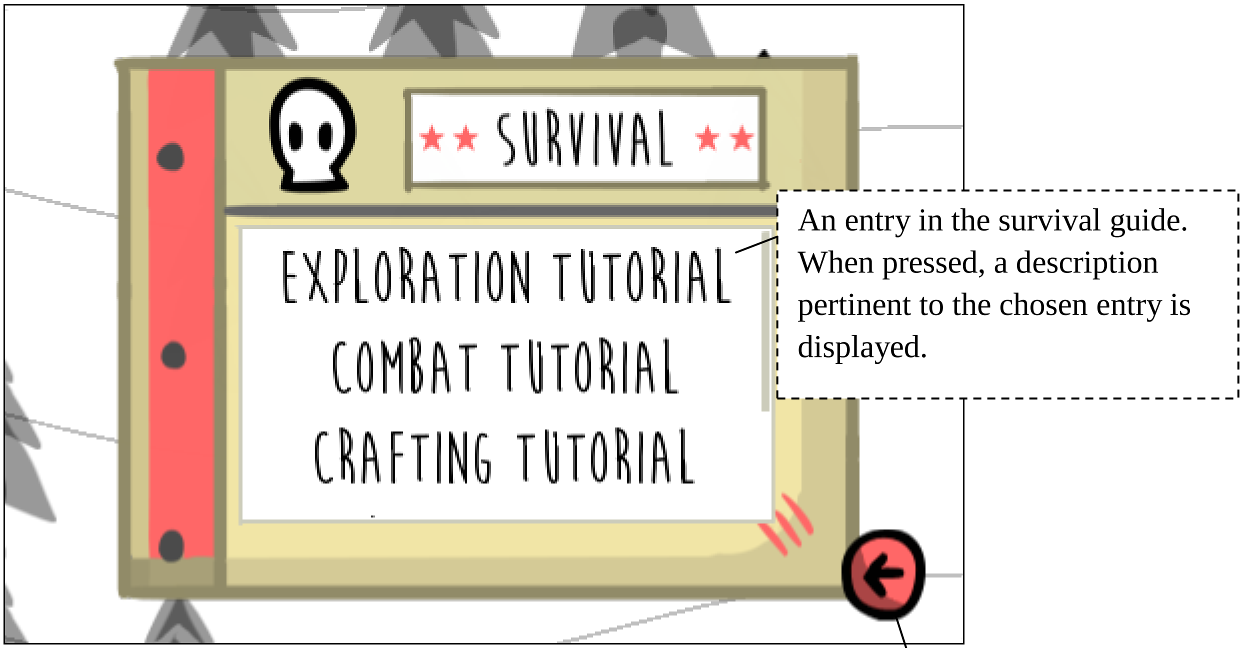
When pressed, a confirmation dialog appears. If confirmed, the above item is created and added to the item list on the left.

**Figure 13 : Visual representation of the crafting menu**

*Survival Guide:*

When the survival guide button is pressed in the backpack menu, this survival guide menu is displayed. The user has a choice of several *entries* in the guide. When an entry is pressed, a description of the entry is shown.

(See next page for visual representation of the menu)



An entry in the survival guide. When pressed, a description pertinent to the chosen entry is displayed.

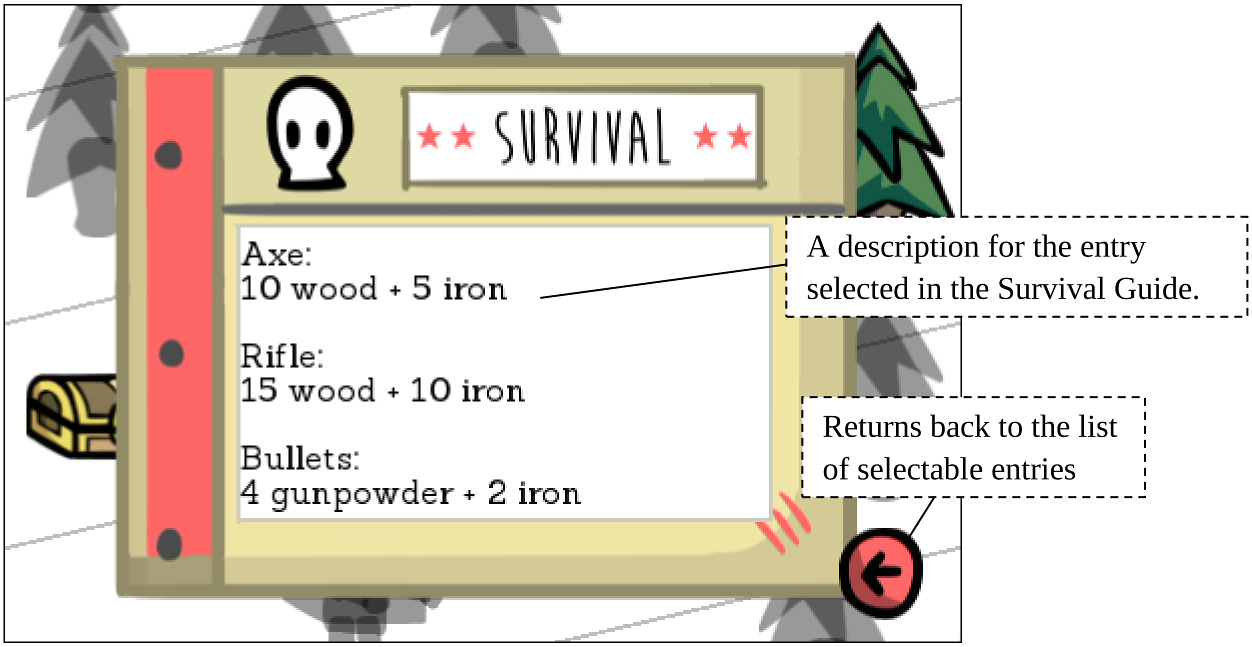
**Figure 14 : Visual representation of the survival guide**

Returns to the *Backpack Menu*

When pressing on the *How to Defend Yourself* entry, for instance, the following description shown in the figure on the following page shows up.

These descriptions aim to help the user understand the features of the game. Without this survival guide, the player would have a lot of trouble navigating his way through the game. Thus, we introduced this guide in order to help the user along his journey through the game.

It also aimed to ease frustration, as a player unaware of the game's mechanics could become easily frustrated at his inability to perform basic procedures.



Axe:
10 wood + 5 iron

Rifle:
15 wood + 10 iron

Bullets:
4 gunpowder + 2 iron

A description for the entry selected in the Survival Guide.

Returns back to the list of selectable entries

**Figure 15 : The survival guide, when one of the entries are pressed and a description appears**

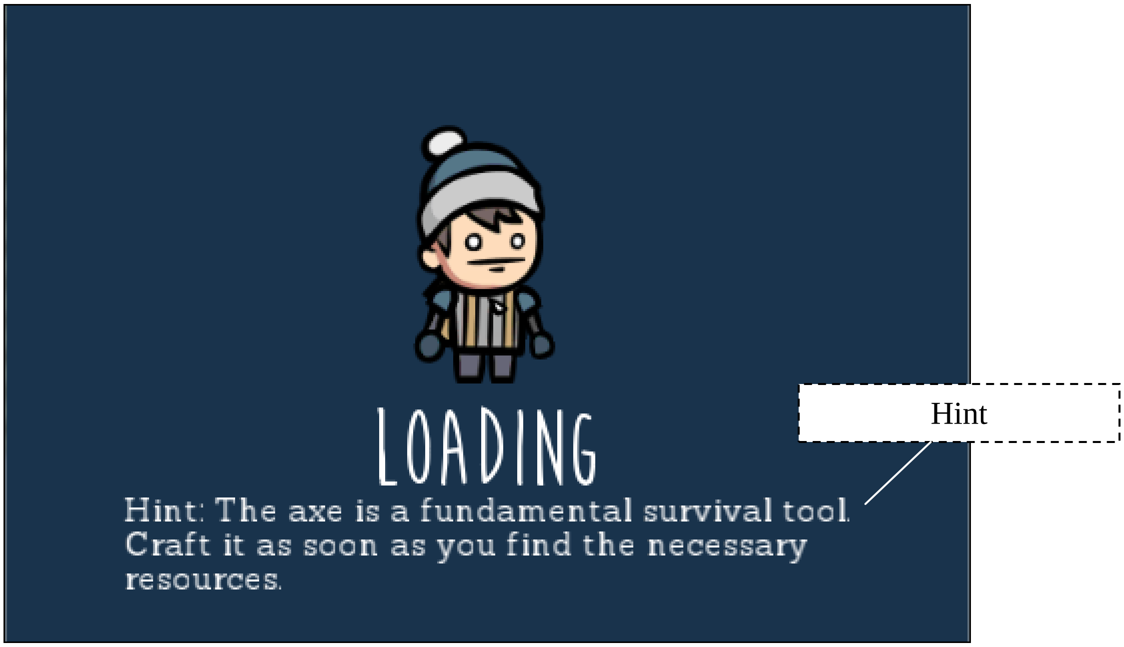
The survival guide informs the player about the game's tutorials, along with recipes needed to craft survival resources. For example, in the *How to Escape* entry, the following description is given:

"Build a teleporter - 40 Sulfur + 30 Iron + 40 Saltpeter + 50 Wood"

*Main Menu Loading Screen:*

This loading screen is displayed whenever the user presses *Quit* in the pause menu and confirms the dialog which opens. It displays a hint, which aims to broaden the user's knowledge of the game.

(See next page for figure)



LOADING

Hint: The axe is a fundamental survival tool. Craft it as soon as you find the necessary resources.

**Figure 16 : Visual representation of the main menu loading screen**

# II.3 UML

## II.3.1 Classes

Important Notes:

1. Floating-point variables are used as opposed to double variables as they take up only four bytes as opposed to eight.

2. Enumerations and inner classes are not presented in this section for the sake of brevity. Their contents normally are usually self-explanatory. Regardless, the full UML diagram of each enumeration and inner class is given in *Section II.3.2 - Class Hierarchies*.

3. Any *render, draw,* or *update* method with a floating-point parameter accepts *deltaTime* as an argument. *deltaTime* is a parameter which holds the time elapsed between the current frame and the previous frame.

4. This section acts as an extension to *Section II.1 Algorithms.* In describing the data and behaviour of each class, more insight is given about the algorithms used in the program.

5. The *Vector2* class is a pre-defined *LibGDX* class used to denote an (x,y) position. It is used to define a position in the world.

6. The Spine animation engine was used to create animations in the game. Thus, there are references to the *Skeleton* class and *attachments* in the class diagrams. A *Skeleton* inside *Spine* refers to a visual entity. This entity has a position and animations, along with attachments. Attachments are simply images attached to a skeleton. They can be activated and deactivated using the attachment's name.

7. The *Sprite* class acts similarly to the *Image* class in Java. It references an image file and can be drawn to the screen using *LibGDX's* pre-defined classes.

8. All classes containing GUI widgets use anonymous inner classes as listeners. Thus, no mention is found in the class diagrams about the widgets' listeners.

9. In certain cases, the setter or a getter for a data field is excluded. This may occur in the case that the member variable should not be mutated by an outside class, or in the case that the variable should not be accessed from an outside class.

10. The *OrthographicCamera* class is defined in the *LibGDX* framework. It represents a camera that can render images to the screen. It has a width, a height, and a position. This class allows us to implement a movable camera that can follow the player's position.

11. The following symbols precede methods and data fields in the UML class diagrams. Their meanings are defined in the figure below.

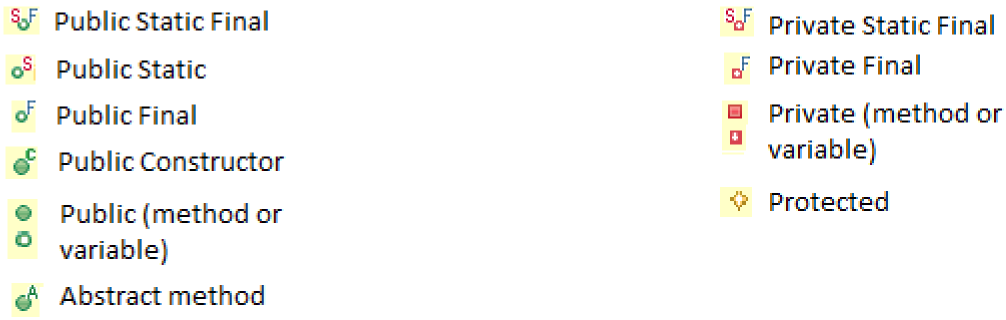
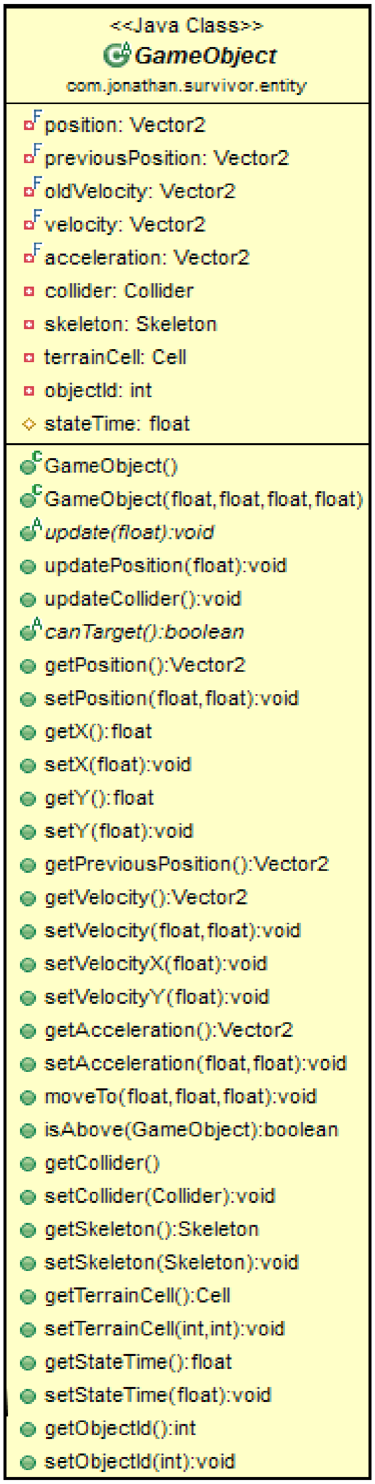


**Figure 17: Legend for UML class diagrams**

II.3.1.1 GameObject (See next page for class diagram)



<<Java Class>>
*GameObject*
com.jonathan.survivor.entity

position: Vector2
previousPosition: Vector2
oldVelocity: Vector2
velocity: Vector2
acceleration: Vector2
collider: Collider
skeleton: Skeleton
terrainCell: Cell
objectId: int
stateTime: float

GameObject()
GameObject(float, float, float, float)
*update(float):void*
updatePosition(float):void
updateCollider():void
*canTarget():boolean*
getPosition():Vector2
setPosition(float, float):void
getX():float
setX(float):void
getY():float
setY(float):void
getPreviousPosition():Vector2
getVelocity():Vector2
setVelocity(float, float):void
setVelocityX(float):void
setVelocityY(float):void
getAcceleration():Vector2
setAcceleration(float, float):void
moveTo(float, float, float):void
isAbove(GameObject):boolean
getCollider()
setCollider(Collider):void
getSkeleton():Skeleton
setSkeleton(Skeleton):void
getTerrainCell():Cell
setTerrainCell(int,int):void
getStateTime():float
setStateTime(float):void
getObjectId():int
setObjectId(int):void

**Figure 18 : GameObject class diagram**

The *GameObject* class represents an entity in the world. It has a position, a velocity, and an acceleration. The updatePosition() method performs the integration of velocity and acceleration. In fact, this method takes the velocity and the acceleration of the *GameObject*, and moves the object's position according to these values. This integration step is done using *Euler* integration. This integration method uses the following procedure to calculate the velocity and the position of a *GameObject* every time the game updates.

```
//Pseudo code

velocity += aceleration * deltaTime;

position += velocity * deltaTime;
```

The *collider* of each GameObject is an instance of *Rectangle*. This rectangle is used to detect collisions between two entities in the world. On the other hand, the *terrainCell:Cell* member variable holds the cell coordinate of the *GameObject* in the *TerrainLevel*. For instance, if the player is in row zero, column zero, then the player is on the layer with row zero and column zero. This member variable allows each *GameObject* to know which entities in the world it may interact with. For instance, the player can only touch *GameObjects* on the same row as the player. Thus, the member variable tells the player if he is on the same row as other objects.

On the other hand, the *stateTime* float is used for animations. It increments by *deltaTime* every frame. It allows the *GameObject* to know how much time it has been in its current state. This allows the *Renderer* objects to know which frame of animation to play for a given entity based on how long the animation has played. Further, every time the *GameObject* changes states, this number is re-set to zero.

Furthermore, the *objectId* variables are integers used to number *GameObjects* in a *TerrainLayer*. The first *GameObject* placed in a layer is numbered zero, the second is numbered one, and the rest are numbered sequentially. Inside the player's profile, a HashMap stores the *objectId's* for each *GameObject* that has been destroyed or scavenged in a layer. This is used to prevent *GameObjects* that have been scavenged on a layer from being placed there again. Like this, *GameObjects* that have been previously scavenged will not re-appear in the world. For instance, a chopped tree will not re-appear once destroyed.

The *GameObject's* constructor accepts four floats: the width and height of the object’s collider, and the x and y positions of the object.

Next, the abstract *update()* method is overridden by subclasses to control the object's game logic. Conversely, the *updateCollider()* method is called every game tick in order to update the *GameObject’s* collider so that it follows his position.

The abstract *canTarget():boolean* method returns true if the GameObject can be targeted and attacked by a *Human* instance. A *GameObject* can be targetted if the player can advance towards the object by clicking on it. The *moveTo(x:float, y:float, time:float): void* method moves the GameObject to the desired (x,y) position in the given amount of time in seconds. It is used to move items towards the player once they are pressed.

```
public abstract class GameObject
extends java.lang.Object
```

## Field Detail

### position

```
private final Vector2 position
```

Stores the bottom-bottom-center position of the GameObject

### previousPosition

```
private final Vector2 previousPosition
```

Holds the GameObject's previous position before the current game tick.

### oldVelocity

```
private final Vector2 oldVelocity
```

Stores the velocity of the gameObject the previous frame.

### velocity

```
private final Vector2 velocity
```

Stores the velocity of the GameObject

### acceleration

```
private final Vector2 acceleration
```

Stores the acceleration of the GameObject

### collider

```
private Collider collider
```

Stores the collider used by the GameObject for collision

### skeleton

```
private com.esotericsoftware.spine.Skeleton skeleton
```

Stores the Spine skeleton that controls the bones of the GameObject and its appearance.

### terrainCell

```
private Cell terrainCell
```

Stores the row and column corresponding to the layer where the GameObject resides on the TerrainLevel.

### objectId

```
private int objectId
```

The GameObject's id, used to identify GameObjects inside a save file. Used to identify whether or not a GameObject should be placed on a TerrainLayer.

### stateTime

```
protected float stateTime
```

Stores the amount of time the GameObject has been in a specific state. (i.e., if the player has been jumping for 0.5 seconds, stateTime = 0.5).

## Constructor Detail

### GameObject

```
public GameObject()
```

Creates a GameObject with a bottom-bottom-center at (0,0) and a rectangle collider with width/height of zero.

### GameObject

```
public GameObject(float x, float y,float width, float height)
```

Creates a GameObject with bottom-center at (x,y) and rectangle collider with given width/height.

**Parameters:**

x - the center x-position of the GameObject (in world units)

y - the center y-position of the GameObject (in world units) `width` - the width of the GameObject's rectangle collider

`height` - the height of the GameObject's rectangle collider

## Method Detail

### update

```
public abstract void update(float deltaTime)
```

Updates the GameObject's game logic.

**Parameters:**

`deltaTime` - the amount of time the last render call took to complete

### updatePosition

```
public void updatePosition(float deltaTime)
```

Updates the position of the GameObject according to its velocity and acceleration.

### updateCollider

```
public void updateCollider()
```

Snaps the GameObject's collider to the GameObject's position

### canTarget

```
public abstract boolean canTarget()
```

Returns true if the GameObject can be targetted by a Human.

### getPosition

```
public Vector2 getPosition()
```

Retrieves the bottom-center position of the gameObject as a Vector2. Operations can be performed on the Vector2 using its instance methods, as it is mutable.

### setPosition

```
public void setPosition(float x,float y)
```

Sets the bottom-center position of the GameObject at the desired (x,y) coordinates.

**Parameters:**

x - the center x-position in which to place the GameObject

y - the center y-position in which to place the GameObject

### getX

```
public float getX()
```

Returns the bottom-center x-position of the GameObject.

### setX

```
public void setX(float x)
```

Sets the center x-position of the GameObject.

### getY

```
public float getY()
```

Returns the bottom-center y-position of the GameObject.

### setY

```
public void setY(float y)
```

Sets the bottom y-position of the GameObject.

### getPreviousPosition

```
public Vector2 getPreviousPosition()
```

Returns the GameObject's previous position before the current game tick.

### getVelocity

```
public Vector2 getVelocity()
```

Retrieves the velocity of the gameObject as a Vector2. Operations can be performed on the Vector2 using its instance methods, as it is mutable.

### setVelocity

```
public void setVelocity(float x, float y)
```

Sets the velocity of the GameObject at their desired (x,y) values.

### setVelocityX

```
public void setVelocityX(float x)
```

Updates the x-velocity of the GameObject.

### setVelocityY

```
public void setVelocityY(float y)
```

Updates the y-velocity of the GameObject.

### getAcceleration

```
public Vector2 getAcceleration()
```

Retrieves the acceleration of the gameObject as a Vector2. Operations can be performed on the Vector2 using its instance methods, as it is mutable.

### setAcceleration

```
public void setAcceleration(float x, float y)
```

Sets the acceleration of the GameObject at their desired (x,y) values.

### moveTo

```
public void moveTo(float x, float y, float time)
```

Moves the GameObject to the desired (x,y) position in a straight line at the given speed in m/s.

**Parameters:**

`x` - the center x-position where to move the GameObject

`y` - the center y-position where to move the GameObject

`time` - the time it will take to move the GameObject to the designated position

### isAbove

```
public boolean isAbove(GameObject gameObject)
```

Returns true if this GameObject is above the given GameObject.

**Parameters:**

`gameObject` - the GameObject which is tested to see if this GameObject is above this parameter

**Returns:**

true, if the GameObject is above the given GameObject

### getCollider

```
public <T extends Collider> T getCollider()
```

Returns the collider used by the gameObject for collisions.

### setCollider

```
public void setCollider(Collider c)
```

Sets the collider used by the gameObject for collisions.

### getSkeleton

```
public com.esotericsoftware.spine.Skeleton getSkeleton()
```

Returns the skeleton used to render the GameObject to the screen. Returns null if the GameObject doesn't use a Spine skeleton.

### setSkeleton

```
public void setSkeleton(com.esotericsoftware.spine.Skeleton skeleton)
```

Sets the spine skeleton used by the GameObject to be rendered on-screen.

### getTerrainCell

```
public Cell getTerrainCell()
```

Gets the cell coordinates where the GameObject is placed on the TerrainLevel.

### setTerrainCell

```
public void setTerrainCell(int row, int col)
```

Sets the cell coordinates where the GameObject is placed on the TerrainLevel.

### getStateTime

```
public float getStateTime()
```

Gets the amount of time the GameObject has been in a specific state. (e.g., stateTime = 0.4 if the GameObject has been in the JUMP state for 0.4 seconds.

### setStateTime

```
public void setStateTime(float stateTime)
```

Sets the amount of time the GameObject has been in a specific state. Only the gameObject's update() method should update this value.

### getObjectId

```
public int getObjectId()
```

Gets the ID of the GameObject, used to identify a GameObject in a specific TerrainLayer.

### setObjectId

```
public void setObjectId(int objectId)
```

Sets the ID of the GameObject, used to identify a GameObject in a specific TerrainLayer.

### II.3.1.2 Human

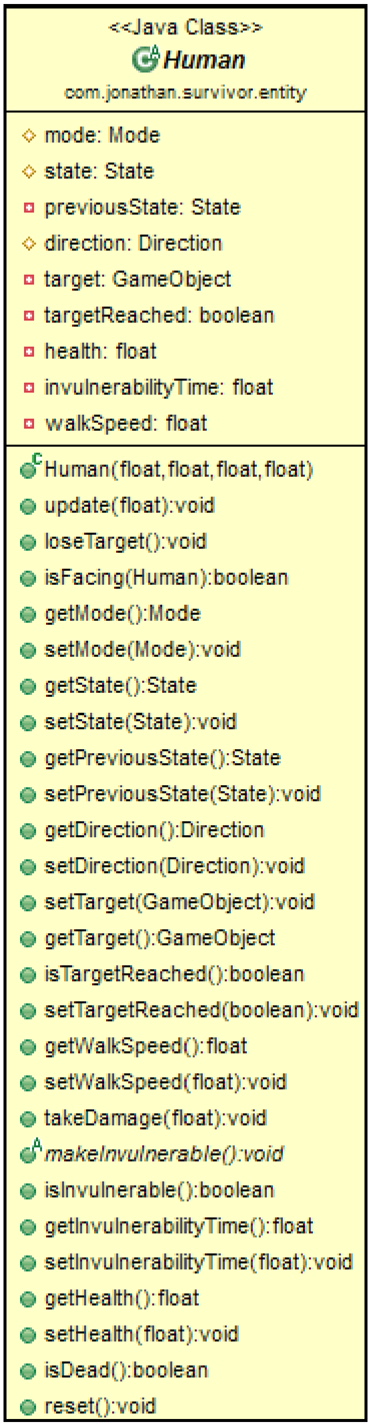


**Figure 19 : Human class diagram**

The Human class is the superclass of every *GameObject* that has some sort of intelligence with regards to movement. The *mode* member variable dictates whether the Human is exploring the world or in combat mode. If the human is in combat mode, the *Human's* game logic takes a different code path inside its *update()* method.

On a separate note, the *state* data field is another variable holding an enumeration constant. It is used to dictate which state the human is in. It tells the human which action it is currently performing. For instance, the *Human* can be in the *WALK* state or the *JUMP* state, indicating that the player is currently walking or running. Changing the state of a *Human* changes his current action.

The *direction* instance variable defines whether the Human is facing left or right. A *Human* also has a *target*. A target is a *GameObject*. It can be a tree, a box, or another *Human*. The *Human* will walk towards his target if it is not null. If the data field is null, it has no effect on the human's actions. The *targetReached* boolean is true if the *Human* has touched his target's collider.

The *Human's* constructor accepts four floats: the width and height of the object's collider, along with the x and y positions of the object upon instantiation.

The abstract *update()* method updates the position of the *Human*. The *loseTarget()* method simply sets the *target* member variable to null. This happens when the player chops down a tree, for instance. In this case, the player has scavenged the tree, and thus has no more target. He no longer needs to move towards said tree.

```
public abstract class Human
extends GameObject
implements com.badlogic.gdx.utils.Pool.Poolable
```

### Field Detail

#### mode

```
protected Human.Mode mode
```

Stores the mode of the Human (EXPLORING or FIGHTING) used to determine how the human's logic should be processed.

### state

```
protected Human.State state
```

Stores the state of the Human (IDLE, WALK, etc.), usually used to dictate which animations to play.

### previousState

```
private Human.State previousState
```

Stores the previous state of the human. Used in humans' renderer. Animations are only changed if the human's state changed from the previous render call.

### direction

```
protected Human.Direction direction
```

Stores the direction the Human is facing

### target

```
private GameObject target
```

Stores the GameObject where the human is trying to walk to. Null if the human has no target.

### targetReached

```
private boolean targetReached
```

Holds true if the Human has reached his target.

### health

```
private float health
```

Stores the human's health. Once it drops below zero, the human is dead.

### invulnerabilityTime

```
private float invulnerabilityTime
```

Holds the amount of time that the Human is invulnerable to attacks.

### walkSpeed

```
private float walkSpeed
```

Holds the walking speed of the human in the x-direction in meters/second.

## Constructor Detail

### Human

```
public Human(float x, float y, float width, float height)
```

Creates a Human GameObject instance whose bottom-center is at (x,y) and whose Rectangle collider is initialized with the given width and height.

**Parameters:**

x - the center x-position of the Human (in world units)

y - the center y-position of the Human (in world units)

`width` - the width of the Human's rectangle collider

`height` - the height of the Human's rectangle collider

## Method Detail

### update

```
public void update(float deltaTime)
```

Updates the Human's game logic, such as his state time.

**Specified by:**

`update` in class `GameObject`

**Parameters:**

`deltaTime` - the amount of time the previous render call took to execute

### loseTarget

```
public void loseTarget()
```

Lose the human's current target so that he stops walking towards his target.

### isFacing

```
public boolean isFacing(Human other)
```

Returns true if the given human is facing the other human, and this human can thus see the other one.

**Parameters:**

`other` - the Human he must be looking at to return true

**Returns:**

true, if this human is facing the given Human

### getMode

```
public Human.Mode getMode()
```

Gets the mode (EXPLORING or FIGHTING) of the GameObject

### setMode

```
public void setMode(Human.Mode mode)
```

Sets the mode (EXPLORING or FIGHTING) of the GameObject

### getState

```
public Human.State getState()
```

Gets the state (IDLE, JUMP, etc.) of the GameObject, used to dictate which animations to use.

### setState

```
public void setState(Human.State state)
```

Sets the state (IDLE, JUMP, etc.) of the GameObject, used to dictate which animations to use. Also sets the stateTime of the GameObject back to zero.

### getPreviousState

```
public Human.State getPreviousState()
```

Retrieves the previous state of the Human. Used to determine whether or not a GameObject's state changes from one render call to the next.

### setPreviousState

```
public void setPreviousState(Human.State previousState)
```

Sets the previous state of the Human. Used to determine whether or not a GameObject's state changes from one render call to the next.

### getDirection

```
public Human.Direction getDirection()
```

Gets the direction (LEFT or RIGHT) that the GameObject is facing.

### setDirection

```
public void setDirection(Human.Direction direction)
```

Sets the direction (LEFT or RIGHT) that the GameObject is facing.

### setTarget

```
public void setTarget(GameObject target)
```

Sets the target where the human wants to walk to.

**Parameters:**

`target` - the human's new target

### getTarget

```
public GameObject getTarget()
```

Gets the target where the human wants to walk to.

### isTargetReached

```
public boolean isTargetReached()
```

Returns true if the human has reached his target.

### setTargetReached

```
public void setTargetReached(boolean targetReached)
```

Sets whether or not the human has reached his target.

### getWalkSpeed

```
public float getWalkSpeed()
```

Gets the human's walking speed in the x-direction.

### setWalkSpeed

```
public void setWalkSpeed(float walkSpeed)
```

Sets the human's walking speed in the x-direction in meters/second.

### takeDamage

```
public void takeDamage(float damage)
```

Deals damage to the human by removing the given amount from its health.

**Parameters:**

`damage` - the damage dealt to the human

### makeInvulnerable

```
public abstract void makeInvulnerable()
```

Makes the Human invulnerable for a given amount of seconds.

### isInvulnerable

```
public boolean isInvulnerable()
```

Returns true if the Human is invulnerable to incoming attacks.

### getInvulnerabilityTime

```
public float getInvulnerabilityTime()
```

Gets the amount of time that the Human is invulnerable for.

### setInvulnerabilityTime

```
public void setInvulnerabilityTime(float invulnerabilityTime)
```

Sets the amount of time that the Human is invulnerable for.

### getHealth

```
public float getHealth()
```

Gets the human's health.

### setHealth

```
public void setHealth(float health)
```

Sets the human's health.

### isDead

```
public boolean isDead()
```

Returns true if the Human's health has dropped to zero or below.

### reset

```
public void reset()
```

Called whenever this box GameObject has been pushed back into a pool. In this case, we reset the box's state back to default.

**Specified by:**

`reset` in interface `com.badlogic.gdx.utils.Pool.Poolable`

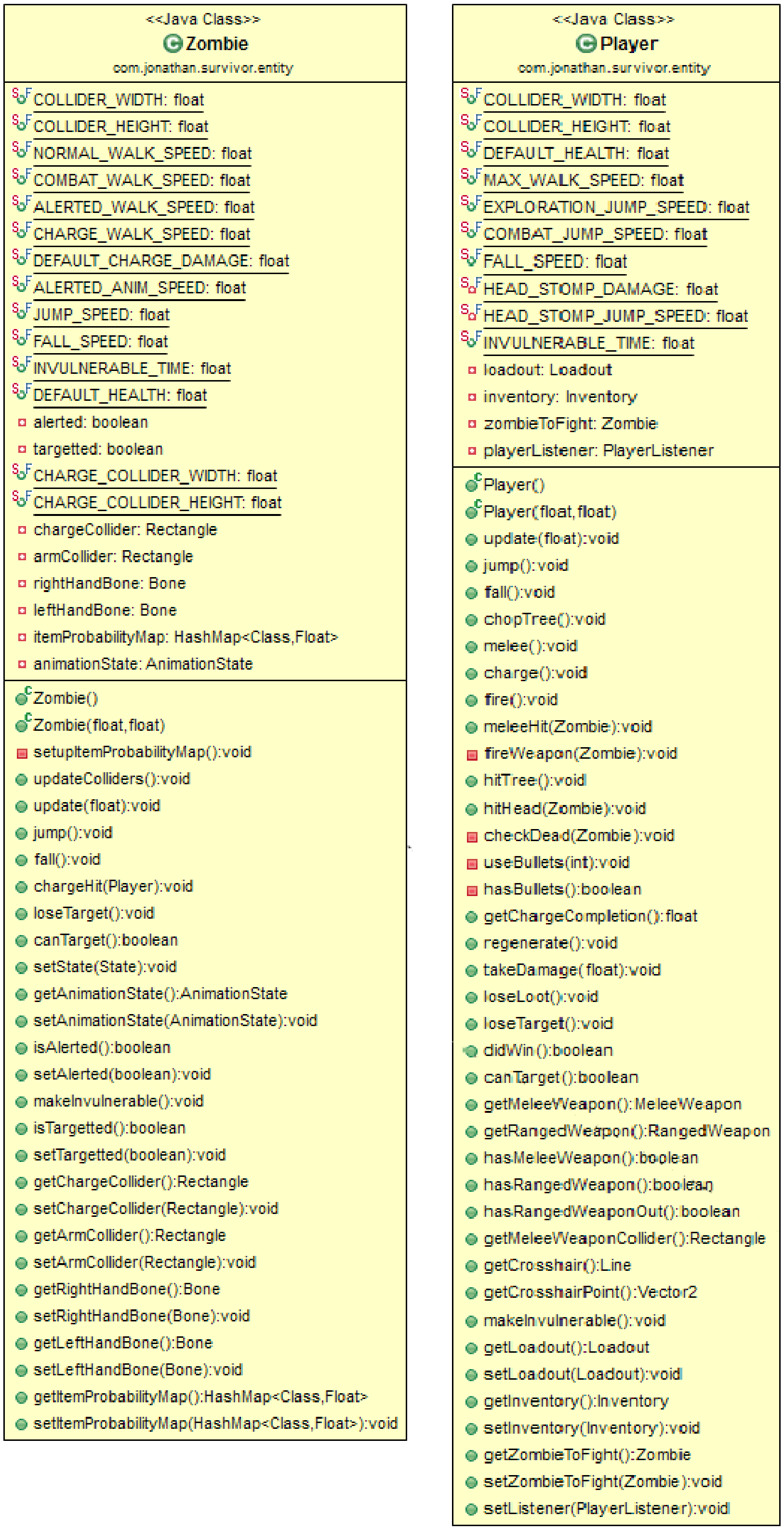
### II.3.1.3 Player & Zombie

The *Player* and *Zombie* classes both have similar member variables. First, the *Player* class has public static final floats *COLLIDER_WIDTH:float* and *COLLIDER_HEIGHT:float*, which indicate the size of their colliders. The *MAX_WALK_SPEED:float* constants indicate the maximum walking speed of each entity. The *JUMP_SPEED:float* and *FALL_SPEED:float* constants both indicate the vertical speed in metres per second when the player either jumps or falls.

The player's *loadout* holds the weapons equipped by the player, and the *inventory* holds the items the player has collected or crafted. The *playerListener* instance is used by the *World* class. The  player delegates method calls to this instance for the world to be aware of any player events.

The default constructors of both classes simply instantiate a *Player* or a *Zombie* at position (0, 0) in the world. The second constructors accept the (x, y) position of the *Player* or the *Zombie* as parameters.

(See next page for class diagrams)



<<Java Class>>
Zombie
com.jonathan.survivor.entity

- COLLIDER_WIDTH: float
- COLLIDER_HEIGHT: float
- NORMAL_WALK_SPEED: float
- COMBAT_WALK_SPEED: float
- ALERTED_WALK_SPEED: float
- CHARGE_WALK_SPEED: float
- DEFAULT_CHARGE_DAMAGE: float
- ALERTED_ANIM_SPEED: float
- JUMP_SPEED: float
- FALL_SPEED: float
- INVULNERABLE_TIME: float
- DEFAULT_HEALTH: float
- alerted: boolean
- targetted: boolean
- CHARGE_COLLIDER_WIDTH: float
- CHARGE_COLLIDER_HEIGHT: float
- chargeCollider: Rectangle
- armCollider: Rectangle
- rightHandBone: Bone
- leftHandBone: Bone
- itemProbabilityMap: HashMap<Class,Float>
- animationState: AnimationState

- Zombie()
- Zombie(float,float)
- setupItemProbabilityMap():void
- updateColliders():void
- update(float):void
- jump():void
- fall():void
- chargeHit(Player):void
- loseTarget():void
- canTarget():boolean
- setState(State):void
- getAnimationState():AnimationState
- setAnimationState(AnimationState):void
- isAlerted():boolean
- setAlerted(boolean):void
- makeInvulnerable():void
- isTargetted():boolean
- setTargetted(boolean):void
- getChargeCollider():Rectangle
- setChargeCollider(Rectangle):void
- getArmCollider():Rectangle
- setArmCollider(Rectangle):void
- getRightHandBone():Bone
- setRightHandBone(Bone):void
- getLeftHandBone():Bone
- setLeftHandBone(Bone):void
- getItemProbabilityMap():HashMap<Class,Float>
- setItemProbabilityMap(HashMap<Class,Float>):void

<<Java Class>>
Player
com.jonathan.survivor.entity

- COLLIDER_WIDTH: float
- COLLIDER_HEIGHT: float
- DEFAULT_HEALTH: float
- MAX_WALK_SPEED: float
- EXPLORATION_JUMP_SPEED: float
- COMBAT_JUMP_SPEED: float
- FALL_SPEED: float
- HEAD_STOMP_DAMAGE: float
- HEAD_STOMP_JUMP_SPEED: float
- INVULNERABLE_TIME: float
- loadout: Loadout
- inventory: Inventory
- zombieToFight: Zombie
- playerListener: PlayerListener

- Player()
- Player(float,float)
- update(float):void
- jump():void
- fall():void
- chopTree():void
- melee():void
- charge():void
- fire():void
- meleeHit(Zombie):void
- fireWeapon(Zombie):void
- hitTree():void
- hitHead(Zombie):void
- checkDead(Zombie):void
- useBullets(int):void
- hasBullets():boolean
- getChargeCompletion():float
- regenerate():void
- takeDamage(float):void
- loseLoot():void
- loseTarget():void
- didWin():boolean
- canTarget():boolean
- getMeleeWeapon():MeleeWeapon
- getRangedWeapon():RangedWeapon
- hasMeleeWeapon():boolean
- hasRangedWeapon():boolean
- hasRangedWeaponOut():boolean
- getMeleeWeaponCollider():Rectangle
- getCrosshair():Line
- getCrosshairPoint():Vector2
- makeInvulnerable():void
- getLoadout():Loadout
- setLoadout(Loadout):void
- getInventory():Inventory
- setInventory(Inventory):void
- getZombieToFight():Zombie
- setZombieToFight(Zombie):void
- setListener(PlayerListener):void

**Figure 17: Player and Zombie class diagrams**

```
public class Player
extends Human
```

## Field Detail

### COLLIDER_WIDTH

```
public static final float COLLIDER_WIDTH
```

Stores the width of the player's rectangle collider in world units.

### COLLIDER_HEIGHT

```
public static final float COLLIDER_HEIGHT
```

Stores the height of the player's rectangle collider in world units.

### DEFAULT_HEALTH

```
public static final float DEFAULT_HEALTH
```

Holds the player's default health.

### MAX_WALK_SPEED

```
public static final float MAX_WALK_SPEED
```

Stores the maximum walk speed of the player in the horizontal direction.

### EXPLORATION_JUMP_SPEED

```
public static final float EXPLORATION_JUMP_SPEED
```

Stores the jump speed of the player in the vertical direction when in EXPLORING mode.

### COMBAT_JUMP_SPEED

```
public static final float COMBAT_JUMP_SPEED
```

Stores the jump speed of the player in the vertical direction when in COMBAT mode.

### FALL_SPEED

```
public static final float FALL_SPEED
```

Stores the downwards speed at which the player falls through a TerrainLayer.

### HEAD_STOMP_DAMAGE

```
private static final float HEAD_STOMP_DAMAGE
```

Holds the amount of damage delivered to a zombie when it is stomped on the head by the player.

### HEAD_STOMP_JUMP_SPEED

```
private static final float HEAD_STOMP_JUMP_SPEED
```

Stores the speed at which the player jumps after hitting the zombie's head.

### INVULNERABLE_TIME

```
public static final float INVULNERABLE_TIME
```

Holds the amount of time that the player is invulnerable when hit.

### loadout

```
private Loadout loadout
```

Stores the player's loadout, containing the player's active weapons.

### inventory

```
private Inventory inventory
```

Holds the player's inventory, which contains all of the player's collected items.

### zombieToFight

```
private Zombie zombieToFight
```

Holds the zombie that the player will fight once he enters combat mode. Convenience member variable to avoid large work-arounds.

### playerListener

```
private PlayerListener playerListener
```

Stores the PlayerListener instance where methods are delegated upon player events.

## Constructor Detail

### Player

```
public Player()
```

Creates a player whose bottom-center is at position (0, 0).

### Player

```
public Player(float x, float y)
```

Creates a player whose bottom-center is at position (x, y).

**Parameters:**

x - the center x-position where to place the player (in world coordinates) y - the center x-position where to place the player (in world coordinates)

## Method Detail

### update

```
public void update(float deltaTime)
```

Updates the player's internal game logic.

**Overrides:**

`update` in class `Human`

**Parameters:**

`deltaTime` - the execution time of the previous render call

### jump

```
public void jump()
```

Makes the player jump.

### fall

```
public void fall()
```

Makes the player fall through one layer.

### chopTree

```
public void chopTree()
```

Makes the player start chopping a tree

### melee

```
public void melee()
```

Makes the player swing his melee weapon if he has one.

### charge

```
public void charge()
```

Makes the player start to charge his gun. Call only once, when the player starts charging his ranged weapon.

### fire

```
public void fire()
```

Makes the player fire his ranged weapon

### meleeHit

```
public void meleeHit(Zombie zombie)
```

Deals damage to the zombie with the player's melee weapon. Only deals damage if the player's melee weapon is colliding with the given zombie.

**Parameters:**

`zombie` - the zombie to hit

### fireWeapon

```
private void fireWeapon(Zombie zombie)
```

Fire the player's currently equipped ranged weapon at the given zombie.

**Parameters:**

`zombie` - the zombie to hit

### hitTree

```
public void hitTree()
```

Called when the player has hit the tree stored as his target.

### hitHead

```
public void hitHead(Zombie zombie)
```

Called when the player hits a zombie's head. Deals damage to this zombie and changes its state.

**Parameters:**

`zombie` - the zombie to hit

### checkDead

```
private void checkDead(Zombie zombie)
```

Checks if the zombie is dead. If so, plays the KO animation.

**Parameters:**

`zombie` - the zombie who is checked to be dead

### useBullets

```
private void useBullets(int quantity)
```

Uses a given amount of bullets inside the player's inventory.

**Parameters:**

`quantity` - the quantity of bullets to lose

### hasBullets

```
public boolean hasBullets()
```

Returns true if the player has bullets in his inventory.

### getChargeCompletion

```
public float getChargeCompletion()
```

Returns a float between 0 and 1 representing the charge completion of the player's ranged weapon. 1 means that the weapon is done charging completely.

### regenerate

```
public void regenerate()
```

Regenerates the player to default health.

### takeDamage

```
public void takeDamage(float amount)
```

Overrides the takeDamage() method to take note of when the player dies, in order to display the KO animation.

**Overrides:**

`takeDamage` in class `Human`

**Parameters:**

`amount` - the amount of damage dealt to the player

### loseLoot

```
public void loseLoot()
```

Makes the player lose all of the items in his inventory.

### loseTarget

```
public void loseTarget()
```

Called when the player loses his target.

**Overrides:**

loseTarget in class Human

### didWin

```
public boolean didWin()
```

Returns true if the player has won the game.

### canTarget

```
public boolean canTarget()
```

Override the canTarget method as always returning false since the Player can never be targetted.

**Specified by:**

canTarget in class GameObject

### getMeleeWeapon

```
public MeleeWeapon getMeleeWeapon()
```

Returns the melee weapon that the player has equipped.

### getRangedWeapon

```
public RangedWeapon getRangedWeapon()
```

Returns the ranged weapon that the player has equipped.

### hasMeleeWeapon

```
public boolean hasMeleeWeapon()
```

Returns true if the player has a melee weapon equipped.

### hasRangedWeapon

```
public boolean hasRangedWeapon()
```

Returns true if the player has a ranged weapon equipped.

### hasRangedWeaponOut

```
public boolean hasRangedWeaponOut()
```

Returns true if the player has his ranged weapon out and visible.

### getMeleeWeaponCollider

```
public Rectangle getMeleeWeaponCollider()
```

Returns the Collider of the player's melee weapon. Allows to test if the player has hit a zombie with his weapon.

### getCrosshair

```
public Line getCrosshair()
```

Returns the crosshair line, which dictates where the player's ranged weapon will fire and where the bullet will travel.

### getCrosshairPoint

```
public Vector2 getCrosshairPoint()
```

Returns the position where the weapon crosshair should be placed on the player in world units. This is usually the tip of the player's ranged weapon.

### makeInvulnerable

```
public void makeInvulnerable()
```

Makes the player invulnerable from attacks for a given amount of seconds.

**Specified by:**

> `makeInvulnerable` in class `Human`

### getLoadout

```
public Loadout getLoadout()
```

Retrieves the player's loadout containing the player's weapons.

### setLoadout

```
public void setLoadout(Loadout loadout)
```

Sets the player's loadout.

### getInventory

```
public Inventory getInventory()
```

Gets the loadout which stores the items held by the player.

### setInventory

```
public void setInventory(Inventory inventory)
```

Sets the loadout which stores the items held by the player.

### getZombieToFight

```
public Zombie getZombieToFight()
```

Returns the zombie that the player should fight once he enters combat mode. Set when the player collides with a zombie.

### setZombieToFight

```
public void setZombieToFight(Zombie zombieToFight)
```

Sets the zombie the player should fight once he enters combat mode. Set when the player collides with a zombie.

### setListener

```
public void setListener(PlayerListener listener)
```

Sets the given listener to have its methods delegated by the player instance.

```
public class Zombie
extends Human
implements Clickable
```

## Field Detail

### COLLIDER_WIDTH

```
public static final float COLLIDER_WIDTH
```

Stores the width of the zombie's rectangle collider in world units.

### COLLIDER_HEIGHT

```
public static final float COLLIDER_HEIGHT
```

Stores the height of the zombie's rectangle collider in world units.

### CHARGE_COLLIDER_WIDTH

```
public static final float CHARGE_COLLIDER_WIDTH
```

Stores the width of the zombie's charge collider in world units. The charge collider dictates the region where the zombie can hit the player while charging.

### CHARGE_COLLIDER_HEIGHT

```
public static final float CHARGE_COLLIDER_HEIGHT
```

Stores the height of the zombie's charge collider in world units. The charge collider dictates the region where the zombie can hit the player while charging.

### NORMAL_WALK_SPEED

```
public static final float NORMAL_WALK_SPEED
```

Stores the walk speed of the zombie in the horizontal direction.

### COMBAT_WALK_SPEED

```
public static final float COMBAT_WALK_SPEED
```

Stores the walk speed of the zombie in the horizontal direction when in COMBAT mode.

### ALERTED_WALK_SPEED

```
public static final float ALERTED_WALK_SPEED
```

Holds the walking speed of the zombie when he is following the player.

### CHARGE_WALK_SPEED

```
public static final float CHARGE_WALK_SPEED
```

Stores the horizontal speed of the zombie when he is charging.

### DEFAULT_CHARGE_DAMAGE

```
public static final float DEFAULT_CHARGE_DAMAGE
```

Holds the default amount of damage the CHARGE attack deals to the player.

### ALERTED_ANIM_SPEED

```
public static final float ALERTED_ANIM_SPEED
```

Stores the multiplier of the zombie's walk animation when he is alerted and is following the player.

### JUMP_SPEED

```
public static final float JUMP_SPEED
```

Stores the jump speed of the zombie in the vertical direction.

### FALL_SPEED

```
public static final float FALL_SPEED
```

Stores the downwards speed at which the zombie falls through a TerrainLayer.

### INVULNERABLE_TIME

```
public static final float INVULNERABLE_TIME
```

Holds the amount of time that the zombie is invulnerable when hit.

### DEFAULT_HEALTH

```
public static final float DEFAULT_HEALTH
```

Holds the player's default health.

### alerted

```
private boolean alerted
```

Holds true if the Zombie is aware that the Player is within range of him. Makes him go towards the player.

### targetted

```
private boolean targetted
```

Stores true if the Zombie is being targetted by the player, and the player is trying to walk towards it.

### chargeCollider

```
private Rectangle chargeCollider
```

Stores the collider which goes around the zombie when he's charging. Determines if the zombie has hit the player.

### armCollider

```
private Rectangle armCollider
```

Holds the collider mapped to the zombie's arms. Used to dictate whether the zombie has hit a player with his arms.

### rightHandBone

```
private com.esotericsoftware.spine.Bone rightHandBone
```

Stores the bone which controls the zombie's right hand. Allows to compute the position and size of the arm's collider.

### leftHandBone

```
private com.esotericsoftware.spine.Bone leftHandBone
```

Stores the bone which controls the zombie's left hand. Allows to compute the position and size of the arm's collider.

### itemProbabilityMap

```
private java.util.HashMap<java.lang.Class,java.lang.Float>
itemProbabilityMap
```

Holds the HashMap of possible items that can be dropped from scavenging the Interactive GameObject. Key is the type of item, and Float is the probability of it being dropped from 0 (least probable) to 1 (most probable).

### animationState

```
private com.esotericsoftware.spine.AnimationState animationState
```

Controls the zombie's animations. Allows for crossfading between animations.

## Constructor Detail

### Zombie

```
public Zombie()
```

Creates a zombie whose bottom-center is at position (0, 0).

### Zombie

```
public Zombie(float x,
      float y)
```

Creates a zombie whose bottom-center is at position (x, y).

**Parameters:**

x - the center x-position of the zombie (in world units)

y - the bottom y-position of the zombie (in world units)

## Method Detail

### setupItemProbabilityMap

```
private void setupItemProbabilityMap()
```

Called on zombie creation in order to populate the HashMap which dictates the probability of certain items dropping when the zombie is killed.

### updateColliders

```
public void updateColliders()
```

Updates the various colliders mapped to the zombie.

### update

```
public void update(float deltaTime)
```

Updates the zombie's internal game logic.

**Overrides:**

`update` in class `Human`

**Parameters:**

`deltaTime` - the time elapsed since the last render call

### jump

```
public void jump()
```

Makes the zombie jump.

### fall

```
public void fall()
```

Makes the zombie fall through one layer.

### chargeHit

```
public void chargeHit(Player player)
```

Make the zombie charge hit the player.

**Parameters:**

`player` - the player to hit

### loseTarget

```
public void loseTarget()
```

Called when the zombie loses his target.

**Overrides:**

`loseTarget` in class `Human`

### canTarget

```
public boolean canTarget()
```

Override the canTarget method as always returning false since the Zombie can never be targetted.

**Specified by:**

`canTarget` in class `GameObject`

### setState

```
public void setState(Human.State state)
```

**Description copied from class: `Human`**

Sets the state (IDLE, JUMP, etc.) of the GameObject, used to dictate which animations to use. Also sets the stateTime of the GameObject back to zero.

**Overrides:**

`setState` in class `Human`

### getAnimationState

```
public com.esotericsoftware.spine.AnimationState getAnimationState()
```

Retrieves the Spine AnimationState instance used to change the zombie's animations and control them.

### setAnimationState

```
public void setAnimationState(com.esotericsoftware.spine.AnimationState animationState)
```

Sets the Spine AnimationState instance used to modify the zombie's animations and control them.

### isAlerted

```
public boolean isAlerted()
```

Returns true if the Zombie is aware that the Player is there.

### setAlerted

```
public void setAlerted(boolean alerted)
```

Sets whether or not the Zombie is aware of the player. If so, the Zombie walks towards the player

### makeInvulnerable

```
public void makeInvulnerable()
```

Makes the zombie invulnerable from the player's attacks for a given amount of seconds.

**Specified by:**

`makeInvulnerable` in class `Human`

### isTargetted

```
public boolean isTargetted()
```

Returns true if the zombie is being targetted by the player. If so, the player has clicked the zombie, and is walking towards it.

### setTargetted

```
public void setTargetted(boolean targetted)
```

Sets whether or not the zombie is being targetted by the player. If so, the player has clicked the zombie, and is walking towards it.

### getChargeCollider

```
public Rectangle getChargeCollider()
```

Gets the collider mapped to the zombie when he's charging. Used to determine if the zombie has charge hit the player.

### setChargeCollider

```
public void setChargeCollider(Rectangle chargeCollider)
```

Sets the collider mapped to the zombie when he's charging. Used to determine if the zombie has charge hit the player.

### getArmCollider

```
public Rectangle getArmCollider()
```

Returns the collider mapped to the zombie's arms. Used to dictate if the zombie melee hit the player.

### setArmCollider

```
public void setArmCollider(Rectangle armCollider)
```

Sets the collider mapped to the zombie's arms. Used to dictate if the zombie melee hit the player.

### getRightHandBone

```
public com.esotericsoftware.spine.Bone getRightHandBone()
```

Gets the bone mapped to the zombie's right hand. Allows to position the zombie's arm collider to dictate if the zombie hit the player.

### setRightHandBone

```
public void setRightHandBone(com.esotericsoftware.spine.Bone rightHand
Bone)
```

Sets the bone mapped to the zombie's right hand. Allows to position the zombie's arm collider to dictate if the zombie hit the player.

### getLeftHandBone

```
public com.esotericsoftware.spine.Bone getLeftHandBone()
```

Gets the bone mapped to the zombie's left hand. Allows to position the zombie's arm collider to dictate if the zombie hit the player.

### setLeftHandBone

```
public void setLeftHandBone(com.esotericsoftware.spine.Bone leftHandBo
ne)
```

Sets the bone mapped to the zombie's left hand. Allows to position the zombie's arm collider to dictate if the zombie hit the player.

### getItemProbabilityMap

```
public java.util.HashMap<java.lang.Class,java.lang.Float> getItemProba
bilityMap()
```

Returns the HashMap which holds which items will be dropped when this InteractiveObject is scavenged. The key is the type of the Item dropped and the float is the probability of it dropping from [0,1].

### setItemProbabilityMap

```
public void setItemProbabilityMap(java.util.HashMap<java.lang.Class,ja
va.lang.Float> itemProbabilityMap)
```

Sets the HashMap which holds which items will be dropped when this InteractiveObject is scavenged. The key is the type of the Item dropped and the float is the probability of it dropping from [0,1].

**reset**

```
public void reset()
```

Called when the Zombie instance is put back into a pool. All his data fields must be reset to default.

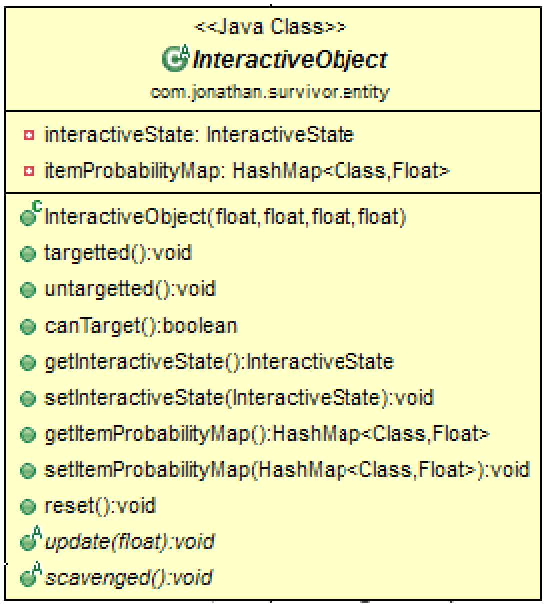
**Specified by:**

`reset` in interface `com.badlogic.gdx.utils.Pool.Poolable`

**Overrides:**

`reset` in class `Human`

II.3.1.4 InteractiveObject



<<Java Class>>
InteractiveObject
com.jonathan.survivor.entity
interactiveState: InteractiveState
itemProbabilityMap: HashMap<Class,Float>
InteractiveObject(float,float,float,float)
targetted():void
untargetted():void
canTarget():boolean
getInteractiveState():InteractiveState
setInteractiveState(InteractiveState):void
getItemProbabilityMap():HashMap<Class,Float>
setItemProbabilityMap(HashMap<Class,Float>):void
reset():void
update(float):void
scavenged():void

**Figure 20 : InteractiveObject class diagram**

An *InteractiveObject* represents a *GameObject* that can be scavenged by the player for resources, such as a tree or a box. The *interactiveState* dictates the state of this *GameObject*. For instance, it can be *IDLE* if the object is just standing there, or *SCAVENGED* if the player has already scavenged the object. For instance, a tree enters the *SCAVENGED* state when it is chopped down by the player.

The *itemProbabilityMap* is a HashMap with a Class as a key, and a Float as a value. The key holds an *Item* subclass (see II.3 Classes)**.** The value mapped to the class is a number from *0* to *1.0*, which indicates the probability of that item class from being spawned when the *InteractiveObject* is scavenged. For instance, if *itemProbabilityMap.get(Wood.class)* is equal to *0.5*, then there is a 50% chance that the *InteractiveObject* will drop wood when scavenged.

The constructor accepts four floats: the width and height of the object's collider, along with the x and y positions of the object upon instantiation.

The *targetted()* method is called whenever the *InteractiveObject* is targeted by a human. In such as case, the object knows to transition to its *TARGET* state. The *untargeted()* method is called when this object has been lost as a target. In this case, the object is set back to *IDLE* state.

The *update(float)* method is called every game tick in order to update the object's position and *stateTime*. The abstract *scavenged()* method is called when the *InteractiveObject* has been scavenged by the player.

```
public abstract class InteractiveObject
extends GameObject
implements Clickable, com.badlogic.gdx.utils.Pool.Poolable
```

## Field Detail

### interactiveState

```
private InteractiveObject.InteractiveState interactiveState
```

Stores the current state of the interactive object for logic and rendering purposes.

### itemProbabilityMap

```
private java.util.HashMap<java.lang.Class,java.lang.Float>
itemProbabilityMap
```

Holds the HashMap of possible items that can be dropped from scavenging the Interactive GameObject. Key is the type of item, and Float is the probability of it being dropped from 0 (least probable) to 1 (most probable).

## Constructor Detail

### InteractiveObject

```
public InteractiveObject(float x, float y, float width, float height)
```

Creates the Interactive GameObject with the given bottom-center position and the given collider width and height.

**Parameters:**

`x` - the center x-position (in world units)

`y` - the center y-position (in world units)

`width` - the width of the rectangle collider

`height` - the height of the rectangle collider

## Method Detail

### targetted

```
public void targetted()
```

Called when the Interactive GameObject has just been targetted by a Human

### untargetted

```
public void untargetted()
```

Called when the Interactive GameObject has just been untargetted by a Human

### canTarget

```
public boolean canTarget()
```

Returns true if the Interactive GameObject can be targetted by the player.

**Specified by:**

`canTarget` in class `GameObject`

### getInteractiveState

```
public InteractiveObject.InteractiveState getInteractiveState()
```

Gets the current state of the interactive object for logic and rendering purposes.

### setInteractiveState

```
public void setInteractiveState(InteractiveObject.InteractiveState interactiveState)
```

Sets the current state of the interactive object for logic and rendering purposes.

### getItemProbabilityMap

```
public java.util.HashMap<java.lang.Class,java.lang.Float> getItemProbabilityMap()
```

Returns the HashMap which holds which items will be dropped when this InteractiveObject is scavenged. The key is the type of the Item dropped and the float is the probability of it dropping from [0,1].

### setItemProbabilityMap

```
public void setItemProbabilityMap(java.util.HashMap<java.lang.Class,java.lang.Float> itemProbabilityMap)
```

Sets the HashMap which holds which items will be dropped when this InteractiveObject is scavenged. The key is the type of the Item dropped and the float is the probability of it dropping from [0,1].

### reset

```
public void reset()
```

Called whenever this GameObject has been pushed back into a pool. In this case, we reset the box's state back to default.

**Specified by:**

`reset` in interface `com.badlogic.gdx.utils.Pool.Poolable`

### update

```
public abstract void update(float deltaTime)
```

Called every frame to update logic.

**Specified by:**

`update` in class `GameObject`

**Parameters:**

`deltaTime` - the execution time of the previous render call

### scavenged

```
public abstract void scavenged()
```

Called when the Interactive GameObject has been scavenged and can no longer be targetted.

II.3.1.5 Tree & Box

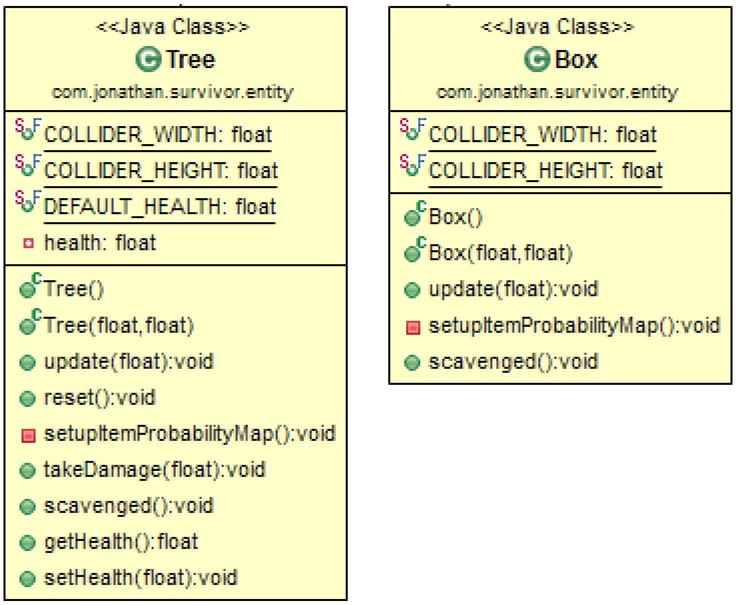


**Figure 21 : Tree and Box class diagrams**

A *Tree* represents a physical tree in the world, whereas the *Box* class represents a physical box. They bot hhave public static final *COLLIDER_WIDTH:float* and *COLLIDER_HEIGHT:float* constants, which indicate the size of each of their colliders. The *Tree* has his default health as a constant, along with a private *health* member variable, which indicates how much damage the tree can receive from the player before being chopped down.

```
public class Box
extends InteractiveObject
implements com.badlogic.gdx.utils.Pool.Poolable
```

### Field Detail

### COLLIDER_WIDTH

```
public static final float COLLIDER_WIDTH
```

Stores the width of a box's rectangle collider in world units.

### COLLIDER_HEIGHT

```
public static final float COLLIDER_HEIGHT
```

Stores the height of a box's rectangle collider in world units.

## Constructor Detail

### Box

```
public Box()
```

Creates a box whose bottom-center is at position (0, 0).

### Box

```
public Box(float x, float y)
```

Creates a box whose bottom-center is at position (x, y).

**Parameters:**

x - the center x-position of the box (in world units)

y - the center y-position of the box (in world units)

## Method Detail

### update

```
public void update(float deltaTime)
```

Updates the box every frame.

**Specified by:**

`update` in class `InteractiveObject`

**Parameters:**

`deltaTime` - the execution time of the previous render call

### setupItemProbabilityMap

```
private void setupItemProbabilityMap()
```

Called on box creation in order to populate the HashMap which dictates the probability of certain items dropping when the box is destroyed.

### scavenged

```
public void scavenged()
```

Called when the box has been opened by the player. Tells the box to enter its SCAVENGED state.

**Specified by:**

> `scavenged` in class `InteractiveObject`

```
public class Tree
extends InteractiveObject
```

## Field Detail

### COLLIDER_WIDTH

```
public static final float COLLIDER_WIDTH
```

Stores the width of a tree's rectangle collider in world units.

### COLLIDER_HEIGHT

```
public static final float COLLIDER_HEIGHT
```

Stores the height of a tree's rectangle collider in world units.

### DEFAULT_HEALTH

```
public static final float DEFAULT_HEALTH
```

Stores the default health of the tree.

### health

```
private float health
```

Stores the tree's health. Once it drops below zero, it is destroyed.

### Constructor Detail

#### Tree

```
public Tree()
```

Creates a tree whose bottom-center is at position (0, 0).

#### Tree

```
public Tree(float x, float y)
```

Creates a tree whose bottom-center is at position (x, y).

**Parameters:**

x - the center x-position (in world units)

y - the center y-position (in world units)

### Method Detail

#### update

```
public void update(float deltaTime)
```

Updates the tree every frame.

**Specified by:**

`update` in class `InteractiveObject`

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

#### reset

```
public void reset()
```

Called whenever this tree GameObject has been pushed back into a pool. In this case, we reset the tree's state and health back to default.

**Specified by:**

`reset` in interface `com.badlogic.gdx.utils.Pool.Poolable`

**Overrides:**

`reset` in class `InteractiveObject`

### setupItemProbabilityMap

```
private void setupItemProbabilityMap()
```

Called on tree creation in order to populate the HashMap which dictates the probability of certain items dropping when the tree is destroyed.

### takeDamage

```
public void takeDamage(float damage)
```

Deals damage to the tree by removing the given amount from its health.

**Parameters:**

`damage` - the amount of damage to deal

### scavenged

```
public void scavenged()
```

Called when the tree's health has been depleted to zero and has been scavenged.

**Specified by:**

`scavenged` in class `InteractiveObject`

### getHealth

```
public float getHealth()
```

Gets the tree's health.

### setHealth

```
public void setHealth(float health)
```

Sets the tree's health.

### II.3.1.6 *Projectile*

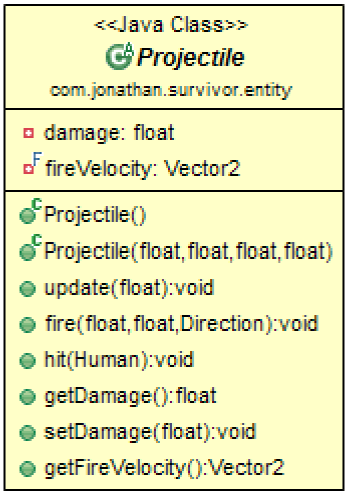


**Figure 22: Projectile class diagram**

A *Projectile* models a *GameObject* which is fired at a certain velocity. It has a collider of its own, and, upon colliding with another *GameObject*, a projectile will inflict damage to the *GameObject* with which it collides.

```
public abstract class Projectile
extends GameObject
```

## Field Detail

### damage

```
private float damage
```

Stores the amount of damage the projectile deals when colliding with a Human.

### fireVelocity

```
private final Vector2 fireVelocity
```

Holds the velocity at which the projectile will be fired. Note that this is in the right direction. The vector is transformed to point to the left when desired.

## Constructor Detail

### Projectile

```
public Projectile()
```

Spawns a projectile at position (0,0) with collider size zero.

### Projectile

```
public Projectile(float x, float y, float width, float height)
```

Creates a Projectile with the given (x,y) coordinates and the given collider size.

**Parameters:**

x - the center x-position (in world units)

y - the center y-position (in world units)

`width` - the width of the rectangle collider

`height` - the height of the rectangle collider

## Method Detail

### update

```
public void update(float deltaTime)
```

Updates the Projectile's game logic.

**Specified by:**

`update` in class `GameObject`

**Parameters:**

`deltaTime` - the execution time of the previous frame

### fire

```
public void fire(float x, float y, Human.Direction direction)
```

Fires the projectile at the given bottom-center (x,y) position and in the given direction.

**Parameters:**

x - the center x-position at which to fire the projectile

y - the bottom y-position at which to fire the projectile `direction` - the direction to fire the projectile

### hit

```
public void hit(Human human)
```

Called when the projectile hits a Human. Deals damage to the hit Human.

**Parameters:**

`human` - the human to hit

### getDamage

```
public float getDamage()
```

Gets the amount of damage dealt by the projectile when hitting a Human instance.

### setDamage

```
public void setDamage(float damage)
```

Sets the amount of damage dealt by the projectile when hitting a Human instance.

### getFireVelocity

```
public Vector2 getFireVelocity()
```

Gets the velocity at which the projectile is fired when the fire() method is called.

### II.3.1.7 *Earthquake*

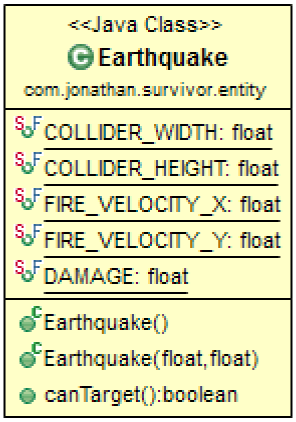


**Figure 23: Earthquake class diagram**

An *Earthquake* instance is spawned whenever the zombie performs a smash attack. It travels towards the player with a certain velocity, and, upon collision, deals damage to the *GameObject* which which it collides.

```
public class Earthquake
extends Projectile
```

**Field Detail**

**COLLIDER_WIDTH**

```
public static final float COLLIDER_WIDTH
```

Stores the width of the Earthquake's rectangle collider in world units. (Smaller than actual image so that collisions are forgiving)

**COLLIDER_HEIGHT**

```
public static final float COLLIDER_HEIGHT
```

Stores the height of the Earthquake's rectangle collider in world units. (Smaller than actual image so that collisions are forgiving)

**FIRE_VELOCITY_X**

```
public static final float FIRE_VELOCITY_X
```

Holds the velocity at which the earthquake travels.

### FIRE_VELOCITY_Y

```
public static final float FIRE_VELOCITY_Y
```

Holds the velocity at which the earthquake travels.

### DAMAGE

```
public static final float DAMAGE
```

Stores the amount of damage dealt by the Earthquake when hitting a Human.

## Constructor Detail

### Earthquake

```
public Earthquake()
```

Creates a default Earthquake instance at bottom-center position (0,0).

### Earthquake

```
public Earthquake(float x, float y)
```

Instantiates an Earthquake instance at the given bottom-center (x,y) position.

**Parameters:**

x - the center x-position of the earthquake (in world units)

y - the center y-position of the earthquake (in world units)

## Method Detail

### canTarget

```
public boolean canTarget()
```

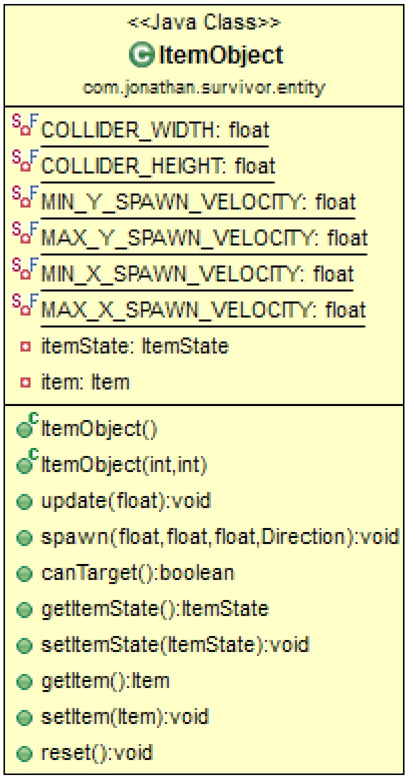
**Description copied from class: `GameObject`**

Returns true if the GameObject can be targetted by a Human.

**Specified by:**

`canTarget` in class `GameObject`

### II.4.1.8 *ItemObject*



<<Java Class>>
ItemObject
com.jonathan.survivor.entity

COLLIDER_WIDTH: float
COLLIDER_HEIGHT: float
MIN_Y_SPAWN_VELOCITY: float
MAX_Y_SPAWN_VELOCITY: float
MIN_X_SPAWN_VELOCITY: float
MAX_X_SPAWN_VELOCITY: float
itemState: ItemState
item: Item

ItemObject()
ItemObject(int,int)
update(float):void
spawn(float,float,float,Direction):void
canTarget():boolean
getItemState():ItemState
setItemState(ItemState):void
getItem():Item
setItem(Item):void
reset():void

**Figure 24 : ItemObject class diagrams**

An *ItemObject* is a GameObject which displays an item. That is, when a player chops down a tree, for instance, *ItemObjects* are spawned next to the tree. Subsequently, the player can pick up these *ItemObjects* by clicking on them and gain an item in his inventory.

The class's COLLIDER_WIDTH:float and COLLIDER_HEIGHT:float constants indicate the size of the GameObject's rectangle collider. This is the region the player can press on the object in order to collect it. The next four constants specify the minimum and maximum (x,y) velocity of an *ItemObject*. This is needed when the item is spawned in the world, as it is launched in the air in a parabolic arc. These constants determine the initial speed of the item when it is spawned.

Next, the *itemState* variable holds the state of the *ItemObject*. For example, if the item was just spawned, it will be in *ItemState.SPAWN* state. Finally, and most importantly, the *itemClass* instance variable holds the class which the *ItemObject* represents. This is because an item is represented as two different things. First, each *Item* instance represents the data for an item. It defines the name and properties of a specific item. An *ItemObject* instance, on the other hand, is a physical representation of such an item. Thus, if *itemClass* holds the value of *Iron.class*, then the *ItemObject* displays a picture of an iron plate.

The default constructor spawns the GameObject at position (0,0), whereas the constructor accepting two floats accepts the (x,y) position of the *ItemObject*. The most important method is the *spawn(Class, float, float, Direction):void* method. This method is called whenever the *ItemObject* should be spawned in the world. The first parameter accepts the class of the item that wants to be spawned. The next two floats accept the (x,y) position where the *ItemObject* should spawn. The last indicates whether the *ItemObject* should fly towards the left or the right when spawned.

```
public class ItemObject
extends GameObject
implements com.badlogic.gdx.utils.Pool.Poolable
```

### Field Detail

#### COLLIDER_WIDTH

```
private static final float COLLIDER_WIDTH
```

Stores the width of an item GameObject's collider. This is the touchable region of the item. All Item GameObjects have the same collider size.

#### COLLIDER_HEIGHT

```
private static final float COLLIDER_HEIGHT
```

Stores the height of an item GameObject's collider. This is the touchable region of the item. All Item GameObjects have the same collider size

#### DEFAULT_SPRITE_WIDTH

```
private static final float DEFAULT_SPRITE_WIDTH
```

Stores the default width of an item sprite

#### DEFAULT_SPRITE_HEIGHT

```
private static final float DEFAULT_SPRITE_HEIGHT
```

Stores the default height of an item sprite

#### MIN_Y_SPAWN_VELOCITY

```
private static final float MIN_Y_SPAWN_VELOCITY
```

Holds the minimum y-velocity of the Item GameObject when it is spawned.

#### MAX_Y_SPAWN_VELOCITY

```
private static final float MAX_Y_SPAWN_VELOCITY
```

Holds the maximum y-velocity of the Item GameObject when it is spawned.

#### MIN_X_SPAWN_VELOCITY

```
private static final float MIN_X_SPAWN_VELOCITY
```

Holds the minimum x-velocity of the Item GameObject when it is spawned.

#### MAX_X_SPAWN_VELOCITY

```
private static final float MAX_X_SPAWN_VELOCITY
```

Holds the maximum x-velocity of the Item GameObject when it is spawned.

#### itemState

```
private ItemObject.ItemState itemState
```

Stores the state of the item, which determines the animation it plays.

#### item

```
private Item item
```

Stores the Item held by the GameObject.

### Constructor Detail

#### ItemObject

```
public ItemObject()
```

Creates an ItemObject at bottom-center position (0,0).

### ItemObject

```
public ItemObject(int x, int y)
```

Creates an Item GameObject at the given bottom-center (x,y) position.

**Parameters:**

x - the center x-position where to place the ItemObject

y - the bottom x-position where to place the ItemObject

## Method Detail

### update

```
public void update(float deltaTime)
```

**Description copied from class: `GameObject`**

Updates the GameObject's game logic.

**Specified by:**

`update` in class `GameObject`

**Parameters:**

`deltaTime` - the amount of time the last render call took to complete

### updateCollider

```
public void updateCollider()
```

**Description copied from class: `GameObject`**

Snaps the GameObject's collider to the GameObject's position

**Overrides:**

`updateCollider` in class `GameObject`

**See Also:**

`Overriden to ensure that the ItemObject's collider is well centered at the object's position`

### spawn

```
public <T extends Item> void spawn(float x, float y,
                float velocityMultiplier, Human.Direction direction)
```

Spawns the item at the given position. Gives the item a random upward velocity to simulate confetti explosion. The position is the bottom-center position of the gameObject.

**Parameters:**

x - Center x-position where the item is spawned

y - Bottom y-position where the item is spawned

`velocityMultiplier` - Scalar by which velocity is multiplied, to allow certain items to fly further. Allows items to be spread apart if many are spawned.

`direction` - Specifies the direction in which the items fly when spawned

### canTarget

```
public boolean canTarget()
```

**Description copied from class: `GameObject`**

Returns true if the GameObject can be targetted by a Human.

**Specified by:**

`canTarget` in class `GameObject`

### getItemState

```
public ItemObject.ItemState getItemState()
```

Gets the ItemState which determines which animation the object should be playing when dropped into the world.

### setItemState

```
public void setItemState(ItemObject.ItemState itemState)
```

Sets the ItemState which determines which animation the object should be playing when dropped into the world.

### getItem

```
public Item getItem()
```

Gets the item represented by the ItemObject.

### setItem

```
public void setItem(Item item)
```

Sets the item the GameObject contains and displays.

### reset

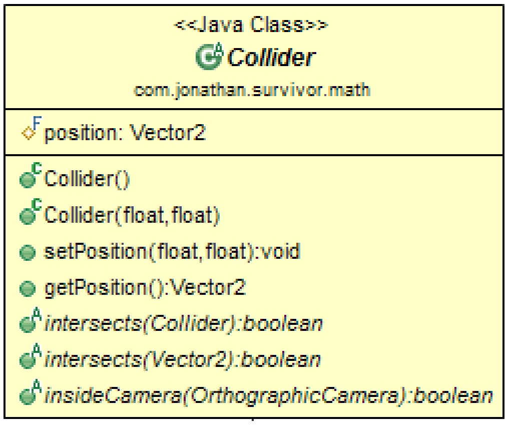
```
public void reset()
```

Called when an ItemObject is placed back into a pool. The GameObject must be reset to default configuration.

**Specified by:**

`reset` in interface `com.badlogic.gdx.utils.Pool.Poolable`

### II.3.1.9 *Collider*



<<Java Class>>
Collider
com.jonathan.survivor.math
position: Vector2
Collider()
Collider(float,float)
setPosition(float,float):void
getPosition():Vector2
intersects(Collider):boolean
intersects(Vector2):boolean
insideCamera(OrthographicCamera):boolean

**Figure 25 : Collider class diagram**

A *Collider* represents a geometric shape bound to a *GameObject*. They are used to detect collisions between one or more entities in the world. First, a *Collider* has a position denoted by a *Vector2*. In the case of a rectangle, the (x,y) position represents the bottom-left point of the shape.

The *Collider* has two different constructors. The first intializes the *position* of the *Collider* to (0,0), and the second places the *Collider* at the (x,y) position given by the two parameters of the method. Then, the abstract *intersects(Collider)* method accepts a *Collider* instance as a parameter, and returns true if the two *Colliders* intersect. The next overloaded *intersects(Vector2)* method accepts a point on the screen denoted as a *Vector2*. It returns true if the point is within the collider. Next, the *insideCamera(OrthographicCamera)* method returns *true* if the collider intersects with the given camera. It is used for camera culling, and is explained in detail in *section II.1 Algorithms*.

```
public abstract class Collider
extends java.lang.Object
```

**Field Detail**

**position**

```
protected final Vector2 position
```

Stores the position of the collider. If a rectangle collider is used, this position specifies the lower-left position of the rectangle.

## Constructor Detail

### Collider

```
public Collider()
```

Creates a collider placed at (0,0).

### Collider

```
public Collider(float x, float y)
```

Creates a collider placed at the desired (x,y) coordinates.

**Parameters:**

x - the x-position of the collider

y - the y-position of the collider

## Method Detail

### setPosition

```
public void setPosition(float x, float y)
```

Sets the position of the collider.

**Parameters:**

x - the new x-position of the collider

y - the new y-position of the collider

### getPosition

```
public Vector2 getPosition()
```

Returns the position of the collider. Since Vector2s are mutable, the position can be changed using the Vector2's instance methods.

### intersects

```
public abstract boolean intersects(Collider r)
```

Returns true if this collider intersects with another collider.

**Parameters:**

`r` - the collider to test intersection with

**Returns:**

true, if this collider intersects the given collider.

### intersects

```
public abstract boolean intersects(Vector2 point)
```

Returns true if this point is within the collider.

**Parameters:**

`point` - the point which is tested for instersection with the collider.

**Returns:**

true, if the collider intersects the given point.

### insideCamera

```
public
abstract boolean insideCamera(com.badlogic.gdx.graphics.OrthographicCa
mera cam)
```

Returns true if this collider is inside the camera.

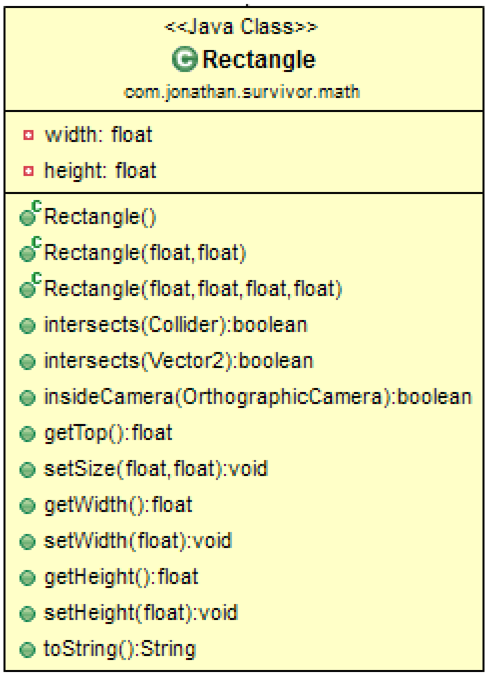
**Parameters:**

`cam` - the camera which is tested to be able to view the collider.

**Returns:**

true, if the collider is inside the viewable region of the camera.

### II.3.1.10 *Rectangle*



<<Java Class>>
Rectangle
com.jonathan.survivor.math

width: float
height: float

Rectangle()
Rectangle(float,float)
Rectangle(float,float,float,float)
intersects(Collider):boolean
intersects(Vector2):boolean
insideCamera(OrthographicCamera):boolean
getTop():float
setSize(float,float):void
getWidth():float
setWidth(float):void
getHeight():float
setHeight(float):void
toString():String

**Figure 26 : Rectangle class diagram**

The *Rectangle* class extends the *Collider* class. It is used to represent a bounding box around a *GameObject*. It is the only type of *Collider* used in *Free the Bob*. The class's behaviour is straight-forward, so its explanation will be brief. First, a *Rectangle* object has a width and a height. The default constructor instantiates a *Rectangle* at bottom-left position (0,0), and with width and height zero. The second constructor accepts the width and height of the *Rectangle*, and the third accepts the (x,y) position of the *Rectangle*, along with its width and height.

```
public class Rectangle
extends Collider
```

**Field Detail**

**width**

```
private float width
```

Stores the width of the rectangle

**height**

```
private float height
```

Stores the height of the rectangle

## Constructor Detail

### Rectangle

```
public Rectangle()
```

Creates a rectangle at lower-left position (0,0) with width/height of zero

### Rectangle

```
public Rectangle(float width, float height)
```

Creates a rectangle at lower-left position (0,0) with given width/height.

**Parameters:**

`width` - the width of the rectangle

`height` - the height of the rectangle

### Rectangle

```
public Rectangle(float x, float y, float width, float height)
```

Creates a rectangle at lower-left position (x,y) with given width/height.

**Parameters:**

`x` - the center x-position of the rectangle

`y` - the bottom y-position of the rectangle

`width` - the width of the rectangle

`height` - the height of the rectangle

## Method Detail

### intersects

```
public boolean intersects(Collider c)
```

Returns true if this rectangle intersects with another collider.

**Specified by:**

intersects in class Collider

**Parameters:**

c - the collider to test intersection for

**Returns:**

true, if the rectangle intersects the given collider

### intersects

```
public boolean intersects(Vector2 point)
```

Returns true if this point is within the rectangle.

**Specified by:**

intersects in class Collider

**Parameters:**

point - the point to test intersection for

**Returns:**

true, if the rectangle intersects the given point

### insideCamera

```
public boolean insideCamera(com.badlogic.gdx.graphics.OrthographicCame
ra camera)
```

Returns true if this rectangle collider is inside the bounds of the camera. Used for camera culling.

**Specified by:**

insideCamera in class Collider

**Parameters:**

camera - the camera which is tested to be able to view the rectangle.

**Returns:**

true, if the rectangle is inside the viewable region of the camera.

### getTop

```
public float getTop()
```

Returns the y-position of the top of the collider in world units.

### setSize

```
public void setSize(float width, float height)
```

Sets the width and height of the rectangle from its bottom-left position.

**Parameters:**

`width` - the width of the rectangle

`height` - the height of the rectangle

### getWidth

```
public float getWidth()
```

Retrieves the width of the rectangle

### setWidth

```
public void setWidth(float width)
```

Sets the width of the rectangle

### getHeight

```
public float getHeight()
```

Gets the height of the rectangle

### setHeight

```
public void setHeight(float height)
```

Sets the height of the rectangle

### toString

```
public java.lang.String toString()
```

Returns a string representation of the rectangle, for debugging purposes.

**Overrides:**

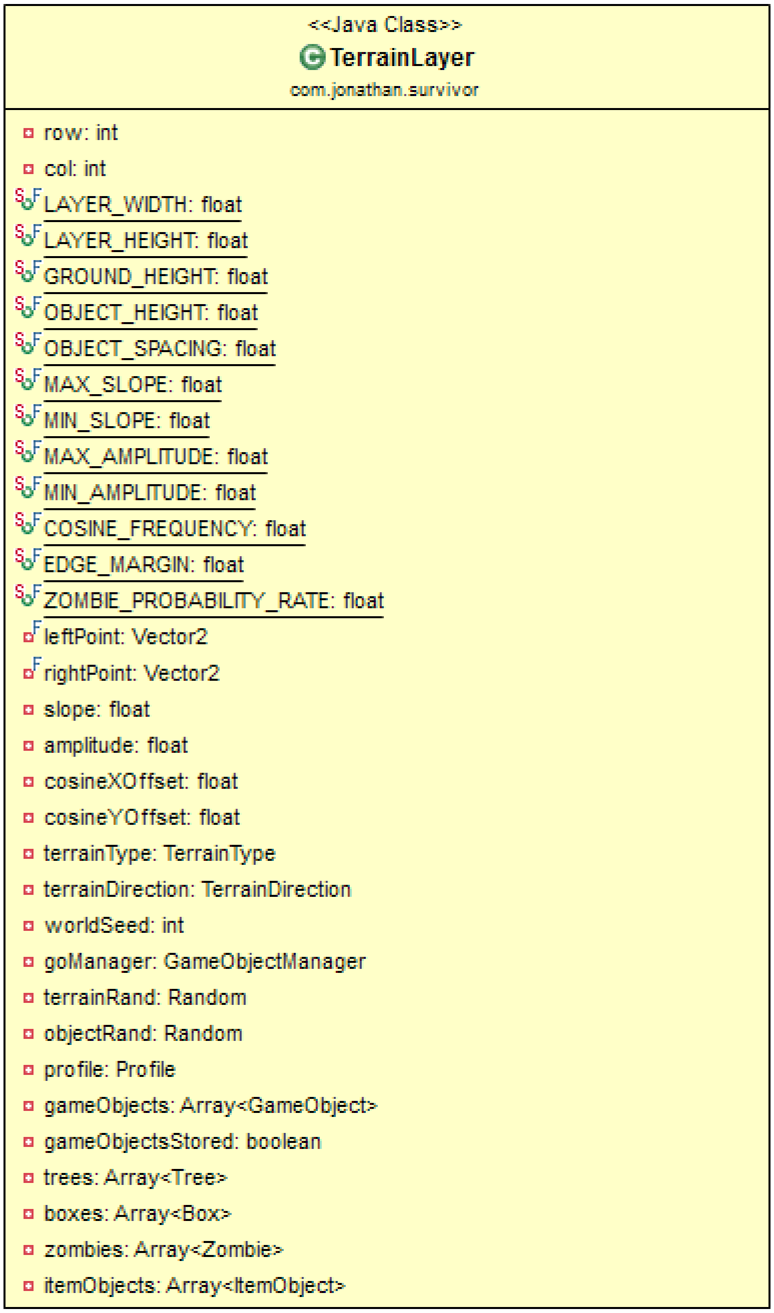
`toString` in class `java.lang.Object`

**See Also:**

`Object.toString()`

(See next page for *TerrainLayer*)

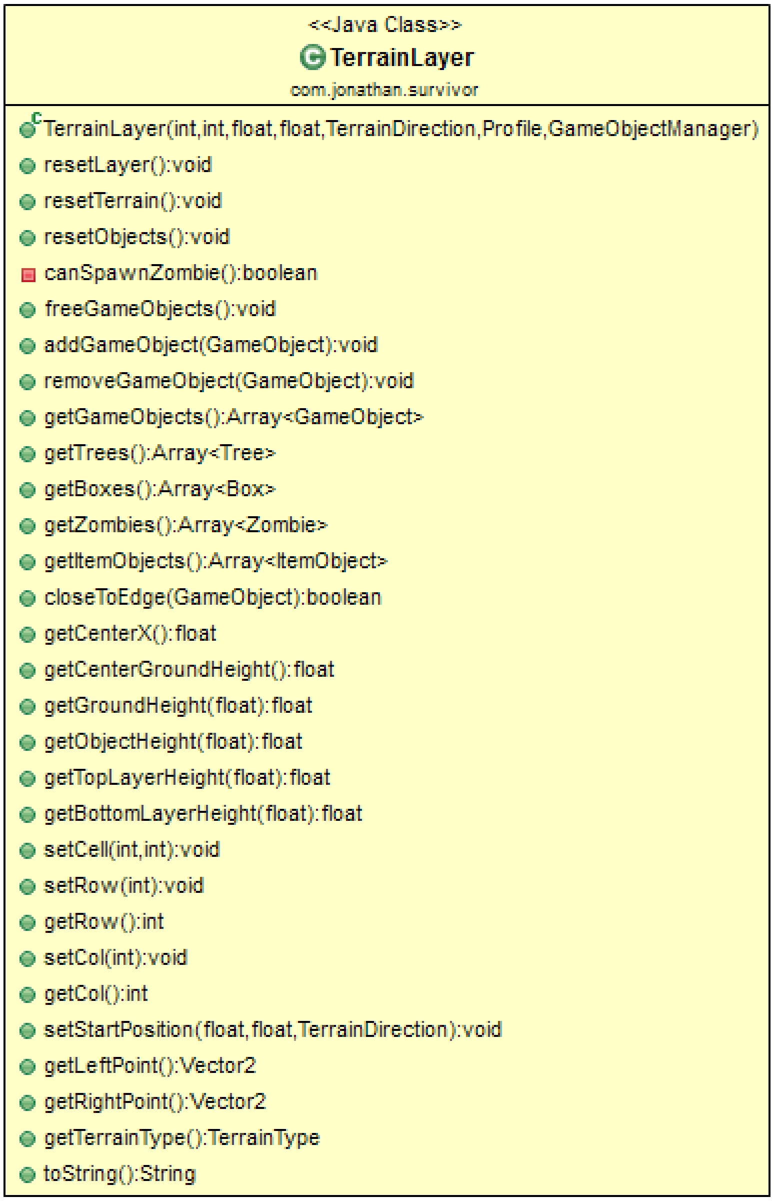
### II.3.1.11 *TerrainLayer*



<<Java Class>>
TerrainLayer
com.jonathan.survivor

row: int
col: int
LAYER_WIDTH: float
LAYER_HEIGHT: float
GROUND_HEIGHT: float
OBJECT_HEIGHT: float
OBJECT_SPACING: float
MAX_SLOPE: float
MIN_SLOPE: float
MAX_AMPLITUDE: float
MIN_AMPLITUDE: float
COSINE_FREQUENCY: float
EDGE_MARGIN: float
ZOMBIE_PROBABILITY_RATE: float
leftPoint: Vector2
rightPoint: Vector2
slope: float
amplitude: float
cosineXOffset: float
cosineYOffset: float
terrainType: TerrainType
terrainDirection: TerrainDirection
worldSeed: int
goManager: GameObjectManager
terrainRand: Random
objectRand: Random
profile: Profile
gameObjects: Array<GameObject>
gameObjectsStored: boolean
trees: Array<Tree>
boxes: Array<Box>
zombies: Array<Zombie>
itemObjects: Array<ItemObject>

**Figure 27 : TerrainLayer class diagram (data fields)**



<<Java Class>>

**TerrainLayer**

com.jonathan.survivor

● TerrainLayer(int,int,float,float,TerrainDirection,Profile,GameObjectManager)<br>
● resetLayer():void<br>
● resetTerrain():void<br>
● resetObjects():void<br>
■ canSpawnZombie():boolean<br>
● freeGameObjects():void<br>
● addGameObject(GameObject):void<br>
● removeGameObject(GameObject):void<br>
● getGameObjects():Array<GameObject><br>
● getTrees():Array<Tree><br>
● getBoxes():Array<Box><br>
● getZombies():Array<Zombie><br>
● getItemObjects():Array<ItemObject><br>
● closeToEdge(GameObject):boolean<br>
● getCenterX():float<br>
● getCenterGroundHeight():float<br>
● getGroundHeight(float):float<br>
● getObjectHeight(float):float<br>
● getTopLayerHeight(float):float<br>
● getBottomLayerHeight(float):float<br>
● setCell(int,int):void<br>
● setRow(int):void<br>
● getRow():int<br>
● setCol(int):void<br>
● getCol():int<br>
● setStartPosition(float,float,TerrainDirection):void<br>
● getLeftPoint():Vector2<br>
● getRightPoint():Vector2<br>
● getTerrainType():TerrainType<br>
● toString():String

**Figure 28 : TerrainLayer class diagram (constructors and methods)**

A *TerrainLayer* is a piece of terrain on which the player can walk. It is represented as either a constant, linear, or cosine function. Since a lot of its functionality was explained in *section II.1 Algorithms*, the explanation of this class diagram will be shortened.

First, each instance of this class has a row and a column relative to the world. From this cell coordinate dictates the layer's geometry and the objects that are placed on top of it.

On a separate note, the class has a *leftPoint* and a *rightPoint* Vector2. They store the coordinates of the bottom-left end point and the bottom-right end point of the layer. These Vector2s dictate the points where the layer's function should start and end. Next, if the layer is a linear function, it will have a *slope*, stored inside a data field. Conversely, if the layer is a cosine function, it will have an *amplitude*, stored inside another data field.

Further, the class has a *terrainType* variable, which holds a constant from the *TerrainType* enumeration (either CONSTANT, LINEAR, or COSINE). The type of the terrain dictates how its geometry is defined.

Next, the class has a *worldSeed* integer, which it uses to randomly generate numbers that will define the layer's terrain, along with the objects placed on top of it. Additionally, the *TerrainLayer* has access to a *gameObjectManager* variable, which allows the class to retrieve pooled *GameObjects* to place on top of the layer.

The first *Random* instance, *terrainRand*, is used to randomly generate numbers that will determine the geometry of the layer. For instance, it will dictate the slope of a linear layer or the amplitude of a cosine function. Conversely, the *objectRand* variable generates numbers that will create and place objects on the layer.

The class also has a *profile* variable, from which it accesses a list of all the objects that have already been scavenged on the layer. This prevents the layer from placing objects on itself that have already been destroyed or scavenged by the player. Moreover, the class has an array of *GameObjects*. This array holds all of the *GameObjects* which are placed in the *TerrainLayer*. It allows the world to fetch all of the *GameObjects* from each layer, update them and render them.

The constructor of a *TerrainLayer* first accepts its row and column in the world, followed by its x and y position, the *Profile* and the *GameObjectManager* instances from which the class's data fields are populated.

In terms of methods, *resetLayer()* is called whenever the layer's geometry and objects need to be redefined. It is called when the layer shifts places inside the *TerrainLevel's* matrix and changes cell coordinates. In turn, *resetLayer()* calls the *resetLayer()* and *resetObjects()* methods.

The former generates the layer's geometry, while the latter places *GameObjects* on the layer.

Moreover, the *addGameObject(GameObject):void* method is called whenever an *ItemObject* is dropped in the world. When this happens, it is passed as an argument to the method and added the *TerrainLayer's gameObjects* array. As such, the *ItemObject* will belong to the layer. Conversely, *freeGameObjects()* frees all of the *GameObjects* inside the *GameObject* array back into the *GameObjectManager's* internal *poolMap*. As such, the *GameObjects* will be recycled for another layer to use. The *getGroundHeight(float)* method accepts any x-position. It returns the y-position of the ground at that specific x-position. It allows objects to know the position of the ground. To function, it simply plugs in the x-position into the layer's constant, linear  or cosine function which in turn returns the y-point at that position. The *getBottomLayerHeight(float):float* acts very similarly, except that it returns the y-position of the bottom of the layer. This is the part of the layer that is drawn. Thus, this method is used to render lines that represent the bottom of the layer. The *getTopLayerHeight(float)* returns the same y-position, plus a certain height. This height is the vertical distance between two layers that are stacked on top of each other.

Finally, the last method that needs explaining is the *setCell(row:int, col:int):void*, which simply sets the row and the column of the layer to the given parameters. It also calls the *resetLayer()* method to re-initialize the layer according to its new cell coordinates.

## Field Detail

### row

```
private int row
```

Stores the row and column of the layer. Note that the row defines the layer's geometry.

### col

```
private int col
```

Stores the row and column of the layer. Note that the row defines the layer's geometry.

### LAYER_WIDTH

```
public static final float LAYER_WIDTH
```

Stores the width of a layer. The width is measured from the left end of the layer to the right end. The height goes from the bottom y-position to the top y-position of any x-position on the layer. If it is a cosine layer, it does not go from the bottom-most point to the top-most point; it is the same regardless.

### LAYER_HEIGHT

```
public static final float LAYER_HEIGHT
```

Stores the height of a layer. The height goes from the bottom y-position to the top y-position of any x-position on the layer. If it is a cosine layer, it does not go from the bottom-most point to the top-most point; it is the same regardless.

### GROUND_HEIGHT

```
public static final float GROUND_HEIGHT
```

Stores the height from any bottom point of the layer to its ground point where the user walks

### OBJECT_HEIGHT

```
public static final float OBJECT_HEIGHT
```

Stores the bottom height of objects on the layer. Increasing this places objects higher on the layer.

### OBJECT_SPACING

```
public static final float OBJECT_SPACING
```

Stores the minimum horizontal spacing between GameObjects on a layer.

### MAX_SLOPE

```
public static final float MAX_SLOPE
```

Stores the maximum slope a linear layer can have

### MIN_SLOPE

```
public static final float MIN_SLOPE
```

Stores the minimum slope a linear layer can have

### MAX_AMPLITUDE

```
public static final float MAX_AMPLITUDE
```

Stores the maximum amplitude a cosine layer can have. (Note that amplitude = half-height of cosine function)

### MIN_AMPLITUDE

```
public static final float MIN_AMPLITUDE
```

Stores the minimum amplitude a cosine layer can have. (Note that amplitude = half-height of cosine function)

### COSINE_FREQUENCY

```
public static final float COSINE_FREQUENCY
```

Stores the b-value in a cosine function, the frequency, found by 2pi/period, where the period is the width of the cosine function.

### EDGE_MARGIN

```
public static final float EDGE_MARGIN
```

Holds the distance in meters. Determines how close a GameObject has to be to the layer's edge be considered "near" the edge. Used in closeToEdge(...)

### ZOMBIE_PROBABILITY_RATE

```
public static final float ZOMBIE_PROBABILITY_RATE
```

Holds the probability rate (0: lowest chance, 1: highest chance) that a zombie gets spawned on the TerrainLayer.

### leftPoint

```
private final Vector2 leftPoint
```

Stores the position of the bottom-left and bottom-right ends of the layer.

### rightPoint

```
private final Vector2 rightPoint
```

Stores the position of the bottom-left and bottom-right ends of the layer.

### slope

```
private float slope
```

Stores the slope of the layer if it is linear

### amplitude

```
private float amplitude
```

Stores the amplitude of the layer if it is a cosine function

### cosineXOffset

```
private float cosineXOffset
```

Stores the 'h' variable of the cosine function if the layer is a cosine function

### cosineYOffset

```
private float cosineYOffset
```

Stores the 'k' variable of the cosine function if the layer is a cosine function

### terrainType

```
private TerrainLayer.TerrainType terrainType
```

Stores the type of the terrain layer

### terrainDirection

```
private TerrainLayer.TerrainDirection terrainDirection
```

Stores whether the terrain goes from left to right or from right to left.

### worldSeed

```
private int worldSeed
```

Stores the world seed used to randomly generate the geometry of the layer.

### goManager

```
private GameObjectManager goManager
```

Stores the GameObjectManager used to fetch GameObjects to populate the TerrainLayer with objects.

### terrainRand

```
private java.util.Random terrainRand
```

Stores the random object used to define the terrain geometry of the layer.

### objectRand

```
private java.util.Random objectRand
```

Stores the random object used to define the objects stacked on the layer.

### profile

```
private Profile profile
```

Stores the profile used to create the TerrainLayer. Specifies the world seed, and the GameObjects already scavenged on each layer.

### gameObjects

```
private com.badlogic.gdx.utils.Array<GameObject> gameObjects
```

Stores an array of all the GameObjects that are on this layer. If gameObjectsStored==true, the array is already populated. Helper array to avoid GC.

### gameObjectsStored

```
private boolean gameObjectsStored
```

Stores true if the gameObjects array has already been populated with the correct objects. Prevents having to re-populate the array every frame.

### trees

```
private com.badlogic.gdx.utils.Array<Tree> trees
```

Holds arrays containing the different types of GameObjects on the layer.

### boxes

```
private com.badlogic.gdx.utils.Array<Box> boxes
```

### zombies

```
private com.badlogic.gdx.utils.Array<Zombie> zombies
```

### itemObjects

```
private com.badlogic.gdx.utils.Array<ItemObject> itemObjects
```

Stores an array of all the ItemObjects that have been dropped on this TerrainLayer. These items can be picked up.

## Constructor Detail

### TerrainLayer

```
public TerrainLayer(int row, int col, float startX, float startY,
TerrainLayer.TerrainDirection terrainDirection, Profile profile,
GameObjectManager goManager)
```

Constructor used to create a terrain layer.

**Parameters:**

`row` - The row of the layer

`col` - The column of the layer

`startX` - The starting x-position of the layer, specified as the left-most x-position of the layer if terrainDirection = RIGHT or the right-most x-position of the layer if terrainDirection == LEFT.

`startY` - The starting y-position of the layer, specified as the bottom y-position of either end of the layer.

`terrainDirection` - The direction the terrain faces. If TerrainDirection.RIGHT is specified, the (startX,startY) parameters specify the bottom-left end-point of the layer

`profile` - Profile where the layer retrieves the world seed, along with information on which GameObjects have already been scavenged.

`goManager` - The GameObjectManager which manages the World's GameObjects. Used to populate the layer with objects.

## Method Detail

### resetLayer

```
public void resetLayer()
```

Resets the layer. Re-computes the geometry of the layer and the objects it contains depending on its row and column. Must be called any time the layer is re-purposed to fit another row or column.

### resetTerrain

```
public void resetTerrain()
```

Resets and re-calculates the terrain geometry according to the world seed and the column number of the layer.

### resetObjects

```
public void resetObjects()
```

Resets the objects placed on the layer. This essentially places the correct objects on the layer depending on its column and row.

### canSpawnZombie

```
private boolean canSpawnZombie()
```

Returns true if a zombie can be spawned on this layer. The only reason it could not spawn is if this layer is the one where the player has spawned.

### freeGameObjects

```
public void freeGameObjects()
```

Frees the GameObjects stored inside the layer. Puts them back into the GameObjectManager's pools, so that they can be reused. Called when the layer should be re-purposed into another layer.

### addGameObject

```
public void addGameObject(GameObject gameObject)
```

Adds the given GameObject to the list of GameObjects contained by the TerrainLayer. This way, the GameObjectRenderer will know to render this GameObject.

**Parameters:**

`gameObject` - GameObject to add to the level.

### removeGameObject

```
public void removeGameObject(GameObject gameObject)
```

Removes the given GameObject from the list of GameObjects contained by the TerrainLayer. The GameObjectRenderer will know that it should not render the GameObject.

**Parameters:**

`gameObject` - GameObject to remove from the level.

### getGameObjects

```
public com.badlogic.gdx.utils.Array<GameObject> getGameObjects()
```

Returns an array of all GameObjects contained in this layer.

### getTrees

```
public com.badlogic.gdx.utils.Array<Tree> getTrees()
```

Returns an array consisting of all the Tree GameObjects that are on this layer.

### getBoxes

```
public com.badlogic.gdx.utils.Array<Box> getBoxes()
```

Returns an array containing all the Box GameObjects that are on this layer.

### getZombies

```
public com.badlogic.gdx.utils.Array<Zombie> getZombies()
```

Returns an array containing all the Zombies GameObjects that are on this layer.

### getItemObjects

```
public com.badlogic.gdx.utils.Array<ItemObject> getItemObjects()
```

Returns an array consisting of all the Item GameObjects that have been dropped on this layer and have yet to be picked up.

### closeToEdge

```
public boolean closeToEdge(GameObject gameObject)
```

Returns true if the given GameObject is close to the edge of this TerrainLayer.

**Parameters:**

`gameObject` - The GameObject which is tested to be close to the layer's edge.

**Returns:**

true, if the GameObject is close to the edge of the TerrainLayer

### getCenterX

```
public float getCenterX()
```

Gets the world x-position at the center of the terrain layer

### getCenterGroundHeight

```
public float getCenterGroundHeight()
```

Gets the height of the ground at the center of the layer in world units.

### getGroundHeight

```
public float getGroundHeight(float xPos)
```

Gets the ground height at any given x-position of the layer in world units.

**Parameters:**

`xPos` - The x-position of the TerrainLayer where the ground height must be found

**Returns:**

The y-position of the ground at the given x-position on the TerrainLayer.

### getObjectHeight

```
public float getObjectHeight(float xPos)
```

Gets the GameObject height at any given x-position on the layer in world units.

**Parameters:**

`xPos` - The x-position where the object must be placed

**Returns:**

The y-position where objects should be placed on the layer, given the above x-position

### getTopLayerHeight

```
public float getTopLayerHeight(float xPos)
```

Returns the y-position of the top of the layer at any given x-position in world units.

**Parameters:**

`xPos` - The x-position from which the TopLayerHeight is found (world units)

**Returns:**

Returns the y-position of the top of the layer at the given x-position

### getBottomLayerHeight

```
public float getBottomLayerHeight(float xPos)
```

Retrieves the height of the bottom portion of the layer at a specified x-position.

**Parameters:**

`xPos` - The x-position where the bottom height of the layer is found

**Returns:**

the y-position of the bottom of the layer at the given x-position

### setCell

```
public void setCell(int row, int col)
```

Sets the cell coordinates of the layer. The resetLayer() method must be called after this.

### setRow

```
public void setRow(int row)
```

Sets the row of the layer. The resetLayer() method must be called after this.

### getRow

```
public int getRow()
```

Gets the row of the layer.

### setCol

```
public void setCol(int col)
```

Sets the column of the layer. The resetLayer() method must be called after this.

### getCol

```
public int getCol()
```

Gets the column placement of the layer.

### setStartPosition

```
public void setStartPosition(float startX, float startY,
                             TerrainLayer.TerrainDirection terrainDirection)
```

Sets the start position of the terrain layer. Note that the specified position must either be the coordinates for the bottom-left or bottom-right position of the layer in world coordinates. If terrainDirection == TerrainDirection.RIGHT, the left coordinate must be specified. If TerrainDirection.LEFT is passed, the right end-position must be specified.

**Parameters:**

`startX` - the starting x-position of the terrain

`startY` - the starting y-position of the terrain

`terrainDirection` - the direction at which the terrain is drawn (left to right or right to left)

### getLeftPoint

```
public Vector2 getLeftPoint()
```

Returns the bottom-left end point of the layer in world coordinates.

### getRightPoint

```
public Vector2 getRightPoint()
```

Returns the bottom-right end point of the layer in world coordinates.

### getTerrainType

```
public TerrainLayer.TerrainType getTerrainType()
```

Gets the terrain type of the layer, dictating what type of equation models its geometry.

### toString

```
public java.lang.String toString()
```

**Overrides:**

`toString` in class `java.lang.Object`

#### II.3.1.12 *Level*

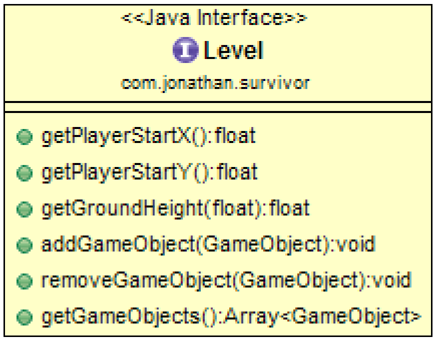

<<Java Interface>>
Level
com.jonathan.survivor
getPlayerStartX():float
getPlayerStartY():float
getGroundHeight(float):float
addGameObject(GameObject):void
removeGameObject(GameObject):void
getGameObjects():Array<GameObject>

**Figure 29 : Level class diagram**

The *Level* interface acts as a superclass to each level in the world, whether it be a *CombatLevel* or a *TerrainLevel*. First, the *getPlayerStartX():float* and the *getPlayerStartY():float* methods returns the x and y position where the player should be placed when the game is loaded. Second, the interface has a *getGroundHeight(float):float* method, which accepts any x-position. The y-position of the ground at that x-point will be returned. This y-position will dictate where a *GameObject* should be placed to be to be right on top the ground. It allows the player to stay locked to the ground at the correct height. Next, the *addGameObject(GameObject)* adds a *GameObject* to a level subclass's *gameObjects* array. This added *GameObject* will then be drawn to the screen when the level is displayed.

Finally, the *getGameObjects():Array<GameObject>* returns an array of all the *GameObjects* contained in the level. The world uses this list to update all of the level's objects, and the *GameObjectRenderer* uses it to render these same level objects.

```
public interface Level
```

## Method Detail

### getPlayerStartX

```
float getPlayerStartX()
```

Returns the x position where the player should spawn when the level is first created.

### getPlayerStartY

```
float getPlayerStartY()
```

Returns the y position where the player should spawn when the level is first created.

### getGroundHeight

```
float getGroundHeight(float xPos)
```

Returns the y-position of the ground in world coordinates at the specified x-position of the layer in world coordinates.

### outOfBounds

```
boolean outOfBounds(GameObject gameObject)
```

Returns true if the given GameObject is out of bounds of the level.

**Parameters:**

`gameObject` - the GameObject who's tested to be out of bounds of the level

**Returns:**

true, if the GameObject is out of bounds of the level

### addGameObject

```
void addGameObject(GameObject go)
```

Adds the given GameObject to the level. Like this, the level will be aware that it contains this GameObject, and this GameObject will be drawn to the screen.

**Parameters:**

`go` - the GameObject to add to the level

### removeGameObject

```
void removeGameObject(GameObject go)
```

Removes the given GameObject from the level. Like this, the level will be aware that it no longer contains this GameObject, and will thus no longer be drawn to the screen.

**Parameters:**

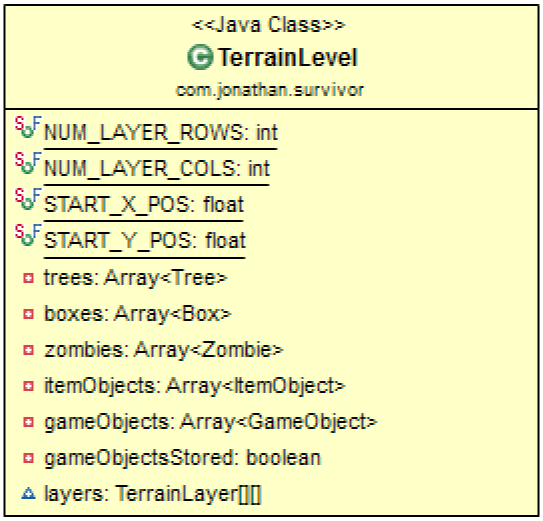
`go` - the GameObject to remove from the level

### getGameObjects

```
com.badlogic.gdx.utils.Array<GameObject> getGameObjects()
```

Returns all the GameObjects contained in the level

II.3.1.13 *TerrainLevel*



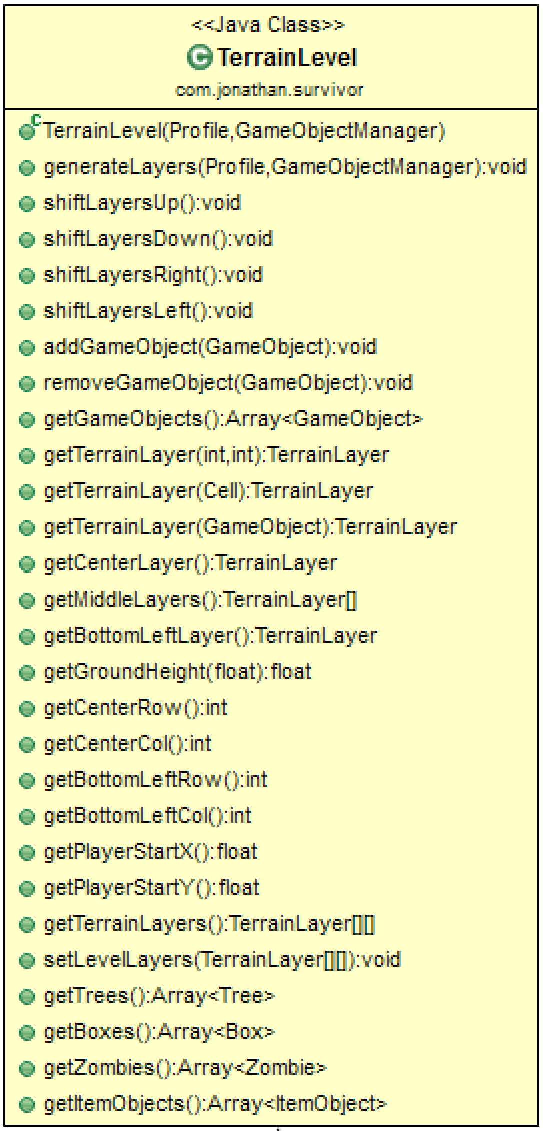
<<Java Class>>
TerrainLevel
com.jonathan.survivor

```
NUM_LAYER_ROWS: int
NUM_LAYER_COLS: int
START_X_POS: float
START_Y_POS: float
trees: Array<Tree>
boxes: Array<Box>
zombies: Array<Zombie>
itemObjects: Array<ItemObject>
gameObjects: Array<GameObject>
gameObjectsStored: boolean
layers: TerrainLayer[][]
```

**Figure 30 : TerrainLevel class diagram (data fields)**

The *TerrainLevel* class was explained in detail in *section II.1 Algorithms*. Therefore, the explanation of its class diagram will be shortened for the sake of brevity. First, the *TerrainLevel* holds an array of *GameObjects*. These are the *gameObjects* contained in the level, such as zombies, trees, or boxes. If this *TerrainLevel* is the currently-active level in the world, its *gameObjects* array will be retrieved, and each *GameObject* will be updated and rendered. The *previousPosition* Vector2 holds the position of the player before entering combat mode. If the player beats the zombie, he will spawn at this position in the level once the player returns to exploration mode. Next, the class holds a matrix of *TerrainLayers*. Each of these layers act as an elementary function which the player can walk on.

The constructor of the class accepts the *Profile* from which to generate the *TerrainLevel*. Save data will be extracted from this instance in order to generate the level where the user last ended when saving his profile. It also accepts the *GameObjectManager*, which it will give to each *TerrainLayer* instance. These layers will then fetch pooled *GameObjects* from this manager.



<<Java Class>>

TerrainLevel

com.jonathan.survivor

● TerrainLevel(Profile,GameObjectManager)
● generateLayers(Profile,GameObjectManager):void
● shiftLayersUp():void
● shiftLayersDown():void
● shiftLayersRight():void
● shiftLayersLeft():void
● addGameObject(GameObject):void
● removeGameObject(GameObject):void
● getGameObjects():Array<GameObject>
● getTerrainLayer(int,int):TerrainLayer
● getTerrainLayer(Cell):TerrainLayer
● getTerrainLayer(GameObject):TerrainLayer
● getCenterLayer():TerrainLayer
● getMiddleLayers():TerrainLayer[]
● getBottomLeftLayer():TerrainLayer
● getGroundHeight(float):float
● getCenterRow():int
● getCenterCol():int
● getBottomLeftRow():int
● getBottomLeftCol():int
● getPlayerStartX():float
● getPlayerStartY():float
● getTerrainLayers():TerrainLayer[][]
● setLevelLayers(TerrainLayer[][]):void
● getTrees():Array<Tree>
● getBoxes():Array<Box>
● getZombies():Array<Zombie>
● getItemObjects():Array<ItemObject>

**Figure 31 : TerrainLevel class diagram (constructors and methods)**

On a separate note, the *shiftLayersUp/Down/Left/Right():void* methods shift the *terrainLayers* matrix up, down, left and right respectively. The methods are called so that the player always remains in the center layer in the matrix. As such, if the player jumps, the *shiftLayersUp()* method will be called and *terrainLayers[0]* will be shifted to *terrainLayers[terrainLayers.length-1]*, and the rest of the rows in the matrix will be rotated down. Then, the bottom layers which are shifted to the top are reset so that they contain the correct *GameObjects* which correspond to the layer's new row. The three other methods work similarly, but rotate the layers in different directions. Finally, the *getCenterLayer():TerrainLayer* method returns the *TerrainLayer* in the center of the *terrainLayers* matrix. This is the *TerrainLayer* where the player resides. The only getter that needs explaining is the *getTerrainLayer(Cell):TerrainLayer* method, which returns the *TerrainLayer* at the given row and column. Here, we can note that the row and column are relative to the layers, as they do not act as indices for the two-dimensional *terrainLayers* matrix. The *getTerrainLayer(row:int, col:int):TerrainLayer* acts similarly, except that it accepts the row and the column of the layer as separate integers.

```
public class TerrainLevel
extends java.lang.Object
implements Level
```

### Field Detail

#### NUM_LAYER_ROWS

```
public static final int NUM_LAYER_ROWS
```

Stores the number of rows of terrain layers that are displayed at once in a level. Should be odd numbers so that the center layers are integers.

#### NUM_LAYER_COLS

```
public static final int NUM_LAYER_COLS
```

Stores the number of columns of terrain layers that are displayed at once in a level. Should be odd numbers so that the center layers are integers.

#### START_X_POS

```
public static final float START_X_POS
```

Stores the left x-position of the first terrain layer. Only relevant when layers first created.

- #### START_Y_POS

```
public static final float START_Y_POS
```

Stores the bottom y-position of the first terrain layer. Only relevant when layers first created.

#### profile

```
private Profile profile
```

Stores the Profile instance used to create the TerrainLevel. This profile dictates where the player should start, and where the TerrainLevel last left off.

### trees

```
private com.badlogic.gdx.utils.Array<Tree> trees
```

Holds arrays containing the trees on the layer.

### boxes

```
private com.badlogic.gdx.utils.Array<Box> boxes
```

Holds arrays containing the boxes on the layer.

### zombies

```
private com.badlogic.gdx.utils.Array<Zombie> zombies
```

Holds arrays containing the zombies on the layer.

### itemObjects

```
private com.badlogic.gdx.utils.Array<ItemObject> itemObjects
```

Stores an array of all the ItemObjects that have been dropped on this TerrainLayer. These items can be picked up.

### gameObjects

```
private com.badlogic.gdx.utils.Array<GameObject> gameObjects
```

Helper array used to store all the GameObjects in the level. Avoids activating the garbage collector.

### gameObjectsStored

```
private boolean gameObjectsStored
```

Stores true if the gameObjects array has already been populated with the GameObjects contained in the level. Prevents having to re-populate the array every frame.

### layers

```
TerrainLayer[][] layers
```

Stores the 2d array of TerrainLayers which make up the level's geometry. Note that [0][0] is the bottom-left layer and that [NUM_LAYER_ROWS-1][NUM_LAYER_COLS-1] is always the top-right-most layer.

## Constructor Detail

### TerrainLevel

```
public TerrainLevel(Profile profile, GameObjectManager goManager)
```

Creates a terrain level given a profile, which dictates how the terrainLayers should be generated.

**Parameters:**

`profile` - The player's profile, used to create the TerrainLevel so that the user restarts where he left off.

`goManager` - The GameObjectManager from which GameObjects are retrieved and stored to be placed on individual TerrainLayers.

## Method Detail

### generateLayers

```
public void generateLayers(GameObjectManager goManager)
```

Generates the TerrainLayers for the level to display. The profile member variable populates generate the layers the way they were before application quit. Accepts the GameObjectManager used by the world. This allows each TerrainLayer to populate itself with pooled GameObjects.

**Parameters:**

`goManager` - The GameObjectManager from which pooled GameObjects are retrieved and placed on the TerrainLayer.

### shiftLayersUp

```
public void shiftLayersUp()
```

Shifts the bottom TerrainLayers to the top. Called when the user moves up a layer.

### shiftLayersDown

```
public void shiftLayersDown()
```

Shifts the top TerrainLayers to the bottom. Called when the user moves up a layer.

### shiftLayersRight

```
public void shiftLayersRight()
```

Shifts the left TerrainLayers to the right. Called when the user moves to the right of the center layer.

### shiftLayersLeft

```
public void shiftLayersLeft()
```

Shifts the right-most TerrainLayers to the left. Called when the user moves to the left of the center layer.

### addGameObject

```
public void addGameObject(GameObject gameObject)
```

Adds the given GameObject to the TerrainLayer where it belongs. Allows the GameObject to be added to the list of GameObjects contained by the correct TerrainLayer.

**Specified by:**

`addGameObject` in interface `Level`

**Parameters:**

`gameObject` - the GameObject to add to the level

### removeGameObject

```
public void removeGameObject(GameObject gameObject)
```

Removes the given GameObject from the TerrainLayer where it belongs. Allows the GameObject to be removed from the list of GameObjects of the correct TerrainLayer.

**Specified by:**

`removeGameObject` in interface `Level`

**Parameters:**

`gameObject` - the game object to remove from the level

### getGameObjects

```
public com.badlogic.gdx.utils.Array<GameObject> getGameObjects()
```

Returns an array of all the GameObjects contained in the level.

**Specified by:**

`getGameObjects` in interface `Level`

### getTerrainLayer

```
public TerrainLayer getTerrainLayer(int row, int col)
```

Returns the terrain layer with the given cell coordinates. Note that the layer must exist in the current level's layer matrix.

**Parameters:**

`row` - The row of the TerrainLayer (relative to the world, not the terrainLayers[][] matrix)

`col` - The TerrainLayer's column (relative to the world, not the terrainLayers[][] matrix)

**Returns:**

The TerrainLayer at the given cell coordinates

### getTerrainLayer

```
public TerrainLayer getTerrainLayer(Cell cell)
```

Returns the terrain layer with the given cell coordinates. Note that the layer must exist in the current level's layer matrix.

### getTerrainLayer

```
public TerrainLayer getTerrainLayer(GameObject gameObject)
```

Returns the terrain layer where the GameObject resides. Note that the layer must exist in the current level's layer matrix, or an exception will occur.

### inCenterRow

```
public boolean inCenterRow(TerrainLayer terrainLayer)
```

Returns true if the given TerrainLayer is in the center row of the level. This is the row where the player resides.

**Parameters:**

`terrainLayer` - the terrain layer whose row to test

**Returns:**

true, if the given layer is at the center row of the level

### outOfBounds

```
public boolean outOfBounds(GameObject gameObject)
```

Returns true if the given GameObject is out of bounds of the level. That is, if the GameObject is outside the TerrainLayers of the level, the object is out of bounds. Note that this method only checks if the x-position of the GameObject is out of bounds of the level. This is because the x-position of the GameObject is the only one which should dictate whether the GameObject is out of bounds of the level or not.

**Specified by:**

`outOfBounds` in interface `Level`

**Parameters:**

`gameObject` - the game object which is tested to be out of bounds of the level

**Returns:**

true, if the GameObject is out of bounds of the level

### getCenterLayer

```
public TerrainLayer getCenterLayer()
```

Returns the TerrainLayer at the center of the level. This is the layer where the player resides.

### getMiddleLayers

```
public TerrainLayer[] getMiddleLayers()
```

Returns an array of all the TerrainLayers in the middle of the level, in terms of height.

### getBottomLeftLayer

```
public TerrainLayer getBottomLeftLayer()
```

Gets the bottom-left-most layer which visible in the level.

### getTopRightLayer

```
private TerrainLayer getTopRightLayer()
```

Gets the top-right-most layer which visible in the level.

### getGroundHeight

```
public float getGroundHeight(float xPos)
```

Returns the height of the ground at a given x-position. We retrieve the height of the ground for the center layer, since none are specified.

**Specified by:**

`getGroundHeight` in interface `Level`

**See Also:**

`Level.getGroundHeight(float)`

### getCenterRow

```
public int getCenterRow()
```

Returns the row of the TerrainLayer contained in the center of the level.

### getCenterCol

```
public int getCenterCol()
```

Returns the column of the TerrainLayer at the center of the level.

### getBottomLeftRow

```
public int getBottomLeftRow()
```

Returns the row of the TerrainLayer at the bottom-left of the level.

### getBottomLeftCol

```
public int getBottomLeftCol()
```

Returns the column of the TerrainLayer at the bottom-left of the level.

### getPlayerStartX

```
public float getPlayerStartX()
```

Returns the x-position where the user should spawn when he is first dropped in the level. In this case, the center TerrainLayer of the level.

**Specified by:**

`getPlayerStartX` in interface `Level`

### getPlayerStartY

```
public float getPlayerStartY()
```

Returns the y-position where the user should spawn when he is first dropped in the level. In this case, the center layer of the level.

**Specified by:**

`getPlayerStartY` in interface `Level`

### getTerrainLayers

```
public TerrainLayer[][] getTerrainLayers()
```

Returns the 2d array which stores the TerrainLayers which dictate the TerrainLevel's geometry.

### setLevelLayers

```
public void setLevelLayers(TerrainLayer[][] layers)
```

Sets the TerrainLayer array used by the TerrainLevel. Calling this method is not recommended, as it may cause unforeseen consequences.

### getTrees

```
public com.badlogic.gdx.utils.Array<Tree> getTrees()
```

Gets the list of all trees contained in the level.

### getBoxes

```
public com.badlogic.gdx.utils.Array<Box> getBoxes()
```

Gets the list of all boxes contained in the level.

### getZombies

```
public com.badlogic.gdx.utils.Array<Zombie> getZombies()
```

Gets the list of all zombies contained in the level.

### getItemObjects

```
public com.badlogic.gdx.utils.Array<ItemObject> getItemObjects()
```

Gets the list of all ItemObjects contained in the level.

(See next page for *CombatLevel* class)

II.3.1.14 *CombatLevel*

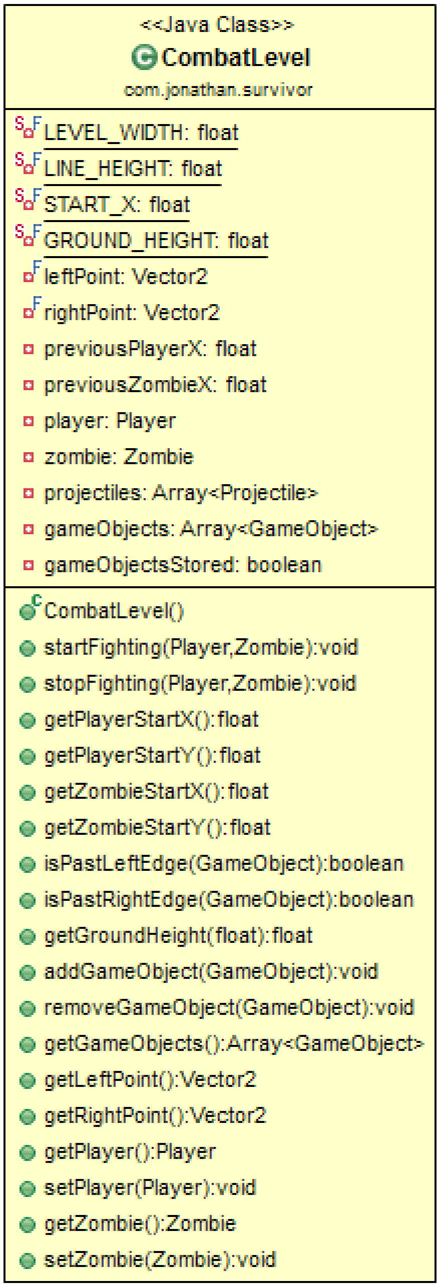


**Figure 31 : CombatLevel class diagram**

The *CombatLevel* is used as the world's active level when the player is fighting against a zombie. The zombie which the player is fighting is stored inside the *zombie* instance variable. Conversely, the *rightPoint* and the *leftPoint* Vector2s hold the two end-points of the line used to denote the level’s ground.

```
public class CombatLevel
extends java.lang.Object
implements Level
```

## Field Detail

### LEVEL_WIDTH

```
private static final float LEVEL_WIDTH
```

Stores the width of the level in meters. This is simply the length of the black line on the level.

### LINE_HEIGHT

```
private static final float LINE_HEIGHT
```

Stores the y-position of the level's black line.

### START_X

```
private static final float START_X
```

Holds the x-position at which the player and the zombie should reside relative to the center of the level.

### GROUND_HEIGHT

```
private static final float GROUND_HEIGHT
```

Stores the y-position at which the humans' feet should reside on the level

### leftPoint

```
private final Vector2 leftPoint
```

Holds the left and right end points of the black line on the level.

### rightPoint

```
private final Vector2 rightPoint
```

Holds the left and right end points of the black line on the level.

### previousPlayerX

```
private float previousPlayerX
```

Holds the x-position of the zombie and the player before entering the CombatLevel. Allows them to re-transition back to the TerrainLevel.

### previousZombieX

```
private float previousZombieX
```

### player

```
private Player player
```

Stores the player contained in the level who is fighting the zombie.

### zombie

```
private Zombie zombie
```

Holds the Zombie contained in the level which is fighting the player.

### projectiles

```
private com.badlogic.gdx.utils.Array<Projectile> projectiles
```

Stores the lists of each type of GameObject contained and rendered in the level.

### gameObjects

```
private com.badlogic.gdx.utils.Array<GameObject> gameObjects
```

Helper array used to store all the GameObjects in the level. Avoids activating the garbage collector.

### gameObjectsStored

```
private boolean gameObjectsStored
```

Stores true if the gameObjects array has already been populated with the GameObjects contained in the level. Prevents having to re-populate the array every frame.

## Constructor Detail

### CombatLevel

```
public CombatLevel()
```

Creates a default combat level

## Method Detail

### startFighting

```
public void startFighting(Player player, Zombie zombie)
```

Makes the given player and zombie start fighting on this CombatLevel.

**Parameters:**

`player` - The player engaging in combat

`zombie` - The zombie engaging in combat

### stopFighting

```
public void stopFighting(Player player, Zombie zombie)
```

Makes the given player and the given zombie stop fighting. Resets their states and their positions to cleanly switch to EXPLORATION mode.

**Parameters:**

`player` - The player meant to leave combat

`zombie` - The zombie meant to leave combat

### getPlayerStartX

```
public float getPlayerStartX()
```

Returns the x-position where the player should start on the level.

**Specified by:**

`getPlayerStartX` in interface `Level`

### getPlayerStartY

```
public float getPlayerStartY()
```

Returns the x-position where the player should start on the level.

**Specified by:**

`getPlayerStartY` in interface `Level`

### getZombieStartX

```
public float getZombieStartX()
```

Returns the x-position at which the zombie should start on the level.

### getZombieStartY

```
public float getZombieStartY()
```

Returns the x-position where the zombie should start on the level.

### outOfBounds

```
public boolean outOfBounds(GameObject gameObject)
```

Returns true if the given GameObject has passed the left or right edge of the CombatLevel.

**Specified by:**

`outOfBounds` in interface `Level`

**Parameters:**

`gameObject` - the game object who is tested to be out of bounds of the level

**Returns:**

true, if the GameObject is out of bounds of the level

### isPastLeftEdge

```
public boolean isPastLeftEdge(GameObject go)
```

Returns true if the given GameObject is past the left edge of the level.

**Parameters:**

go - the GameObject who is checked to be past the left edge of the combat level

**Returns:**

true, if the GameObject is past the left edge of the level

### isPastRightEdge

```
public boolean isPastRightEdge(GameObject go)
```

Returns true if the given GameObject is past the right edge of the level.

**Parameters:**

go - the GameObject who is checked to be past the left edge of the combat level

**Returns:**

true, if the GameObject is past the right edge of the level

### getGroundHeight

```
public float getGroundHeight(float xPos)
```

Returns the y-position of the ground at the given x-position.

**Specified by:**

`getGroundHeight` in interface `Level`

**Parameters:**

`xPos` - the x position of the level

**Returns:**

the ground height of the level at the given x-position

### addGameObject

```
public void addGameObject(GameObject go)
```

Adds a GameObject to the level. Consequently, the GameObject will start to be updated and rendered as part of the level.

**Specified by:**

`addGameObject` in interface `Level`

**Parameters:**

go - the GameObject to add to the level

### removeGameObject

```
public void removeGameObject(GameObject go)
```

Removes a GameObject from the level. Consequently, the GameObject will no longer be updated or rendered as a part of the level.

**Specified by:**

`removeGameObject` in interface `Level`

**Parameters:**

go - the GameObject to remove from the level

### getGameObjects

```
public com.badlogic.gdx.utils.Array<GameObject> getGameObjects()
```

**Description copied from interface: `Level`**

Returns all the GameObjects contained in the level

**Specified by:**

`getGameObjects` in interface `Level`

**See Also:**

`Level.getGameObjects()`

### getLeftPoint

```
public Vector2 getLeftPoint()
```

Gets the left end-point of the black line of the level.

### getRightPoint

```
public Vector2 getRightPoint()
```

Gets the right end-point of the black line of the level.

### getPlayer

```
public Player getPlayer()
```

Gets the player fighting on the level.

### setPlayer

```
public void setPlayer(Player player)
```

Sets the player that is fighting on the level.

### getZombie

```
public Zombie getZombie()
```

Gets the zombie fighting on the level.

### setZombie

```
public void setZombie(Zombie zombie)
```

Sets the zombie that is fighting on the level.

### getPreviousPlayerX

```
public float getPreviousPlayerX()
```

Gets the player's x-position before entering the combat level.

### setPreviousPlayerX

```
public void setPreviousPlayerX(float previousPlayerX)
```

Sets the player's x-position before entering the combat level.

### getPreviousZombieX

```
public float getPreviousZombieX()
```

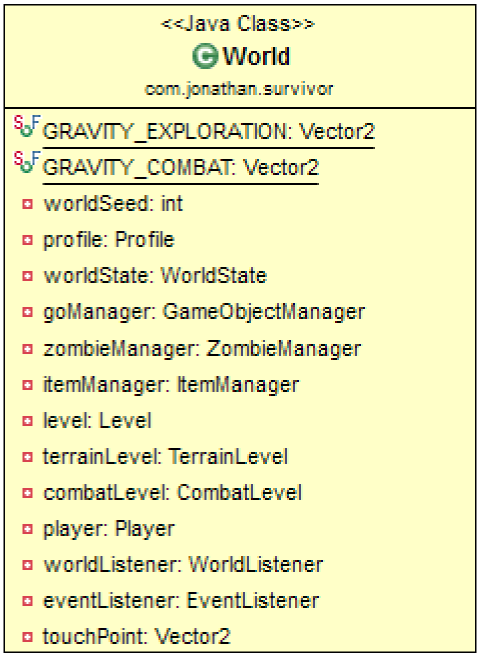
Gets the zombie's x-position before entering the combat level.

### setPreviousZombieX

```
public void setPreviousZombieX(float previousZombieX)
```

Sets the zombie's x-position before entering the combat level.

### II.3.1.15 *World*



<<Java Class>>
World
com.jonathan.survivor

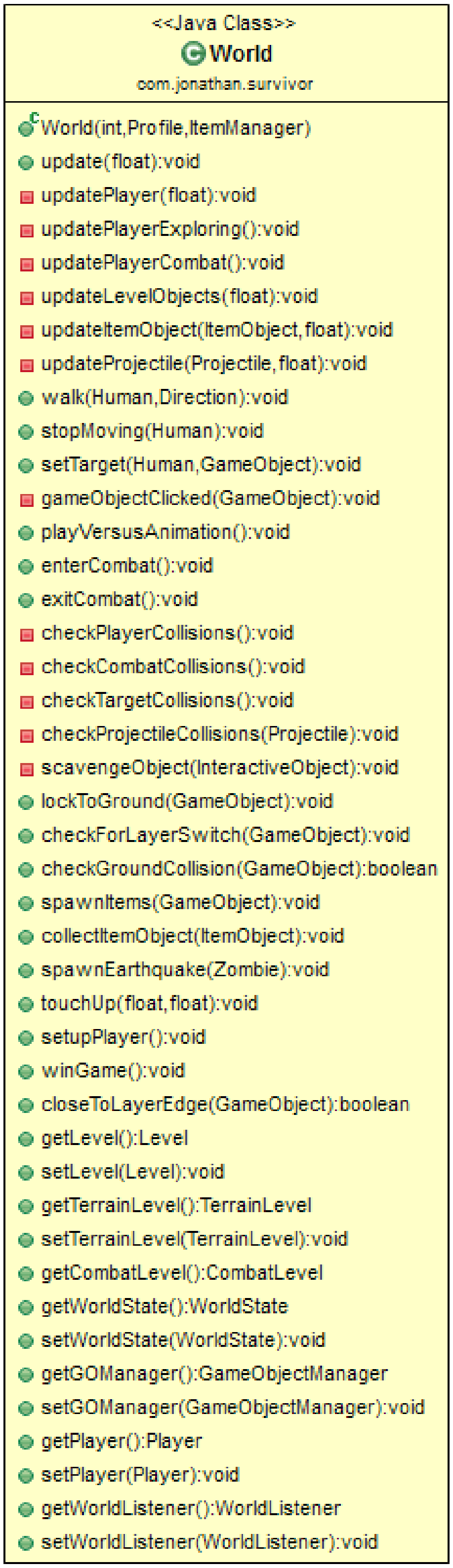
GRAVITY_EXPLORATION: Vector2
GRAVITY_COMBAT: Vector2
worldSeed: int
profile: Profile
worldState: WorldState
goManager: GameObjectManager
zombieManager: ZombieManager
itemManager: ItemManager
level: Level
terrainLevel: TerrainLevel
combatLevel: CombatLevel
player: Player
worldListener: WorldListener
eventListener: EventListener
touchPoint: Vector2

**Figure 32 : World class diagram (data fields)**

Note: the class diagram for the *World* was too large to fit into a single page, and was thus split into two different figures. *See next page for second part of UML diagram.*

The *World* class controls all game logic, such as character movement and zombie movement. It holds a reference to the level being displayed by the game, along with the player being controlled by the user.

First, the gravity vector constant is used when the player jumps or falls from one layer to another. It represents the gravity to which each *GameObject* in the world is subject to. The *Human's* acceleration vector is set to this constant when he is jumping or falling. Next, the *worldSeed:int* is the number seed used to generate the world and the objects inside of it. The *profile* variable holds the *Profile* instance from which the world extracts save data information. The next variable, *worldState*, holds an enumeration constant. It holds the current state of the world. For instance, if the world is in FIGHTING state, the *hud* shown will be the *Combat HUD*.



<<Java Class>>

World

com.jonathan.survivor

● World(int,Profile,ItemManager)
● update(float):void
■ updatePlayer(float):void
■ updatePlayerExploring():void
■ updatePlayerCombat():void
■ updateLevelObjects(float):void
■ updateItemObject(ItemObject,float):void
■ updateProjectile(Projectile,float):void
● walk(Human,Direction):void
● stopMoving(Human):void
● setTarget(Human,GameObject):void
■ gameObjectClicked(GameObject):void
● playVersusAnimation():void
● enterCombat():void
● exitCombat():void
■ checkPlayerCollisions():void
■ checkCombatCollisions():void
■ checkTargetCollisions():void
■ checkProjectileCollisions(Projectile):void
■ scavengeObject(InteractiveObject):void
● lockToGround(GameObject):void
● checkForLayerSwitch(GameObject):void
● checkGroundCollision(GameObject):boolean
● spawnItems(GameObject):void
● collectItemObject(ItemObject):void
● spawnEarthquake(Zombie):void
● touchUp(float,float):void
● setupPlayer():void
● winGame():void
● closeToLayerEdge(GameObject):boolean
● getLevel():Level
● setLevel(Level):void
● getTerrainLevel():TerrainLevel
● setTerrainLevel(TerrainLevel):void
● getCombatLevel():CombatLevel
● getWorldState():WorldState
● setWorldState(WorldState):void
● getGOManager():GameObjectManager
● setGOManager(GameObjectManager):void
● getPlayer():Player
● setPlayer(Player):void
● getWorldListener():WorldListener
● setWorldListener(WorldListener):void

**Figure 33 : World class diagram (constructors and methods)**

Further, the *goManager* holds the *GameObjectManager* which instantiates and pools *GameObjects*. For example, when the world needs to drop an item in the world, it calls the manager's methods and retrieves a new *ItemObject* instance. Next, the *level* variable holds the level that the player is currently traversing. For instance, it can hold a reference to the *terrainLevel* variable. The *TerrainLevel* is the level where the player walks around whilst exploring the forest. If interior buildings are not implemented, the *level* variable will always hold the same reference as the *terrainLevel* variable. This is due to the fact that the user is always traversing the same terrain.

The *player* variable holds the *Player* instance that the user controls using touch input. On the other hand, the *eventListener* variable holds an instance of the *EventListener* inner class. The listener receives events pertinent to the world. For instance, if the player chops down a tree, *eventListener.scavengedObject()* is called, and the world drops items next to the scavenged object.

In terms of constructors, *World(int, Profile)* accepts the world seed used to randomly generate the world along with the profile where save data is stored. It allows the world to generate its terrain.

Alternatively, the *update(float)* method is called every game update. It updates every GameObject in the game. Internally, it simply calls *updatePlayer(float),* which updates the player's game logic, and *updateLevelObjects(float),* which updates the logic of each GameObject contained inside the world's *level* variable. Furthermore, the *updateZombie(Zombie)* method updates the game logic of the zombie passed to the method's first and only parameter. It is responsible for controlling the artificial intelligence of the zombie both in combat and exploration mode. The *updateItemObject(ItemObject,float)* accepts the *ItemObject* whose game logic needs to be updated. Its second parameter accepts the game tick's *deltaTime*. In sum, the method, updates every *ItemObject* dropped in the world that is waiting to be picked up.

Conversely, the *walk()* method is self-explanatory. It tells the player or a zombie to either walk to the right or the left. The *stopMoving(Human)* method performs the opposite task. It takes in a *Human* instance as a parameter and stops this instance from walking.

Additionally, *setTarget(Human,GameObject)* method accepts a *Human* instance, along with any *GameObject*. It tells the given *Human* to target the given *GameObject,* subsequently making the *Human* walk towards his new target. In combat mode, the *fireWeapon(Player, Weapon)* method is called when the user fires his ranged weapon. The *Player* instance which fired the shot is passed as the first argument, while the *Zombie* instance which the player shoots is passed as the second argument. Internally, this method deals damage to the zombie and sets him to *HIT* state. In terms of input handling, *gameObjectClicked(GameObject)* is called whenever a *GameObject* is clicked in the world. It accepts the GameObject which was clicked as a parameter.

In terms of collision detection, the *checkPlayerCollisions()* checks if the player has hit any other *GameObject*. If the player has intersected with a zombie, for instance, the player will transition to *Combat* mode. This method uses the helper method *checkTargetCollisions()* to check for any intersections between the player and his target. To detect collisions with the ground, the *lockToGround(GameObject)* is used. It ensures that the *GameObject* passed as an argument is locked to the level's ground. This allows all GameObjects to never fall below the ground. The *checkGroundCollision(GameObject)* works hand in hand with the *lockToGround(GameObject)* method. It takes in a *GameObject* as a parameter, and checks if it has collided with the ground whilst in the air. It returns true, for instance, if the player jumps and hit the ground. In such a case, the player is locked to the ground.

*ItemObjects* are placed in the world using the *spawnItems(GameObject)* method. It spawns items next to the *GameObject* parameter. If the given *GameObject* is an *InteractiveObject,* its *itemProbabilityMap* is used to dictate which items should be spawned next to the object. To collect an item from the ground, the *collectItemObject(ItemObject)* is used. It is called when the user clicks on an *ItemObject*. The object passed as an argument will be added to the player's inventory.

Finally, the *touchUp(float,float)* is called whenever the screen is tapped. It receives the x and y coordinate of the touch. It then calls the *World's* methods in order to manage the touch.

```
public class World
extends java.lang.Object
```

## Field Detail

### GRAVITY_EXPLORATION

```
public static final Vector2 GRAVITY_EXPLORATION
```

Stores the acceleration due to gravity in the world in m/s^2 when the player is in exploration state.

### GRAVITY_COMBAT

```
public static final Vector2 GRAVITY_COMBAT
```

Stores the acceleration due to gravity in the world in m/s^2 when the player is fighting a zombie in combat state.

### worldSeed

```
private int worldSeed
```

Stores the seed used to generate the geometry of the level and randomly place its GameObjects.

### profile

```
private Profile profile
```

Stores the profile used to create the world from a save file.

### worldState

```
private World.WorldState worldState
```

Stores the state of the world, which simply dictates the GUI the GameScreen should display.

### goManager

```
private GameObjectManager goManager
```

Holds the GameObject Manager instance used to manage the GameObjects in the world.

### zombieManager

```
private ZombieManager zombieManager
```

Stores the ZombieManager which updates zombies every game tick and controls their AI.

### itemManager

```
private ItemManager itemManager
```

Holds the ItemManager instance. Used to pool and retrieve Item instances given to every ItemObject spawned in the world..

### level

```
private Level level
```

Stores the currently-active level of the world that is being displayed. Determines the walkable area of the world.

### terrainLevel

```
private TerrainLevel terrainLevel
```

Stores an instance of a terrain level, used when the player is outside in the procedurally-generated terrain.

### combatLevel

```
private CombatLevel combatLevel
```

Holds an instance of a combat level, used when the player is fighting a zombie in combat mode.

### player

```
private Player player
```

Holds the Player GameObject that the user is guiding around the world.

### worldListener

```
private WorldListener worldListener
```

Stores the WorldListener instance which delegates events from the World to the GameScreen.

### eventListener

```
private World.EventListener eventListener
```

Listens to events delegated by the player.

### soundListener

```
private SoundListener soundListener
```

Sends events to the GameScreen whenever a sound effect needs to be played.

### touchPoint

```
private Vector2 touchPoint
```

Helper Vector2 used to store the world coordinates of the last known touch.

## Constructor Detail

### World

```
public World(int worldSeed, Profile profile, ItemManager itemManager)
```

Accepts the world seed from which terrain is generated, the profile from which save data is retrieved, and the ItemManager from which Item instances are retrieved and given to ItemObjects which are spawned in the world.

**Parameters:**

`worldSeed` - the world seed used to create the world's terrain

`profile` - the profile where all save information is loaded and used to generate the world

`itemManager` - the manager which pools item objects and sprites to be displayed in HUDs

## Method Detail

### update

```
public void update(float deltaTime)
```

Called every frame to update the world and its GameObjects.

**Parameters:**

`deltaTime` - the execution time of the last render call

### updatePlayer

```
private void updatePlayer(float deltaTime)
```

Updates the player, his movement, and his game logic.

**Parameters:**

`deltaTime` - the execution time of the last render call

### updatePlayerExploring

```
private void updatePlayerExploring()
```

Updates the player in the world when he's in EXPLORATION state, and is traversing the world.

### updatePlayerCombat

```
private void updatePlayerCombat()
```

Updates the player when he is in COMBAT mode, fighting another zombie

### updateLevelObjects

```
private void updateLevelObjects(float deltaTime)
```

Updates the GameObjects contained by the currently active level of the world.

**Parameters:**

`deltaTime` - the execution time of the last render call

### updateItemObject

```
private void updateItemObject(ItemObject itemObject, float deltaTime)
```

Updates the Item Object's game logic. Note that an ItemObject is an item on the ground that can be looted.

**Parameters:**

`itemObject` - the item object to update

`deltaTime` - the execution time of the last render call

### updateProjectile

```
private void updateProjectile(Projectile projectile, float deltaTime)
```

Updates the given projectile checks for collisions and updates its position based on its velocity.

**Parameters:**

`projectile` - the projectile to update

`deltaTime` - the execution time of the last render call

### walk

```
public void walk(Human human, Human.Direction direction)
```

This method is called once to make the human move in the given direction.

**Parameters:**

`human` - the human who wants to walk

`direction` - the direction in which to walk

### stopMoving

```
public void stopMoving(Human human)
```

Stops the given Human GameObject from moving.

**Parameters:**

`human` - the human to stop moving

### setTarget

```
public void setTarget(Human human, GameObject target)
```

Makes the player walk to the specified GameObject.

**Parameters:**

`human` - the human who is targetting the given target

`target` - the target that the human will move towards

### gameObjectClicked

```
private void gameObjectClicked(GameObject gameObject)
```

Called when a GameObject in the level was touched.

**Parameters:**

`gameObject` - the GameObject which was clicked

### playVersusAnimation

```
public void playVersusAnimation()
```

Makes the versus animation play. When finished the player switches to combat mode.

### enterCombat

```
public void enterCombat()
```

Makes the player enter combat with the zombie he has collided with. Called from the VersusAnimation class when the versus animation is finished playing.

### exitCombat

```
public void exitCombat()
```

Makes the player leave COMBAT mode with the zombie he is fighting. Called after the KO animation plays.

### checkPlayerCollisions

```
private void checkPlayerCollisions()
```

Checks if the player is colliding with any GameObjects.

### checkCombatCollisions

```
private void checkCombatCollisions()
```

Checks if the player has made any collisions whilst in COMBAT mode with another zombie.

### checkTargetCollisions

```
private void checkTargetCollisions()
```

Checks if the player has collided with his target.

### checkProjectileCollisions

```
private void checkProjectileCollisions(Projectile projectile)
```

Checks if the projectile has hit anything of interest. If, for instance, an earthquake hits the player, the player takes damage.

**Parameters:**

`projectile` - the projectile whose bounding box is tested against other GameObjects to check for collision

### scavengeObject

```
private void scavengeObject(InteractiveObject scavengedObject)
```

Scavenges the given object and spawns items from it. Called when a box is opened or a tree is destroyed.

**Parameters:**

`scavengedObject` - the InteractiveObject which will spawn items

### lockToGround

```
public void lockToGround(GameObject gameObject)
```

Locks the GameObject to the ground when it is moving. Makes it so that the GameObject follows the path of the ground.

**Parameters:**

`gameObject` - the game object to lock to the ground

### checkForLayerSwitch

```
public void checkForLayerSwitch(GameObject gameObject)
```

Checks if the given GameObject has switched layers. If so, he is removed from his current layer, and added to the one to which he moved to. Note that this only checks if the GameObject has changed layers horizontally, not vertically.

**Parameters:**

`gameObject` - the game object who is checked for a horizontal layer switch

### checkGroundCollision

```
public boolean checkGroundCollision(GameObject gameObject)
```

Returns true if the GameObject is jumping or falling and has fell past the ground.

**Parameters:**

`gameObject` - the game object which is tested to have touched the ground.

**Returns:**

true, if the GameObject was jumping or falling and has hit the ground

### outOfBounds

```
public boolean outOfBounds(GameObject gameObject)
```

Returns true if the given gameObject is out of bounds of the world. A GameObject satisfies this condition if it is out of bounds of the currently-active level.

**Parameters:**

`gameObject` - the game object to check for being out of bounds

**Returns:**

true, if the GameObject is out of bounds of the world's level

### spawnItems

```
public void spawnItems(GameObject gameObject)
```

Spawns items at the GameObject's location. Called when a tree is chopped down or any other GameObject is scavenged/destroyed.

**Parameters:**

`gameObject` - the game object from which items are spawned

### collectItemObject

```
public void collectItemObject(ItemObject itemObject)
```

Makes the user pick up the given Item GameObject, removing the GameObject from the world and adding it to the inventory.

**Parameters:**

`itemObject` - the item object to collect

### spawnEarthquake

```
public void spawnEarthquake(Zombie zombie)
```

Make the zombie spawn an earthquake right under his feet, which will move towards the player and try to hit him.

**Parameters:**

`zombie` - the zombie who shoots the earthquake. The earthquake is shot at the position of one of his hands.

### touchUp

```
public void touchUp(float x, float y)
```

Called when a touch was registered on the screen. Coordinates given in world units. O(n**2)

**Parameters:**

x - the world x-position of the touch

y - the world y-position of the touch

### setupPlayer

```
public void setupPlayer()
```

Sets up the player's initial variables to ensure that the player is placed at the right location.

### winGame

```
public void winGame()
```

Called when the player's TELEPORT animation is done playing. The player has thus won the game.

### closeToLayerEdge

```
public boolean closeToLayerEdge(GameObject gameObject)
```

Returns true if the GameObject is close to the left or right edges of his TerrainLayer.

**Parameters:**

`gameObject` - the game object who is checked to be next to his current layer's edge

**Returns:**

true, if the GameObject is close to the edge of the layer where he is currently residing.

### playSound

```
public void playSound(SoundListener.Sound sound)
```

Plays the given sound. Delegates an event to the GameScreen through the soundListener to play the particular sound.

**Parameters:**

`sound` - the sound to play

### getLevel

```
public Level getLevel()
```

Returns the currently active level of the world used to dictate the walkable area the world.

### setLevel

```
public void setLevel(Level level)
```

Sets the level used to determine the walkable area of the world. Also changes the GameObjects on screen.

### getTerrainLevel

```
public TerrainLevel getTerrainLevel()
```

Gets the TerrainLevel used by the world.

### setTerrainLevel

```
public void setTerrainLevel(TerrainLevel terrainLevel)
```

Sets the TerrainLevel used by the world.

### getCombatLevel

```
public CombatLevel getCombatLevel()
```

Retrieves the combat level of the world.

### getWorldState

```
public World.WorldState getWorldState()
```

Returns the state of the world, used to tell the GameScreen how to render its GUI.

### setWorldState

```
public void setWorldState(World.WorldState worldState)
```

Sets the state of the world. This can modify the way the GameScreen displays its GUI.

### getGOManager

```
public GameObjectManager getGOManager()
```

Gets the GameObjectManager used to create and manager GameObjects in the world.

### setGOManager

```
public void setGOManager(GameObjectManager goManager)
```

Sets the GameObjectManager used to create and manager GameObjects in the world.

### getPlayer

```
public Player getPlayer()
```

Gets the player the user is controlling in the world.

### setPlayer

```
public void setPlayer(Player player)
```

Sets the Player GameObject being controlled by the user.

### getWorldListener

```
public WorldListener getWorldListener()
```

Retrieves the WorldListener which delegates World events to the GameScreen.

### setWorldListener

```
public void setWorldListener(WorldListener worldListener)
```

Sets the WorldListener which delegates World events to the GameScreen.

### getSoundListener

```
public SoundListener getSoundListener()
```

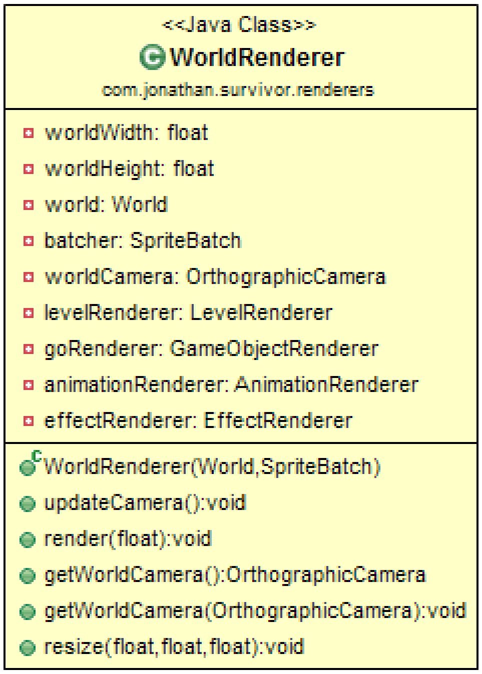
Returns the SoundListener which delegates events to the GameScreen whenever a sound effect needs to be played.

### setSoundListener

```
public void setSoundListener(SoundListener soundListener)
```

Sets the SoundListener which delegates events to the GameScreen whenever a sound effect needs to be played.

### II.3.1.16 *WorldRenderer*



<<Java Class>>
WorldRenderer
com.jonathan.survivor.renderers

worldWidth: float
worldHeight: float
world: World
batcher: SpriteBatch
worldCamera: OrthographicCamera
levelRenderer: LevelRenderer
goRenderer: GameObjectRenderer
animationRenderer: AnimationRenderer
effectRenderer: EffectRenderer

WorldRenderer(World,SpriteBatch)
updateCamera():void
render(float):void
getWorldCamera():OrthographicCamera
getWorldCamera(OrthographicCamera):void
resize(float,float,float):void

**Figure 34 : WorldRenderer class diagram**

The *WorldRenderer* is the master class which oversees all rendering operations. It holds instances to other *Renderers* which draw batches of *GameObjects*.

First, the *WorldRenderer* has two floating-point values: *worldWidth* and *worldHeight*. These values mimic those found inside the abstract *Screen* class. In fact, they store the size of the world in meters. In turn, this defines the size of the camera. Therefore, the *worldWidth* and *worldHeight* act as the size of the camera used to render the world. Next, the class has an instance of the *World* class which controls all *GameObject* logic. This *World* is then passed to the *goRenderer* and the *levelRenderer* so that they can draw the *GameObjects* and the level geometry.

Next, the *WorldRenderer* has a *SpriteBatch* instance. This variable is the universal sprite batcher used to render all visible objects to the screen. What a LibGDX *SpriteBatch* does  is it essentially gathers a batch of images for a specific texture. Then, when the batcher's drawing method is called, it flushes all of the information to *OpenGL* in order to render images from VRAM. However, the *SpriteBatch* instance must know where to draw the images to. This is where the *worldCamera* comes into play. It defines the visible portion of the world where images are drawn. The sprite batcher take the information about this visible region and draws images within a confined space, relative to the camera's position and scale.

Additionally, the class holds a *levelRenderer* variable, which draws level geometry to the screen. Conversely, the *goRenderer* variable is used to draw all the *GameObjects* to the screen.

The constructor of the *WorldRenderer* class accepts the *World* whose *GameObjects* it will render, and the *SpriteBatch* used to draw these *GameObjects* to the screen. Then, the renderer also has an *updateCamera()* method, which updates the camera's position every game tick to follow the center of the player. On the other hand, the *render(deltaTime:float)* calls the *goRenderer* and the *levelRenderer's render(float)* methods. The *resize(width:float, height:float)* method is called whenever the screen is resized, or whenever the game is first created. It resizes the camera in order to fit the screen's size.

```
public class WorldRenderer
extends java.lang.Object
```

## Field Detail

### worldWidth

```
private float worldWidth
```

Stores the width and height of the world. This is the viewable region of the world, in world units. In other words, the camera size. The size is changed inside the resize() method according the aspect ratio of the screen.

### worldHeight

```
private float worldHeight
```

### world

```
private World world
```

Stores the world whose level and gameObjects we render.

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the GameObjects.

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the OrthographicCamera used to view the world.

### levelRenderer

```
private LevelRenderer levelRenderer
```

Stores the LevelRenderer used to render every type of level.

### goRenderer

```
private GameObjectRenderer goRenderer
```

Stores the GameObjectRenderer which takes the world's data and renders it to the screen with sprites.

### animationRenderer

```
private AnimationRenderer animationRenderer
```

Holds the AnimationRenderer instance used to render all of the Spine animations which overlay the screen.

### effectRenderer

```
private EffectRenderer effectRenderer
```

Holds the EffectRenderer instance used to render all of the small effects on screen, such as the crosshairs.

## Constructor Detail

### WorldRenderer

```
public WorldRenderer(World world,
               com.badlogic.gdx.graphics.g2d.SpriteBatch batcher)
```

Creates a WorldRenderer instance used to draw the given world instance with the given SpriteBatch.

**Parameters:**

`world` - The World which is drawn

`batcher` - The SpriteBatch instance used to draw the world.

## Method Detail

### updateCamera

```
public void updateCamera()
```

Called every frame when the game is running to update the position of the camera. MUST be called before render() method.

### render

```
public void render(float deltaTime)
```

Called every frame to render the contents of the world and update the camera.

**Parameters:**

`deltaTime` - The amount of time the last render call took to execute

### getWorldCamera

```
public com.badlogic.gdx.graphics.OrthographicCamera getWorldCamera()
```

Retrieves the world camera used to render the world.

### getWorldCamera

```
public void getWorldCamera(com.badlogic.gdx.graphics.OrthographicCamer
a worldCamera)
```

Sets the world camera used to render the world.

### resize

```
public void resize(float worldWidth, float worldHeight,
                   float screenScale)
```

Called when the screen is resized, or when the world renderer is first created. Resizes the camera to adjust to the new aspect ratio. Note that the parameters specify the new width and height that the camera should have. Third parameter specifies a screen scale. This should be the factor the view frustum had to be stretched to fit the target device. The level's line thickness is multiplied by this factor to compensate for a larger or smaller screen.

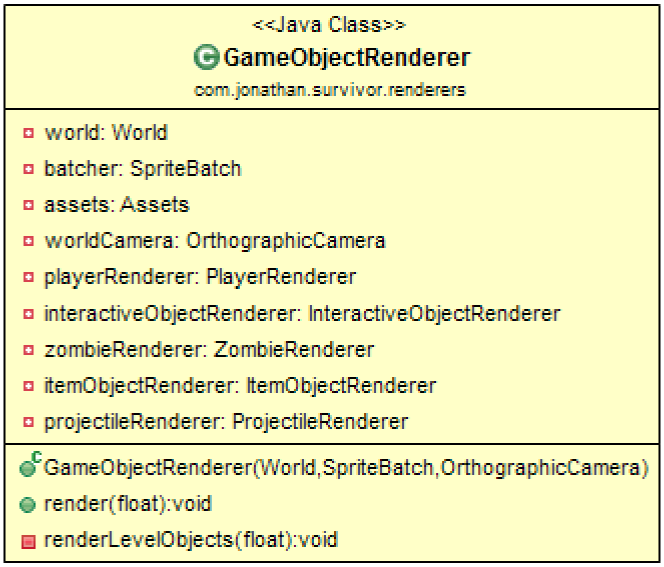
**Parameters:**

`worldWidth` - The viewable width of the world in meters

`worldHeight` - The viewable height of the world in meters

`screenScale` - The amount by which the screen is scaled relative to the target resolution of the application.

II.3.1.17 *GameObjectRenderer*



<<Java Class>>
GameObjectRenderer
com.jonathan.survivor.renderers
world: World
batcher: SpriteBatch
assets: Assets
worldCamera: OrthographicCamera
playerRenderer: PlayerRenderer
interactiveObjectRenderer: InteractiveObjectRenderer
zombieRenderer: ZombieRenderer
itemObjectRenderer: ItemObjectRenderer
projectileRenderer: ProjectileRenderer
GameObjectRenderer(World,SpriteBatch,OrthographicCamera)
render(float):void
renderLevelObjects(float):void

**Figure 35 : GameObjectRenderer class diagram**

The *GameObjectRenderer* class draws all of the *GameObjects* in the world. It renders everything aside from the elementary functions which form the terrain.

First, the renderer has a reference to the *World* from which it gets the *GameObjects* it needs to draw. Next, it has the *SpriteBatch* instance used to batch images and render them in a single draw call. Further, the *GameObjectRenderer* has an instance of the *worldCamera*. This camera defines the bounds which the *GameObjects* must be confined to. Also, the class has a reference to a *playerRenderer*, a *zombieRenderer*, an *interactiveObjectRenderer*, and an *itemObjectRenderer*. These are all helper classes used to draw specific *GameObjects*.

The constructor of this class accepts the *World* instance where the renderer gets its

*GameObject* information, the *SpriteBatch* instance used to draw images, and the *OrthographicCamera* used to display the world.

The *render(float)* method extracts information from the *World*. It gather data about its *GameObjects*, and relays it to interested renderers. Note that the *renderLevelObjects()* is a helper method used to draw the *GameObjects* contained inside a specific *TerrainLayer*. It is called from the *render(float)* method.

```
public class GameObjectRenderer
extends java.lang.Object
```

### Field Detail

#### world

```
private World world
```

Stores the world whose level and gameObjects we render.

#### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the GameObjects.

#### assets

```
private Assets assets
```

Stores the Assets singleton which stores all of the visual assets needed to draw the GameObjects.

#### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the OrthographicCamera where the GameObjects are drawn.

#### playerRenderer

```
private PlayerRenderer playerRenderer
```

Stores the PlayerRenderer instance used to render the Player GameObject.

### interactiveObjectRenderer

```
private InteractiveObjectRenderer interactiveObjectRenderer
```

Stores the InteractiveObjectRenderer instance used to render any InteractiveObjects contained in the world's current level.

### zombieRenderer

```
private ZombieRenderer zombieRenderer
```

Holds the ZombieRenderer object used to render every Zombie instance.

### itemObjectRenderer

```
private ItemObjectRenderer itemObjectRenderer
```

Holds the ItemObjectRenderer used to render any ItemObjects that have been dropped in the world by scavenging GameObjects.

### projectileRenderer

```
private ProjectileRenderer projectileRenderer
```

Stores the ProjectileRenderer instance used to draw projectiles to the screen.

## Constructor Detail

### GameObjectRenderer

```
public GameObjectRenderer(World world,
              com.badlogic.gdx.graphics.g2d.SpriteBatch batcher,

com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Accepts the world from which we find the GameObjects to draw, the SpriteBatch used to draw the GameObjects, and the world camera where the GameObjects are drawn.

**Parameters:**

`world` - The World instance whose GameObjects are extracted and drawn

`batcher` - The SpriteBatch instance used to draw the GameObjects.

`worldCamera` - The camera instance where the GameObjects are drawn

## Method Detail

### render

```
public void render(float deltaTime)
```

Renders the World's GameObjects on-screen.

**Parameters:**

`deltaTime` - The amount of time passed in the last render call

### renderLevelObjects

```
private void renderLevelObjects(float deltaTime)
```

Draws the GameObjects that are contained inside the world's level.

**Parameters:**

`deltaTime` - The amount of time passed in the last render call

II.3.1.18 *LevelRenderer*

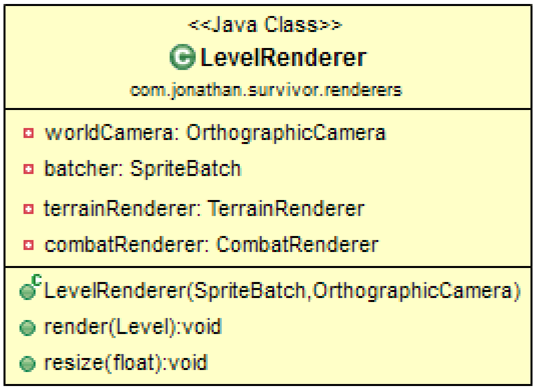

<<Java Class>>
LevelRenderer
com.jonathan.survivor.renderers
worldCamera: OrthographicCamera
batcher: SpriteBatch
terrainRenderer: TerrainRenderer
combatRenderer: CombatRenderer
LevelRenderer(SpriteBatch,OrthographicCamera)
render(Level):void
resize(float):void

**Figure 36 : LevelRenderer class diagram**

A *LevelRenderer* is used to draw the world's level geometry. Its existence is redundant for our current implementation of the project. Only if interior buildings were implemented would this class have a significant purpose. It simply delegates the *render()* call to the *TerrainRenderer.render()* method. If interior buildings were implemented, this class's purpose would be to decide whether to call the *TerrainRenderer* or something like an *InteriorRenderer*.

However, in the current implementation of the game, only the *TerrainRenderer* is needed to render level geometry.

In terms of data, the *LevelRenderer* has a reference to the *worldCamera*, which it will pass to the *terrainRenderer*. The class also holds a reference to a *TerrainRenderer*. As such, the *LevelRenderer* acts as a master class which simply calls the drawing method of the level renderer.

The constructor of the class accepts the *OrthographicCamera* which acts as the *worldCamera* instance variable that the renderer uses to draw the level's geometry. Finally, the *render(Level)* method accepts a level which it will draw to the screen.

```
public class LevelRenderer
extends java.lang.Object
```

## Field Detail

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the camera where the terrain is drawn. In this case, the world camera.

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the level's elements.

### terrainRenderer

```
private TerrainRenderer terrainRenderer
```

Stores the renderer used to draw terrain.

### combatRenderer

```
private CombatRenderer combatRenderer
```

Holds the renderer used to draw the combat level's terrain.

## Constructor Detail

### LevelRenderer

```
public LevelRenderer(
             com.badlogic.gdx.graphics.g2d.SpriteBatch batcher,
             com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Creates a LevelRenderer used to render all the levels in the game.

**Parameters:**

`batcher` - The SpriteBatch instance used to draw the levels' geometry

`worldCamera` - The camera instance where the levels' geometry is drawn

## Method Detail

### render

```
public void render(Level level)
```

Renders a level's geometry. This method is a helper method which delegates the level instance to a more specific renderer.

**Parameters:**

`level` - The level to draw

### resize

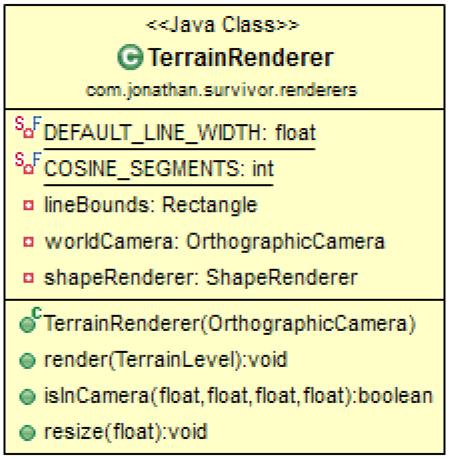
```
public void resize(float screenScale)
```

Called whenever the screen is resized. The argument contains the factor by which the screen had to be scaled compared to the target resolution. We have to rescale our lines by this factor for the lines to look the same size no matter the screen.

**Parameters:**

`screenScale` - The scale of the screen relative to the target resolution

### II.3.1.19 *TerrainRenderer*



<<Java Class>>
TerrainRenderer
com.jonathan.survivor.renderers

DEFAULT_LINE_WIDTH: float
COSINE_SEGMENTS: int
lineBounds: Rectangle
worldCamera: OrthographicCamera
shapeRenderer: ShapeRenderer

TerrainRenderer(OrthographicCamera)
render(TerrainLevel):void
isInCamera(float,float,float,float):boolean
resize(float):void

**Figure 37 : TerrainRenderer class diagram**

The *TerrainRenderer* class is used to draw terrain geometry. It takes all of the *TerrainLayers* contained within the world's *TerrainLevel* and draws their geometry using line rendering.

The first constant defined in the class is <u>*DEFAULT_LINE_WIDTH.*</u> This constant defines the default thickness of a line in world coordinates. In the game's predicted implementation, the line thickness will not vary. However, the prefix of the constant is kept in case we find varying line thicknesses more appealing as we continue development. The <u>*COSINE_SEGMENTS*</u> integer defines the amount of lines that will be used to draw a cosine function.

Next, the *lineBounds* rectangle is a helper variable. Whenever a line needs to be drawn, the two end points of this line are fit inside a rectangle. If this rectangle is inside the camera's view, the line is drawn to the screen. If not, the line is ignored. As always, the renderer also has a reference to the *worldCamera*. It defines the viewing region where lines can be drawn. Next is the *shapeRenderer* instance. The *ShapeRenderer* class is a pre-defined LibGDX class which draws lines to the screen.

The default constructor of the class accepts the *worldCamera*. Lines will be drawn only inside this camera's viewable region. The *render(float)* method uses the *ShapeRenderer* to draw the terrain using a series of lines which describe piecewise functions. However, whenever, a line is about to be drawn, its bottom-left (x1,y1) position and top-right (x2,y2) position is passed to the *isInCamera(x1, y1, x2, y2):boolean*. If the method returns *true*, the line is visible inside the camera. Therefore, the line is drawn. If the method returns *false*, the line does not intersect with the camera's viewable region. Thus, the line is not rendered, saving draw calls.

```
public class TerrainRenderer
extends java.lang.Object
```

## Field Detail

### DEFAULT_LINE_WIDTH

```
private static final float DEFAULT_LINE_WIDTH
```

Stores the default width of a line used to draw the geometry for the terrain. This is the width on the game's target resolution

### COSINE_SEGMENTS

```
private static final int COSINE_SEGMENTS
```

Stores the amount of segments used to draw a cosine function for a TerrainLayer.

### lineBounds

```
private Rectangle lineBounds
```

Helper Rectangle used to check if a TerrainLayer can be seen by the camera.

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the camera where the terrain is drawn. In this case, the world camera.

### shapeRenderer

```
private com.badlogic.gdx.graphics.glutils.ShapeRenderer shapeRenderer
```

Stores the ShapeRenderer instance used to draw the level geometry.

## Constructor Detail

### TerrainRenderer

```
public TerrainRenderer(com.badlogic.gdx.graphics.OrthographicCamera wo
rldCamera)
```

Accepts the camera where the terrain lines will be drawn.

**Parameters:**

`worldCamera` - The camera where the level's geometry is drawn

### Method Detail

### render

```
public void render(TerrainLevel level)
```

Renders the given terrainLevel's geometry using OpenGL ES lines.

**Parameters:**

`level` - The TerrainLevel to draw

### isInCamera

```
public boolean isInCamera(TerrainLayer layer)
```

Returns true if the given layer is inside the viewable region of the world's camera.

**Parameters:**

`layer` - The TerrainLayer whose bounds are checked to see if it is viewable by the worldCamera

**Returns:**

Returns true if the TerrainLayer is visible to the worldCamera.

### isInCamera

```
public boolean isInCamera(float x)
```

Returns true if this x-position is within the viewing area of the world camera.

**Parameters:**

`x` - the world x-position to test with the camera

**Returns:**

true, if an object with the given x-position is within the camera's viewable region.

**resize**

```
public void resize(float screenScale)
```

Called whenever the screen is resized. The argument contains the factor by which the screen had to be scaled to fit the device's screen We have to re-scale our lines by this factor for the lines to look the same size no matter the screen.

**Parameters:**

`screenScale` - The amount by which the screen is scaled relative to the target resolution of the application.

II.3.1.20 *InteractiveObjectRenderer & ItemObjectRenderer*

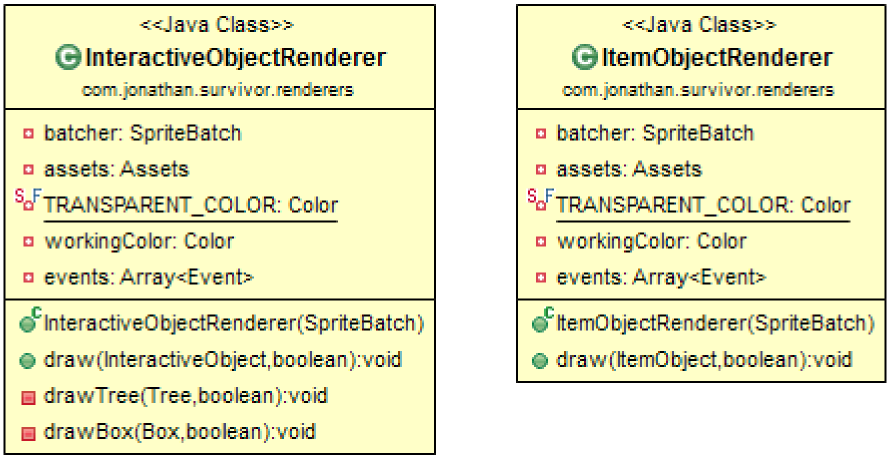

<<Java Class>>
InteractiveObjectRenderer
com.jonathan.survivor.renderers
batcher: SpriteBatch
assets: Assets
TRANSPARENT_COLOR: Color
workingColor: Color
events: Array<Event>
InteractiveObjectRenderer(SpriteBatch)
draw(InteractiveObject,boolean):void
drawTree(Tree,boolean):void
drawBox(Box,boolean):void
<<Java Class>>
ItemObjectRenderer
com.jonathan.survivor.renderers
batcher: SpriteBatch
assets: Assets
TRANSPARENT_COLOR: Color
workingColor: Color
events: Array<Event>
ItemObjectRenderer(SpriteBatch)
draw(ItemObject,boolean):void

**Figure 38 : InteractiveObjectRenderer class and the ItemObjectRenderer class diagrams**

The *InteractiveObjectRenderer* renders an *InteractiveObject* to the screen. The *ItemObjectRenderer*, conversely, renders *ItemObjects* to the screen. Both classes have a *SpriteBatch* instance which they use to draw their respective *GameObjects*.

Their constructors accept the *SpriteBatch* the renderers will use for rendering. The *InteractiveObjectRenderer* has a draw method which accepts an *InteractiveObject* to draw, along with a boolean, which denotes whether or not the object should be drawn transparently. This is true if the object is on a separate row than the player. In this case, the objects should be tinted transparent to indicate that they cannot be pressed by the player. The *ItemObjectRenderer* has a similar method, but rather accepts an *ItemObject* to draw on the screen. The second boolean accepts whether or not to draw the object transparently. Finally, the *InteractiveObjectRenderer* has helper methods *drawTree(Tree, boolean)* and *drawBox(Box, boolean)*, which draws trees and boxes, respectively, for the *draw(InteractiveObject, boolean)* method.

```
public class InteractiveObjectRenderer
extends java.lang.Object
```

### Field Detail

#### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the GameObjects.

#### assets

```
private Assets assets
```

Stores the Assets singleton which stores all of the visual assets needed to draw the interactive GameObjects.

#### TRANSPARENT_COLOR

```
private static final com.badlogic.gdx.graphics.Color TRANSPARENT_COLOR
```

Stores the color of transparent GameObjects.

#### workingColor

```
private com.badlogic.gdx.graphics.Color workingColor
```

Helper Color instance used to color GameObjects and avoid creating new color instances.

#### events

```
private com.badlogic.gdx.utils.Array<com.esotericsoftware.spine.Event>
events
```

Helper Array that's passed to the Animation.set() method.

### Constructor Detail

### InteractiveObjectRenderer

```
public InteractiveObjectRenderer(com.badlogic.gdx.graphics.g2d.SpriteB
atch batcher)
```

Accepts the SpriteBatch instance used to draw the Interactive GameObjects.

**Parameters:**

`batcher` - The SpriteBatch instance used to draw the InteractiveGameObjects.

## Method Detail

### draw

```
public void draw(InteractiveObject gameObject, boolean transparent)
```

Draws the given InteractiveObject. Accepts whether or not the GameObject should be drawn transparent.

**Parameters:**

`gameObject` - The GameObject to draw

`transparent` - Whether or not the GameObject should be drawn transparent (if it is on a separate lane than the player).

### drawTree

```
private void drawTree(Tree tree, boolean drawTransparent)
```

Renders a Tree GameObject, which contains a Spine Skeleton instance which can be drawn to the screen. Accepts whether or not it should be drawn transparent.

**Parameters:**

`tree` - The Tree instance to draw

`drawTransparent` - Whether or not the GameObject should be drawn transparent (if it is on a separate lane than the player).

### drawBox

```
private void drawBox(Box box, boolean drawTransparent)
```

Helper method called when a Box instance needs to be rendered. Second parameter accepts whether box should be drawn transparent.

**Parameters:**

`box` - The Tree instance to draw

`drawTransparent` - Whether or not the GameObject should be drawn transparent (if it is on a separate lane than the player).

```
public class ItemObjectRenderer
extends java.lang.Object
```

## Field Detail

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatch instance used to display the ItemObject.

### assets

```
private Assets assets
```

Stores the Assets singleton which stores all of the visual assets needed to draw the interactive GameObjects.

### TRANSPARENT_COLOR

```
private static final com.badlogic.gdx.graphics.Color TRANSPARENT_COLOR
```

Stores the color of transparent GameObjects.

### workingColor

```
private com.badlogic.gdx.graphics.Color workingColor
```

Helper Color instance used to color GameObjects and avoid creating new color instances.

### events

```
private com.badlogic.gdx.utils.Array<com.esotericsoftware.spine.Event>
events
```

## Constructor Detail

### ItemObjectRenderer

```
public ItemObjectRenderer(com.badlogic.gdx.graphics.g2d.SpriteBatch ba
tcher)
```

Accepts the SpriteBatch instance used to render the ItemObjects passed to render().

**Parameters:**

`batcher` - The SpriteBatch instance which will be used to render the ItemObjects.

**Method Detail**

**draw**

```
public void draw(ItemObject itemObject, boolean transparent)
```

Draws the given ItemObject to the screen

**Parameters:**

`itemObject` - The ItemObject to draw

`transparent` - Whether or not the GameObject should be drawn transparent (if it is on a separate lane than the player).

### II.3.1.21 *AnimationRenderer*

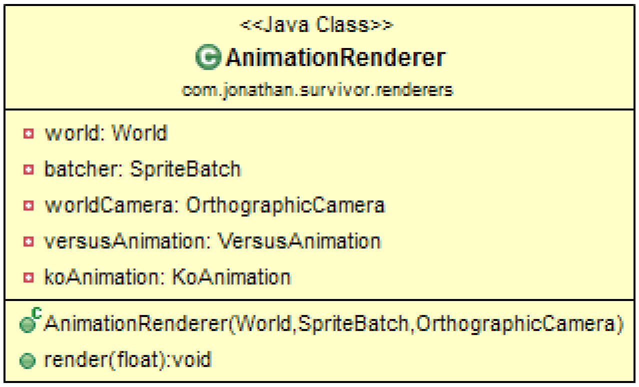

<<Java Class>>
AnimationRenderer
com.jonathan.survivor.renderers
world: World
batcher: SpriteBatch
worldCamera: OrthographicCamera
versusAnimation: VersusAnimation
koAnimation: KoAnimation
AnimationRenderer(World,SpriteBatch,OrthographicCamera)
render(float):void

**Figure 39: Visual representation of the AnmationRenderer class**

The *AnimationRenderer* is responsible for drawing all of the animations to the screen. There are two types of animations: the *KoAnimation* and the *VersusAnimation*. Animations are screen overlays which display short animations. They are responsible for informing the user that a certain event has occurred, such as the end of a battle.

```
public class AnimationRenderer
extends java.lang.Object
```

## Field Detail

### world

```
private World world
```

Stores the world, whose methods we call when certain animations are finished.

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the Spine animations.

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the OrthographicCamera where the Spine animations are drawn.

### versusAnimation

```
private VersusAnimation versusAnimation
```

Holds the VersusAnimation instance used to display the animation of the player and the zombie brawling before entering combat mode.

### koAnimation

```
private KoAnimation koAnimation
```

Stores the KoAnimation instance, which plays the Ko animation when a character dies in COMBAT mode.

## Constructor Detail

### AnimationRenderer

```
public AnimationRenderer(World world,
            com.badlogic.gdx.graphics.g2d.SpriteBatch batcher,
            com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Accepts the world from which we find the GameObjects to draw, the SpriteBatch used to draw the Spine animations, and the world camera where the Spine animations are rendered.

**Parameters:**

`world` - The World instance, whose methods are called when certain animations finish.

`batcher` - The SpriteBatch instance used to draw the animations

`worldCamera` - The world's camera, where all animations are drawn.

### Method Detail

#### render

```
public void render(float deltaTime)
```

Renders all necessary animations to the screen depending on the world's state.

**Parameters:**

`deltaTime` - The amount of time passed in the last render call

### II.3.1.22 *EffectRenderer*

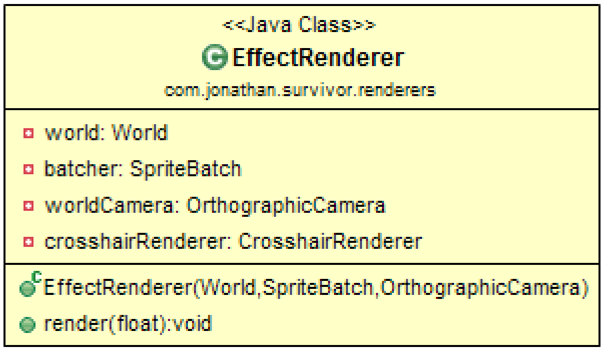

<<Java Class>>
EffectRenderer
com.jonathan.survivor.renderers
world: World
batcher: SpriteBatch
worldCamera: OrthographicCamera
crosshairRenderer: CrosshairRenderer
EffectRenderer(World,SpriteBatch,OrthographicCamera)
render(float):void

**Figure 40: Visual representation of the EffectRenderer class**

The *EffectRenderer* renders visual effects. Currently, the only visual effect that is a part of the game is the *crosshair*. It is displayed when the user charges his ranged weapon. The *EffectRenderer* is a master class used to delegate a render call to the *CrosshairRenderer* when appropriate.

```
public class EffectRenderer
extends java.lang.Object
```

## Field Detail

### world

```
private World world
```

Stores the world, whose methods we call when certain animations are finished.

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the Spine animations.

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the OrthographicCamera where the Spine animations are drawn.

### crosshairRenderer

```
private CrosshairRenderer crosshairRenderer
```

Holds the CrosshairRenderer instance used to draw all of the gun crosshairs to the screen.

## Constructor Detail

### EffectRenderer

```
public EffectRenderer(World world,
             com.badlogic.gdx.graphics.g2d.SpriteBatch batcher,
             com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Accepts the world from which we find the effects to draw, the SpriteBatch used to draw the effects, and the world camera where the effects are rendered.

**Parameters:**

`world` - The World instance, whose effects are drawn to the screen according to the world's contents

`batcher` - The SpriteBatch instance which draws the effects to the screen

`worldCamera` - The camera where the effects are rendered

## Method Detail

### render

```
public void render(float deltaTime)
```

Renders all the effects that the World needs to display.

**Parameters:**

`deltaTime` - The amount of time passed in the last render call

II.3.1.23 *ProjectileRenderer*

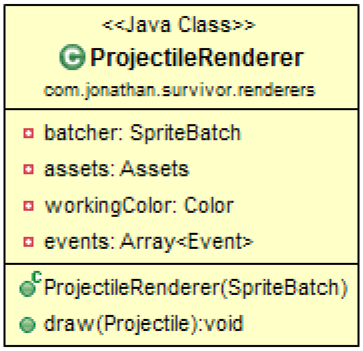


**Figure 41: Visual representation of the ProjectileRenderer class**

The *ProjectileRenderer* draws all of the projectiles to the screen. Currently, the only projectile which it renders is the *earthquake*, as this is the only projectile we have included in the game.

```
public class ProjectileRenderer
extends java.lang.Object
```

## Field Detail

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatch instance used to display the ItemObject.

### assets

```
private Assets assets
```

Stores the Assets singleton which stores all of the visual assets needed to draw the interactive GameObjects.

### workingColor

```
private com.badlogic.gdx.graphics.Color workingColor
```

Helper Color instance used to color the projectiles and to avoid creating new color instances every draw call.

### events

```
private com.badlogic.gdx.utils.Array<com.esotericsoftware.spine.Event>
events
```

## Constructor Detail

### ProjectileRenderer

```
public ProjectileRenderer(com.badlogic.gdx.graphics.g2d.SpriteBatch ba
tcher)
```

Accepts the SpriteBatch instance used to render the Projectiles passed to the draw() method.

**Parameters:**

`batcher` - The SpriteBatch instance used to draw the projectiles

## Method Detail

### draw

```
public void draw(Projectile projectile)
```

Draws the given Projectile on-screen.

**Parameters:**

`projectile` - The projectile to draw

### II.3.1.24 *PlayerRenderer & ZombieRenderer*

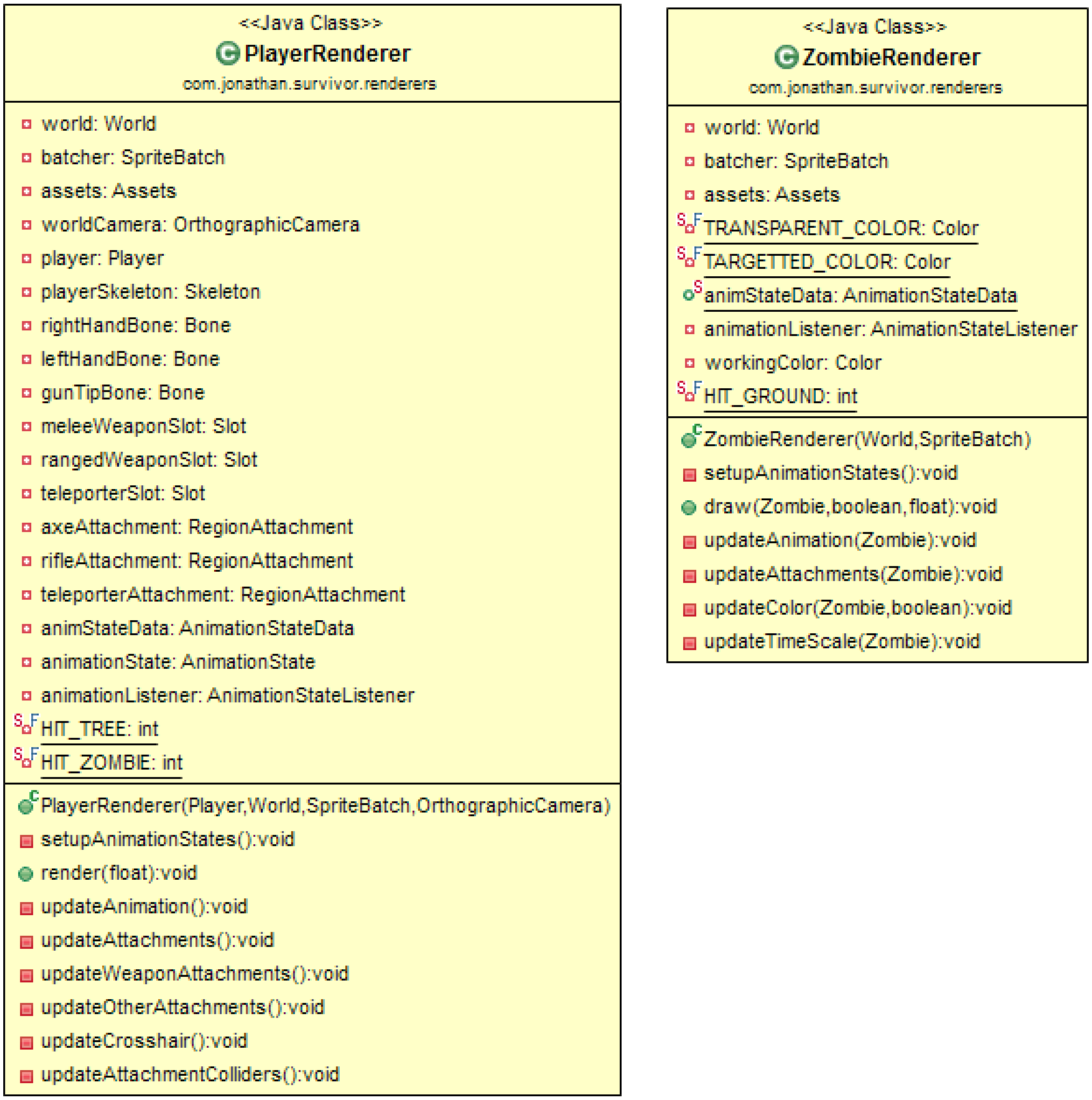

<<Java Class>>
PlayerRenderer
com.jonathan.survivor.renderers
world: World
batcher: SpriteBatch
assets: Assets
worldCamera: OrthographicCamera
player: Player
playerSkeleton: Skeleton
rightHandBone: Bone
leftHandBone: Bone
gunTipBone: Bone
meleeWeaponSlot: Slot
rangedWeaponSlot: Slot
teleporterSlot: Slot
axeAttachment: RegionAttachment
rifleAttachment: RegionAttachment
teleporterAttachment: RegionAttachment
animStateData: AnimationStateData
animationState: AnimationState
animationListener: AnimationStateListener
HIT_TREE: int
HIT_ZOMBIE: int
PlayerRenderer(Player,World,SpriteBatch,OrthographicCamera)
setupAnimationStates():void
render(float):void
updateAnimation():void
updateAttachments():void
updateWeaponAttachments():void
updateOtherAttachments():void
updateCrosshair():void
updateAttachmentColliders():void
<<Java Class>>
ZombieRenderer
com.jonathan.survivor.renderers
world: World
batcher: SpriteBatch
assets: Assets
TRANSPARENT_COLOR: Color
TARGETTED_COLOR: Color
animStateData: AnimationStateData
animationListener: AnimationStateListener
workingColor: Color
HIT_GROUND: int
ZombieRenderer(World,SpriteBatch)
setupAnimationStates():void
draw(Zombie,boolean,float):void
updateAnimation(Zombie):void
updateAttachments(Zombie):void
updateColor(Zombie,boolean):void
updateTimeScale(Zombie):void

**Figure 42 : PlayerRenderer class and ZombieRenderer class diagrams**

The *PlayerRenderer* renders the player, while the *ZombieRenderer* draws any given *Zombie* instance to the screen. First, both classes have a *SpriteBatch* instance which they use to render images. Moreover, the *PlayerRenderer* class has the *player* variable. It extracts the player's data from this variable and renders the player from this information.

Next, the *PlayerRenderer's* constructor accepts the *Player* instance it will draw. Then, when its *render(float)* method accepts a *deltaTime* parameter, and draws its *player* variable to the screen. The constructor also accepts the *SpriteBatch* instance it will use to populate the *batcher* member variable. The *render(float)* method uses the helper methods *updateAnimation()* and *updateAttachments()*. The former method updates the animation of the player, while the *updateAttachements()* method selects which attachment should be given to the player. The *Spine* animation engine uses the concept of *attachments*. Each attachment is essentially an image that is added on top of a character. Thus, the method simply chooses which attachments the player should display. For example, if the player is wearing an axe, *updateAttachments()* will ensure that the image of the axe is attached to the player.

Conversely, the *ZombieRenderer* constructor accepts only the *SpriteBatch* instance it needs to render zombies. To draw a zombie, the *render(Zombie, float, boolean)* method is called. It accepts the zombie to draw, along with the *deltaTime* parameter and a transparency boolean. If true, the zombie is drawn transparently when it is on a different row than the player.

```
public class PlayerRenderer
extends java.lang.Object
```

### Field Detail

#### world

```
private World world
```

Stores the world whose methods are called, for instance, when the player wins the game and the world needs to be informed about it.

#### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the player's sprites.

#### assets

```
private Assets assets
```

Stores the Assets singleton which stores all of the visual assets needed to draw the player.

#### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the OrthographicCamera where the player is drawn.

### player

```
private Player player
```

Stores the Player GameObject we draw to the screen by extracting its position and state.

### playerSkeleton

```
private com.esotericsoftware.spine.Skeleton playerSkeleton
```

Stores the Spine skeleton instance used to display the player and play his animations.

### rightHandBone

```
private com.esotericsoftware.spine.Bone rightHandBone
```

Holds the right hand bone on the player's skeleton in spine, which controls the player's movements.

### leftHandBone

```
private com.esotericsoftware.spine.Bone leftHandBone
```

Holds the left hand bone on the player's skeleton in spine, which controls the player's movements.

### gunTipBone

```
private com.esotericsoftware.spine.Bone gunTipBone
```

Holds the gun tip bone on the player's skeleton in spine, which is placed at the tip of the rifle.

### meleeWeaponSlot

```
private com.esotericsoftware.spine.Slot meleeWeaponSlot
```

Stores the slot which displays the melee weapon on the player's skeleton.

### rangedWeaponSlot

```
private com.esotericsoftware.spine.Slot rangedWeaponSlot
```

Stores the slot which displays the ranged weapon on the player's skeleton.

### teleporterSlot

```
private com.esotericsoftware.spine.Slot teleporterSlot
```

Stores the slot which displays the teleporter on the player's skeleton.

### axeAttachment

```
private com.esotericsoftware.spine.attachments.RegionAttachment
axeAttachment
```

Stores the RegionAttachment which store and display the images of the axe on the player.

### rifleAttachment

```
private com.esotericsoftware.spine.attachments.RegionAttachment
rifleAttachment
```

Stores the RegionAttachment which store and display the images of the rifle on the player.

### teleporterAttachment

```
private com.esotericsoftware.spine.attachments.RegionAttachment
teleporterAttachment
```

Stores the RegionAttachment which store and display the images of the teleporter on the player.

### animStateData

```
private com.esotericsoftware.spine.AnimationStateData animStateData
```

Defines the crossfading times between animations.animationState

```
private com.esotericsoftware.spine.AnimationState animationState
```

Controls the animation of the player and applies the animations the player's skeleton.

### animationListener

```
private
com.esotericsoftware.spine.AnimationState.AnimationStateListener
animationListener
```

Stores the AnimationStateListener that receives animation events.

### HIT_TREE

```
private static final int HIT_TREE
```

Event triggered when player hits a tree.

### HIT_ZOMBIE

```
private static final int HIT_ZOMBIE
```

Event triggered when player hits a zombie.

### SOUND_FOOTSTEP

```
private static final int SOUND_FOOTSTEP
```

Event triggered when player's foot hits the ground.

## Constructor Detail

### PlayerRenderer

```
public PlayerRenderer(Player player, World world,
       com.badlogic.gdx.graphics.g2d.SpriteBatch batcher,
       com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Accepts the player GameObject to render, the world whose methods are called on animation events, the SpriteBatch used to draw the player, and the world camera where the player is drawn.

**Parameters:**

`player` - The player to draw

`world` - The World instance, whose methods are called when a player's animation triggers events

`batcher` - The SpriteBatch instance used to draw the player

`worldCamera` - The camera instance where the player is drawn

## Method Detail

### setupAnimationStates

```
private void setupAnimationStates()
```

Populates the AnimationStateData and AnimationData instances used by the player.

### render

```
public void render(float deltaTime)
```

Draws the player using his Spine skeleton, which stores his animations, sprites, and everything needed to draw the player.

**Parameters:**

`deltaTime` - The amount of time taken for the previous render call

### updateAnimation

```
private void updateAnimation()
```

Updates the current animation of the player depending on his state.

### updateAttachments

```
private void updateAttachments()
```

Updates the attachments being rendered on the player.

### updateWeaponAttachments

```
private void updateWeaponAttachments()
```

Updates the weapon being displayed on the player, depending on the weapon that the player is currently using.

### updateOtherAttachments

```
private void updateOtherAttachments()
```

Updates the miscalaneous attachments on the player to change which images are displayed on him.

### updateCrosshair

```
private void updateCrosshair()
```

Updates the registered position of the tip of the player's ranged weapon. Allows the crosshair to be drawn at the correct position.

### updateAttachmentColliders

```
private void updateAttachmentColliders()
```

Updates the position and scale of the collider on the player's equipped melee weapon.

```
public class ZombieRenderer
extends java.lang.Object
```

## Field Detail

### world

```
private World world
```

Stores the world whose methods are called, for instance, when an Earthquake needs to be spawned by a zombie.

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the zombie's sprites.

### assets

```
private Assets assets
```

Stores the Assets singleton which stores all of the visual assets needed to draw the zombie.

### TRANSPARENT_COLOR

```
private static final com.badlogic.gdx.graphics.Color TRANSPARENT_COLOR
```

Stores the color of transparent zombies, when they are on different layers than the player.

### TARGETTED_COLOR

```
private static final com.badlogic.gdx.graphics.Color TARGETTED_COLOR
```

Holds the color of the zombie when he is being targetted by the player.

### animStateData

```
public static com.esotericsoftware.spine.AnimationStateData
animStateData
```

Defines the crossfading times between the zombies' animations.

### animationListener

```
private com.esotericsoftware.spine.AnimationState.AnimationStateListen
er animationListener
```

Stores the AnimationStateListener that receives animation events.

### workingColor

```
private com.badlogic.gdx.graphics.Color workingColor
```

Helper Color instance used to color the zombies and avoid creating new color instances.

### HIT_GROUND

```
private static final int HIT_GROUND
```

Stores the integer assigned to the "hit ground" event in Spine. Used to indicate which event was caught in the AnimationStateListener.

## Constructor Detail

### ZombieRenderer

```
public ZombieRenderer(World world,
              com.badlogic.gdx.graphics.g2d.SpriteBatch batcher)
```

Accepts the World instance whose methods are called when needed, and the SpriteBatch used to draw the zombies.

**Parameters:**

`world` - The World instance whose methods are called when the Zombie's animations trigger certain events

`batcher` - The SpriteBatch used to draw the Zombies.

## Method Detail

### setupAnimationStates

```
private void setupAnimationStates()
```

Populates the AnimationStateData and AnimationData instances used by the zombie.

### draw

```
public void draw(Zombie zombie, boolean transparent, float deltaTime)
```

Draws the zombie using his Spine skeleton, which stores his animations, sprites, and everything needed to draw the zombie. Accepts a boolean which depicts whether or not the zombie should be drawn transparently.

**Parameters:**

`zombie` - The Zombie to draw

`transparent` - Whether or not the Zombie should be drawn transparently (if it is on a different lane than the player)

`deltaTime` - The execution time of the previous frame

### updateAnimation

```
private void updateAnimation(Zombie zombie)
```

Updates the current animation of the zombie according to his current state.

**Parameters:**

`zombie` - The Zombie whose animation must be updated

### updateAttachments

```
private void updateAttachments(Zombie zombie)
```

Updates the attachments being rendered on the zombie.

**Parameters:**

`zombie` - The Zombie whose attachments are updated

### updateColor

```
private void updateColor(Zombie zombie, boolean transparent)
```

Updates the zombie's color depending on whether its being targetted, and whether or not it should be drawn transparent.

**Parameters:**

`zombie` - The Zombie whose color must be updated

`transparent` - Whether or not the zombie should be coloured transparent, if he is on a different lane than the player.

### updateTimeScale

```
private void updateTimeScale(Zombie zombie)
```

Updates the Zombie's TimeScale so that its animations play faster or slower, depending on the zombie's current state.

**Parameters:**

`zombie` - The Zombie whose time scale will be updated.

### II.3.1.25 *Screen*

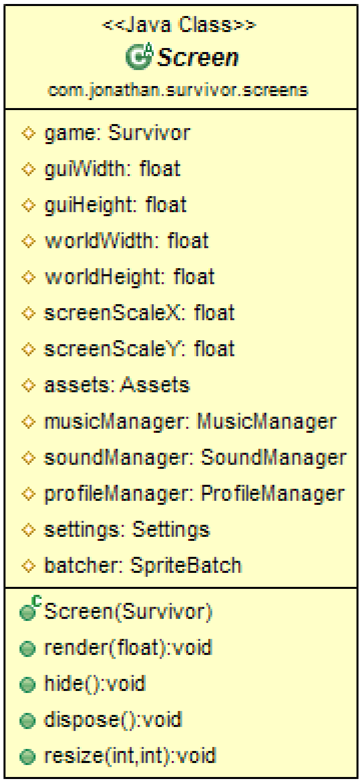


**Figure 43 : Screen class diagram**

The *Screen* class represents a window. Each principle GUI in the game is separated into a *Screen* class. A subclass holds GUI widgets, and renders them to the screen. First, the *game* variable holds the universal *Survivor* instance used by the game. The *Survivor* instance is a sort of central hub to the game. It manages all of the screens in the game and decides when to switch between them. A *Screen* instance holds a reference to this instance to ease communication between the screens and the game.

The *guiWidth* and *guiHeight* floating-points holds the width and height in pixels of the graphical user-interface. This determines the size of the camera used to display the GUI. Conversely, the *worldWidth* and *worldHeight* instance variables hold the size of the world in meters. These dimensions are in the metric system to allow for optimal physics simulations. This size, furthermore, is the size of the camera used to display the game. In fact, the world and the GUI are rendered with two different cameras to allow both to use different coordinate systems.

The *screenScaleX* and *screenScaleY* floating-point values store the amount the game has to be stretched in the x and y axes in order to fit the Android device's screen size.  This allows for the game to stretch from its base resolution to the target device's resolution.

Next, a *Screen* instance holds the *assets* variable, which is used to store the *Assets* singleton. This allows for every screen to have easy access to the visual assets needed to render the game. Furthermore, the *Screen* class has a *profileManager*. This is the manager which is used to load information which was saved to the hard drive using *Profile* instances. The *Settings* class allows any screen to save the game at any time using the *Settings.save()* method.

In terms of constructors, the *Screen* simply accepts the *Survivor* instance used to create the screen. On the other hand, the most important method in the *Screen* class is the *render(float)* method, which accepts a *deltaTime* parameter, and uses it to draw any graphicals components to the screen.  The *hide()* method is called when the screen is changed. Its functionality is to call the *dispose()* method, which subsequently frees any memory allocated to the *Screen*. Finally, the *resize(int, int):void* method accepts the width and height of the Android device. It is called when the screen is first displayed. Its purpose is to re-scale graphical assets to fit the device's resolution.

```
public abstract class Screen
extends java.lang.Object
implements com.badlogic.gdx.Screen
```

### Field Detail

#### game

```
protected Survivor game
```

Stores the game instance which created and manipulates this screen. Used for such things as changing screens using Survivor.setScreen().

### guiWidth

```
protected float guiWidth
```

Holds the width of a GUI camera. All assets drawn by a GUI camera are placed relative to these dimensions. This is the viewing space available for the camera. The viewing space is then stretched to fill the screen space. Note that their values are changed in resize() according to aspect ratio.

### guiHeight

```
protected float guiHeight
```

Holds the height of a GUI camera. All assets drawn by a GUI camera are placed relative to these dimensions. This is the viewing space available for the camera. The viewing space is then stretched to fill the screen space. Note that their values are changed in resize() according to aspect ratio.

### worldWidth

```
protected float worldWidth
```

Stores the width of a GUI camera. All assets drawn by a GUI camera are placed relative to these dimensions. This is the viewing space available for the camera. The viewing space is then stretched to fill the screen space. Note that their values are changed in resize() according to aspect ratio.

### worldHeight

```
protected float worldHeight
```

Stores the height of a GUI camera. All assets drawn by a GUI camera are placed relative to these dimensions. This is the viewing space available for the camera. The viewing space is then stretched to fill the screen space. Note that their values are changed in resize() according to aspect ratio.

### screenScaleX

```
protected float screenScaleX
```

Stores the amount we have to scale the x/y axis to fit the screen. That is, the game has a target resolution of DEFAULT_GUI_WIDTH x DEFAULT_GUI_HEIGHT. These floats store how much we have to stretch the width and height of this target resolution to fit the device. Used, in part, to scale line thickness.

### screenScaleY

```
protected float screenScaleY
```

### assets

```
protected Assets assets
```

Holds the singleton instance of the assets class. Allows for screen subclasses to have easier access to the visual/audio assets loaded from the assets instance.

### musicManager

```
protected MusicManager musicManager
```

Stores the universal Music Manager used by the game controlling this screen. Allows screen to play music and control its volume.

### soundManager

```
protected SoundManager soundManager
```

Stores the universal Sound Manager used by the game controlling this screen. Allows screen to play sound effects and control their volume.

### profileManager

```
protected ProfileManager profileManager
```

Stores the universal Profile Manager used by the game controlling this screen. Used to access profiles and their data.

### prefsManager

```
protected PreferencesManager prefsManager
```

Holds the PreferencesManager instance used to access and modify the user's preferences.

### settings

```
protected Settings settings
```

Stores the Settings instance used to save player information to the hard drive.

### batcher

```
protected com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatch instance used to draw sprites to this screen.

## Constructor Detail

### Screen

```
public Screen(Survivor game)
```

Instantiates a new screen.

**Parameters:**

`game` - the game instance which was used to create the screen. This game instance holds valuable data for each screen.

## Method Detail

### render

```
public void render(float deltaTime)
```

Called every frame to update game logic or draw graphics to the screen.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### hide

```
public void hide()
```

Called when the user switches out of this screen. In this case, we dispose of the memory allocated to the screen. This is because we never store instances of screens. Once the user switches out of them, they are lost in memory. Thus, we need to free resources allocated to it.

**Specified by:**

`hide` in interface `com.badlogic.gdx.Screen`

### dispose

```
public void dispose()
```

Called either when the Application is ended, or when the user switches out of this screen. Here, we need to free resources used by the screen.

**Specified by:**

`dispose` in interface `com.badlogic.gdx.Screen`

### resize

```
public void resize(int width, int height)
```

Called when the screen resizes, and when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

### II.3.1.26 *CompanySplashScreen, LoadingScreen, MainMenuScreen & WorldSelectScreen*

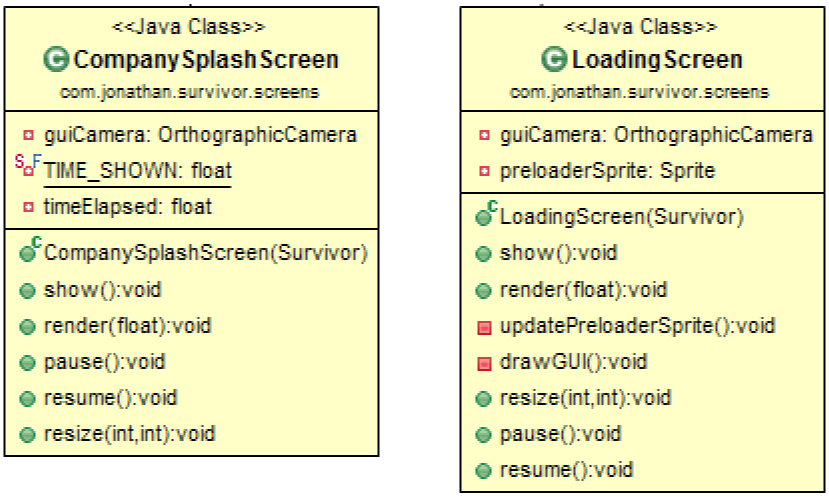


**Figure 44 : CompanySplashScreen and the LoadingScreen class diagrams**

These *Screen* subclasses are all used to display the GUIs shown before the player enters the game. Their appearance is shown inside *section II.2 GUI*. Their member variables all hold the UI widgets used to interact with the screen. Thus, the description of some member variables will be omitted as their purpose is self-explanatory.

```
public class CompanySplashScreen
extends Screen
```

#### Field Detail

##### guiCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera guiCamera
```

Stores the camera displaying the GUI (Splash Image).

##### frameCount

```
private int frameCount
```

Holds the number of frames the game has been in this screen. Used to determine when the splash screen should start loading assets.

##### TIME_SHOWN

```
private static final float TIME_SHOWN
```

Stores the amount of time the splash screen is shown before moving to the loading screen.

### FADE_TIME

```
private static final float FADE_TIME
```

The amount of time it takes for the splash screen to fade out.

### fading

```
private boolean fading
```

True if the splash screen has started fading.

### fadeStartTime

```
private float fadeStartTime
```

Holds the time in seconds at which the splash screen has started fading.

### timeElapsed

```
private float timeElapsed
```

Stores the time elapsed since the splash screen started showing.

## Constructor Detail

### CompanySplashScreen

```
public CompanySplashScreen(Survivor game)
```

Creates a new splash screen which displays the name of the application's creators. Accepts the Survivor instance which controls this screen and calls its render() method.

**Parameters:**

game - the Game instance used to manage the screen

## Method Detail

### show

```
public void show()
```

Called when the screen first becomes visible.

**See Also:**

`Screen.show()`

### render

```
public void render(float deltaTime)
```

Called every game tick to update and render the game.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame.

**See Also:**

`Screen.render(float)`

### update

```
private void update(float deltaTime)
```

Updates the widgets of the splash screen and its game logic before rendering them every frame.

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### fadeWidgets

```
private void fadeWidgets()
```

Fades the widgets displayed in the splash screen for FADE_TIME seconds.

### draw

```
private void draw(float deltaTime)
```

Draws the splash screen and all of its widgets.

### pause

`public void pause()`

Called when the user hides the application or quits the game.

**See Also:**

`Screen.pause()`

### resume

`public void resume()`

Called when the user re-opens the application after having left it.

**See Also:**

`Screen.resume()`

### resize

`public void resize(int width,int height)`

Called when the screen resizes, or when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

**See Also:**

`Screen.resize(int, int)`

```
public class LoadingScreen
extends Screen
```

## Field Detail

### PROGRESS_LABEL_X

```
private static final float PROGRESS_LABEL_X
```

Holds the center x-position of the progress label, relative to the center of the screen. That is, x=0 places the label at the center of the screen.

### PROGRESS_LABEL_Y

```
private static final float PROGRESS_LABEL_Y
```

Holds the center y-position of the progress label, relative to the center of the screen. That is, y=0 places the label at the center of the screen.

### HINT_LABEL_X

```
private static final float HINT_LABEL_X
```

Holds the center x-position of the hint label, relative to the center of the screen. That is, x=0 places the label at the center of the screen.

### HINT_LABEL_Y

```
private static final float HINT_LABEL_Y
```

Holds the center y-position of the hint label, relative to the center of the screen. That is, y=0 places the label at the center of the screen.

### PLAYER_X

```
private static final float PLAYER_X
```

Holds the left x-position of the player relative to the center of the screen. Note that this measurement is in meters, not in pixels, and is thus much smaller.

### PLAYER_Y

```
private static final float PLAYER_Y
```

Holds the center y-position of the player relative to the center of the screen. Note that this measurement is in meters, not in pixels, and is thus much smaller.

### HINT_DISPLAY_TIME

```
private static final float HINT_DISPLAY_TIME
```

Holds the amount of time a hint is displayed before switching to the next hint.

### guiCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera guiCamera
```

Stores the camera displaying the loading screen's GUI.

### progressLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label progressLabel
```

Holds the label showing a percentage of loading progress.

### hintLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label hintLabel
```

Holds the label showing hints which let the user pass time.

### playerSkeleton

```
private com.esotericsoftware.spine.Skeleton playerSkeleton
```

Stores the skeleton instance which draws the player to the screen.

### hints

```
private java.lang.String[] hints
```

Stores the list of all possible hints shown in the loading screen.

### hintTime

```
private int hintTime
```

Stores the amount of time the current hint has been showing.

### playerStateTime

```
private float playerStateTime
```

Holds the amount of time the player has been playing his current animation. Used to tell Spine which time in the animation to play.

### displayTime

```
private float displayTime
```

Stores the amount of time the loading screen has been displayed.

### events

```
private com.badlogic.gdx.utils.Array<com.esotericsoftware.spine.Event>
events
```

Helper Array that's passed to the Animation.set() method when the player plays an animation.

## Constructor Detail

### LoadingScreen

```
public LoadingScreen(Survivor game)
```

Instantiates a new loading screen.

**Parameters:**

game - the game used to create the LoadingScreen

## Method Detail

### show

```
public void show()
```

Called when the screen becomes chosen as the screen for the game.

### render

```
public void render(float deltaTime)
```

Called every game tick to update and render the game.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### update

```
private void update(float deltaTime)
```

Updates the logic of the loading screen, positioning certain widgets and changing certain elements.

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### drawGUI

```
private void drawGUI()
```

Draws the GUI for the Loading Screen

### resize

```
public void resize(int width, int height)
```

Called when the screen resizes, and when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

### pause

```
public void pause()
```

Called when the user hides the application or quits the game.

### resume

```
public void resume()
```

Called when the user re-opens the application after having left it.

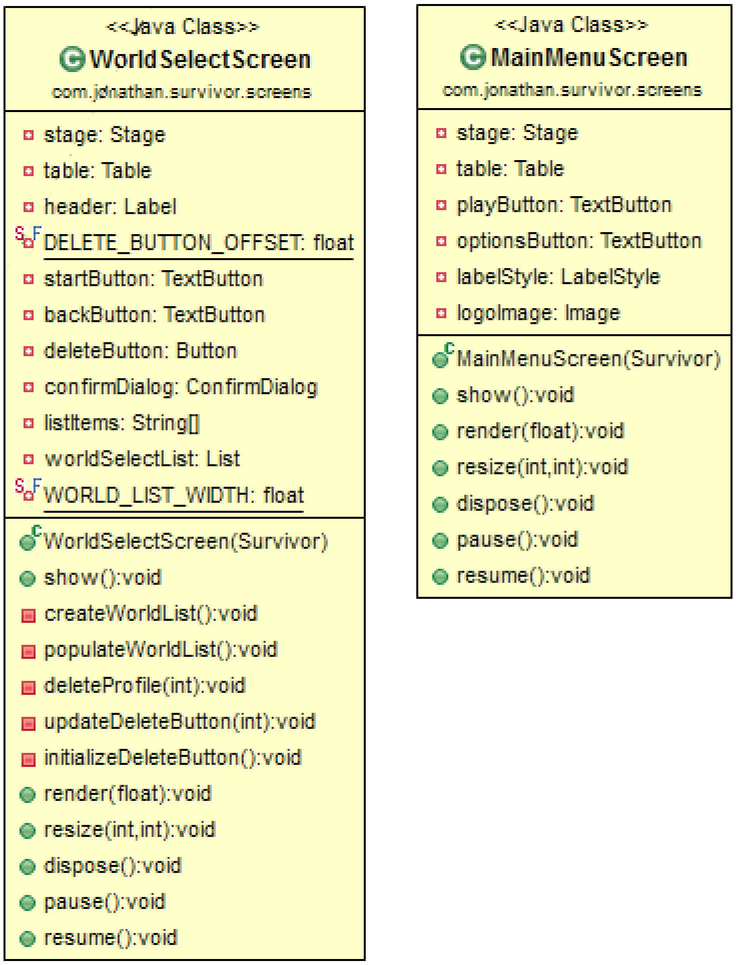


**Figure 45 : MainMenuScreen class and WorldSelectScreen class diagrams**

The *WorldSelectScreen* prompts the user to select a world from his list of created profiles. The user can select one of these profiles and either load it or delete it.

```
public class WorldSelectScreen
extends Screen
```

### Field Detail

### WORLD_LIST_WIDTH

```
private static final float WORLD_LIST_WIDTH
```

Holds the width of the world selection list. That is, the width of the blue bar in pixels for the target resolution (480x320).

### WORLD_LIST_HEIGHT

```
private static final float WORLD_LIST_HEIGHT
```

Stores the height of the world selection list. This is the width in pixels at base resolution (480x320).

### BACKGROUND_X_OFFSET

```
private static final float BACKGROUND_X_OFFSET
```

Stores the x-offset of the background relative to the center of the stage.

### BACKGROUND_Y_OFFSET

```
private static final float BACKGROUND_Y_OFFSET
```

Stores the y-offset of the background relative to the center of the stage.

### BACK_BUTTON_X_OFFSET

```
public static final float BACK_BUTTON_X_OFFSET
```

Stores the xoffset used to anchor the back button to the bottom-right of the backpack background with a certain padding.

### BACK_BUTTON_Y_OFFSET

```
public static final float BACK_BUTTON_Y_OFFSET
```

Stores the y-offset used to anchor the back button to the bottom-right of the backpack background with a certain padding.

### stage

```
private com.badlogic.gdx.scenes.scene2d.Stage stage
```

Stores the stage used as a container for the UI widgets. It is essentially the camera that draws the widgets.

#### inputListener

```
private WorldSelectScreen.InputListener inputListener
```

InputListener which receives an event when the BACK button is pressed on Android devices. Allows the user to switch back to the main menu.

#### inputMultiplexer

```
private com.badlogic.gdx.InputMultiplexer inputMultiplexer
```

Class allowing us to set multiple instance of InputListeners to receive input events.

#### table

```
private com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Stores the table actor. This actor arranges the widgets at the center of the screen in a grid-like fashion.

#### worldSelectBackground

```
private TiledImage worldSelectBackground
```

Holds the background for the WorldSelectScreen, which is formed by a tiles of two images.

#### header

```
private com.badlogic.gdx.scenes.scene2d.ui.Label header
```

Stores the label displaying the header.

#### startButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton startButton
```

Holds the "Start" button

#### deleteButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton deleteButton
```

Holds the "Delete" button

#### backButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button backButton
```

Stores the button used to go back to the main menu.

### confirmDialog

```
private ConfirmDialog confirmDialog
```

Holds the confirm dialog shown when the user presses the 'delete' button.

### profileButtons

```
private com.badlogic.gdx.utils.Array<com.badlogic.gdx.scenes.scene2d.u
i.TextButton> profileButtons
```

Stores the list of buttons which the user can press to load a saved profile.

### profileButtonTable

```
private com.badlogic.gdx.scenes.scene2d.ui.Table profileButtonTable
```

Stores a table containing all of the profile buttons, arranged in a vertical list. The user can scroll through it in a scroll pane and select an profile.

### buttonListener

```
private WorldSelectScreen.ButtonListener buttonListener
```

Holds the Listener which registers the profile button clicks. The selectedProfileId:int integer is updated when a profile button is pressed.

### buttonGroup

```
private com.badlogic.gdx.scenes.scene2d.ui.ButtonGroup buttonGroup
```

Holds the ButtonGroup instance used to ensure that only one profile button can be checked at a time.

### scrollPane

```
private com.badlogic.gdx.scenes.scene2d.ui.ScrollPane scrollPane
```

Stores the ScrollPane which allows the items in the survival guide to be scollable.

### selectedProfileId

```
private int selectedProfileId
```

Stores the id of the selected profile in the profile list.

## Constructor Detail

### WorldSelectScreen

```
public WorldSelectScreen(Survivor game)
```

Instantiates a new world select screen.

**Parameters:**

`game` - the game instance used to create and manage the screen.

## Method Detail

### show

```
public void show()
```

Called when the screen is first shown

### createWorldList

```
private void createWorldList()
```

Creates the world selection list, fetching the profiles to figure out what each item in the list should state.

### createButtonList

```
private void createButtonList()
```

Creates the profile buttons and adds them to the profileButtons array, and to the profileButtonTable.

### createProfileButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton createProfileBut
ton(int profileId)
```

Creates and returns a profile button which displays the information about a profile. Such a button is placed in the profile list.

**Parameters:**

`profileId` - the id of the profile which the button will display

**Returns:**

a new text button which displays the profile information for the given ID. Meant to be placed inside the profile list.

### deleteProfile

```
private void deleteProfile(int index)
```

Deletes the profile with the given index. The index corresponds to the item chosen in the world select list.

**Parameters:**

`index` - the id of the profile to delete

### render

```
public void render(float deltaTime)
```

Renders the screen.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### resize

```
public void resize(int width, int height)
```

Called when the screen resizes, and when the screen is created.

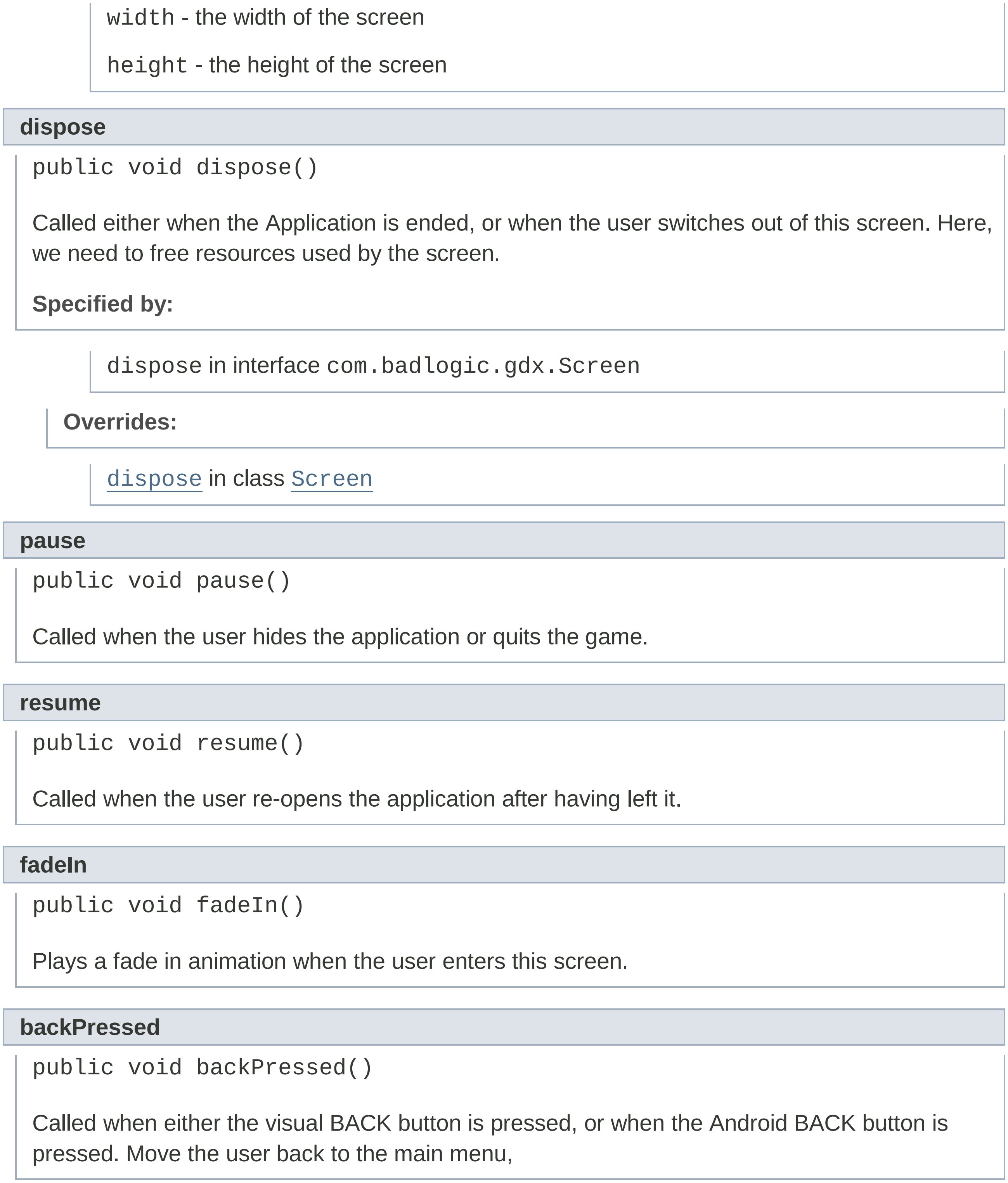
**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**



`width` - the width of the screen

`height` - the height of the screen

**dispose**

```
public void dispose()
```

Called either when the Application is ended, or when the user switches out of this screen. Here, we need to free resources used by the screen.

**Specified by:**

`dispose` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`dispose` in class `Screen`

**pause**

```
public void pause()
```

Called when the user hides the application or quits the game.

**resume**

```
public void resume()
```

Called when the user re-opens the application after having left it.

**fadeIn**

```
public void fadeIn()
```

Plays a fade in animation when the user enters this screen.

**backPressed**

```
public void backPressed()
```

Called when either the visual BACK button is pressed, or when the Android BACK button is pressed. Move the user back to the main menu,

The *MainMenuScreen* is the initial screen which is displayed when the application finishes loading. It simply prompts the user to press the *Play* button.

```
public class MainMenuScreen
extends Screen
```

## Field Detail

### stage

```
private com.badlogic.gdx.scenes.scene2d.Stage stage
```

Stores the stage used as a container for the UI widgets. It is essentially the camera that draws the widgets.

### inputListener

```
private MainMenuScreen.InputListener inputListener
```

InputListener which receives an event when the BACK button is pressed on Android devices. Allows the user to exit the game on back press.

### inputMultiplexer

```
private com.badlogic.gdx.InputMultiplexer inputMultiplexer
```

Class allowing us to set multiple instance of InputListeners to receive input events.

### table

```
private com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Stores the table actor. This simply arranges the widgets at the center of the screen in a grid fashion.

### BUTTON_Y_OFFSET

```
private static final float BUTTON_Y_OFFSET
```

Stores the number of pixels that the buttons are offset down-wards. This offset ensures that the buttons are below the game's logo.

### playButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton playButton
```

Stores the play button.

### optionsButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton optionsButton
```

Holds the options button instance.

### logoImage

```
private com.badlogic.gdx.scenes.scene2d.ui.Image logoImage
```

Stores the image for the logo, displayed at the center of the screen.

### LOGO_X_OFFSET

```
private static final float LOGO_X_OFFSET
```

Stores the x-offset of the logo relative to the center of the stage.

### LOGO_Y_OFFSET

```
private static final float LOGO_Y_OFFSET
```

Stores the y-offset of the logo relative to the center of the stage.

### BACKGROUND_X_OFFSET

```
private static final float BACKGROUND_X_OFFSET
```

Stores the x-offset of the background relative to the center of the stage.

### BACKGROUND_Y_OFFSET

```
private static final float BACKGROUND_Y_OFFSET
```

Stores the y-offset of the background relative to the center of the stage.

### mainMenuBackground

```
private TiledImage mainMenuBackground
```

Holds the background for the MainMenuScreen. This background is formed by a tiles of two images.

### quitConfirmDialog

```
private ConfirmDialog quitConfirmDialog
```

Holds the confirm dialog shown when the user presses the Android 'back' button, and wants to quit the game.

## Constructor Detail

### MainMenuScreen

```
public MainMenuScreen(Survivor game)
```

Instantiates a new main menu screen.

**Parameters:**

`game` - the game instance used to create the screen

## Method Detail

### show

```
public void show()
```

Called when the screen becomes chosen as the screen for the game.

### fadeIn

```
private void fadeIn()
```

Plays the animations of the widgets fading in to the screen.

### fadeOut

```
private void fadeOut()
```

Make the UI elements fade out once one of the buttons are pressed.

### render

```
public void render(float deltaTime)
```

Renders the screen.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame

### resize

```
public void resize(int width, int height)
```

Called when the screen resizes, or when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

### dispose

```
public void dispose()
```

Called when the application closes, or when the user leaves the screen to free up resources allocated to the screen.

**Specified by:**

`dispose` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`dispose` in class `Screen`

### pause

```
public void pause()
```

Called when the user hides the application or quits the game.

### resume

```
public void resume()
```

Called when the user re-opens the application after having left it.

### backPressed

```
public void backPressed()
```

Called when either the visual BACK button is pressed, or when the Android BACK button is pressed. Move the user back to the main menu,

### II.3.1.27 *MainMenuLoadingScreen*

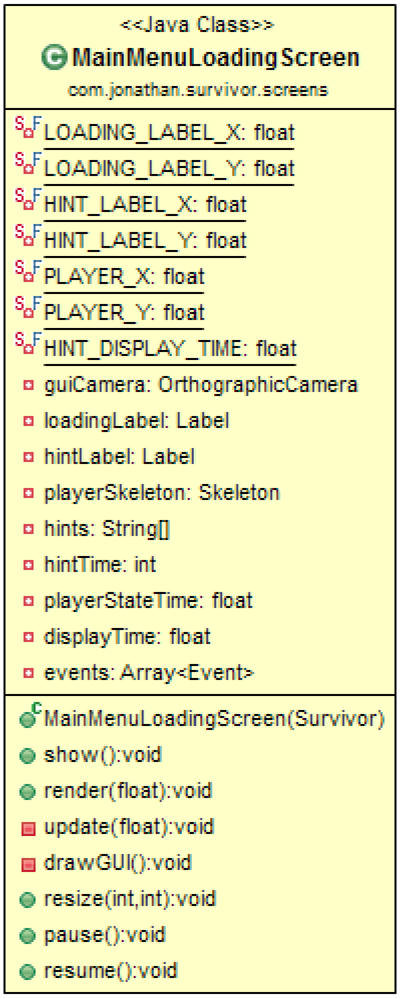


**Figure 46: MainMenuLoadingScreen class diagram**

The *MainMenuLoadingScreen* is displayed when the user goes into the pause menu, presses *Quit*, and confirms the opened dialog. This screen loads all of the assets needed to display the main menu, and transitions the player to the main menu.

Inside the loading screen, helpful hints are displayed to the user. This prevents the user from becoming bored in front of a loading screen.

```
public class MainMenuLoadingScreen
extends Screen
```

## Field Detail

### LOADING_LABEL_X

```
private static final float LOADING_LABEL_X
```

Holds the center x-position of the loading label, relative to the center of the screen. That is, x=0 places the label at the center of the screen.

### LOADING_LABEL_Y

```
private static final float LOADING_LABEL_Y
```

Holds the center y-position of the loading label, relative to the center of the screen. That is, y=0 places the label at the center of the screen.

### HINT_LABEL_X

```
private static final float HINT_LABEL_X
```

Holds the center x-position of the hint label, relative to the center of the screen. That is, x=0 places the label at the center of the screen.

### HINT_LABEL_Y

```
private static final float HINT_LABEL_Y
```

Holds the center y-position of the hint label, relative to the center of the screen. That is, y=0 places the label at the center of the screen.

### PLAYER_X

```
private static final float PLAYER_X
```

Holds the left x-position of the player relative to the center of the screen. Note that this measurement is in meters, not in pixels, and is thus much smaller.

### PLAYER_Y

```
private static final float PLAYER_Y
```

Holds the center y-position of the player relative to the center of the screen. Note that this measurement is in meters, not in pixels, and is thus much smaller.

### HINT_DISPLAY_TIME

```
private static final float HINT_DISPLAY_TIME
```

Holds the amount of time a hint is displayed before switching.

### guiCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera guiCamera
```

Stores the camera displaying the loading screen's GUI.

### loadingLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label loadingLabel
```

Holds the label which states "Loading" at the center of the screen.

### hintLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label hintLabel
```

Holds the label showing hints which let the user pass time.

### playerSkeleton

```
private com.esotericsoftware.spine.Skeleton playerSkeleton
```

Stores the skeleton instance which draws the player to the screen.

### hints

```
private java.lang.String[] hints
```

Stores the list of all possible hints shown in the loading screen.

### hintTime

```
private int hintTime
```

Stores the amount of time the current hint has been showing.

### playerStateTime

```
private float playerStateTime
```

Holds the amount of time the player has been playing his current animation. Used to tell Spine which time in the animation to play.

### displayTime

```
private float displayTime
```

Stores the amount of time the loading screen has been displayed.

### events

```
private com.badlogic.gdx.utils.Array<com.esotericsoftware.spine.Event>
events
```

Helper Array that's passed to the Animation.set() method when the player plays an animation.

## Constructor Detail

### MainMenuLoadingScreen

```
public MainMenuLoadingScreen(Survivor game)
```

Instantiates a new main menu loading screen.

**Parameters:**

`game` - the game instance used to create the screen

## Method Detail

### show

```
public void show()
```

Called when the screen becomes chosen as the screen for the game.

### render

```
public void render(float deltaTime)
```

Called every game tick to update and render the game.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### update

```
private void update(float deltaTime)
```

Updates the logic of the loading screen, positioning certain widgets and changing certain elements.

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### drawGUI

```
private void drawGUI()
```

Draws the GUI for the Loading Screen

### resize

```
public void resize(int width, int height)
```

Called when the screen resizes, or when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

### pause

```
public void pause()
```

Called when the user hides the application or quits the game.

### resume

```
public void resume()
```

Called when the user re-opens the application after having left it.

### II.3.1.28 *GameSelectScreen*

The *GameSelectScreen* allows the user to choose to either load, continue, or create a new profile. This screen was added according to user feedback.

*(See next page for class diagram and details)*

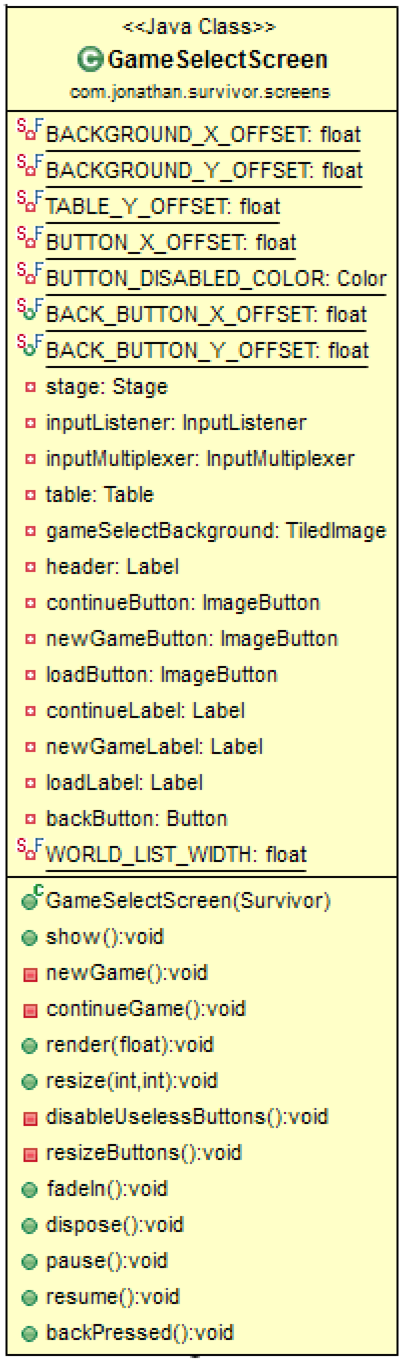


**Figure 47: Visual representation of GameSelectScreen class**

```
public class GameSelectScreen
extends Screen
```

## Field Detail

### BACKGROUND_X_OFFSET

```
private static final float BACKGROUND_X_OFFSET
```

Stores the x-offset of the background relative to the center of the stage.

### BACKGROUND_Y_OFFSET

```
private static final float BACKGROUND_Y_OFFSET
```

Stores the y-offset of the background relative to the center of the stage.

### TABLE_Y_OFFSET

```
private static final float TABLE_Y_OFFSET
```

Holds the amount that the table is offset in the y-axis.

### BUTTON_X_OFFSET

```
private static final float BUTTON_X_OFFSET
```

Stores the horizontal offset between each of the three main buttons on the screen.

### BUTTON_DISABLED_COLOR

```
private static final com.badlogic.gdx.graphics.Color
BUTTON_DISABLED_COLOR
```

Holds the color of the buttons when they are disabled.

### BACK_BUTTON_X_OFFSET

```
public static final float BACK_BUTTON_X_OFFSET
```

Stores the x-offset used to anchor the back button to the bottom-right of the screen with a certain padding.

### BACK_BUTTON_Y_OFFSET

```
public static final float BACK_BUTTON_Y_OFFSET
```

Stores the y-offset used to anchor the back button to the bottom-right of the screen with a certain padding.

### stage

```
private com.badlogic.gdx.scenes.scene2d.Stage stage
```

Stores the stage used as a container for the UI widgets. It is essentially the camera that draws the widgets.

### inputListener

```
private GameSelectScreen.InputListener inputListener
```

InputListener which receives an event when the BACK button is pressed on Android devices. Allows the user to switch back to the main menu.

### inputMultiplexer

```
private com.badlogic.gdx.InputMultiplexer inputMultiplexer
```

Class allowing us to set multiple instance of InputListeners to receive input events.

### table

```
private com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Stores the table actor. This actor arranges the widgets at the center of the screen in a grid-like fashion.

### gameSelectBackground

```
private TiledImage gameSelectBackground
```

Holds the background for the GameSelectScreen, which is formed by a tiles of two images.

### header

```
private com.badlogic.gdx.scenes.scene2d.ui.Label header
```

Stores the label displaying the header.

### continueButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton continueButton
```

Stores the button used to continue the game from the player's last profile.

### newGameButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton newGameButton
```

Holds the "New Game" button

### loadButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton loadButton
```

Stores the button used to load one of the player's saved profiles.

### continueLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label continueLabel
```

Stores the label below the "Continue" button used to indicate its name.

### newGameLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label newGameLabel
```

Stores the label below the "New Game" button used to indicate its name.

### loadLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label loadLabel
```

Stores the label below the "Load" button used to indicate its name.

### backButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button backButton
```

Holds the "Back" button

### WORLD_LIST_WIDTH

```
private static final float WORLD_LIST_WIDTH
```

Holds the width of the world selection list. That is, the width of the blue bar in pixels for the target resolution (480x320).

## Constructor Detail

### GameSelectScreen

```
public GameSelectScreen(Survivor game)
```

Instantiates a new game select screen.

**Parameters:**

`game` - the Game instance used to manage the screen

## Method Detail

### show

```
public void show()
```

Called when the screen becomes chosen as the screen for the game.

### newGame

```
private void newGame()
```

Creates a new profile, and runs the game using that profile.

### continueGame

```
private void continueGame()
```

Continues the game from the last profile that the user saved.

### render

```
public void render(float deltaTime)
```

Called every game tick to update and render the game.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame.

### resize

```
public void resize(int width, int height)
```

Called when the screen resizes, or when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

### disableUselessButtons

```
private void disableUselessButtons()
```

Disables the continueButton and the loadButton if the user has no saved profiles on his hard drive.

### resizeButtons

```
private void resizeButtons()
```

Resizes all of the buttons on the screen to ensure that their proportions are the same, no matter the size of the screen.

### fadeIn

```
public void fadeIn()
```

Plays a fade in animation when the user enters this screen.

### dispose

```
public void dispose()
```

Frees up system resources related to the screen when the application quits.

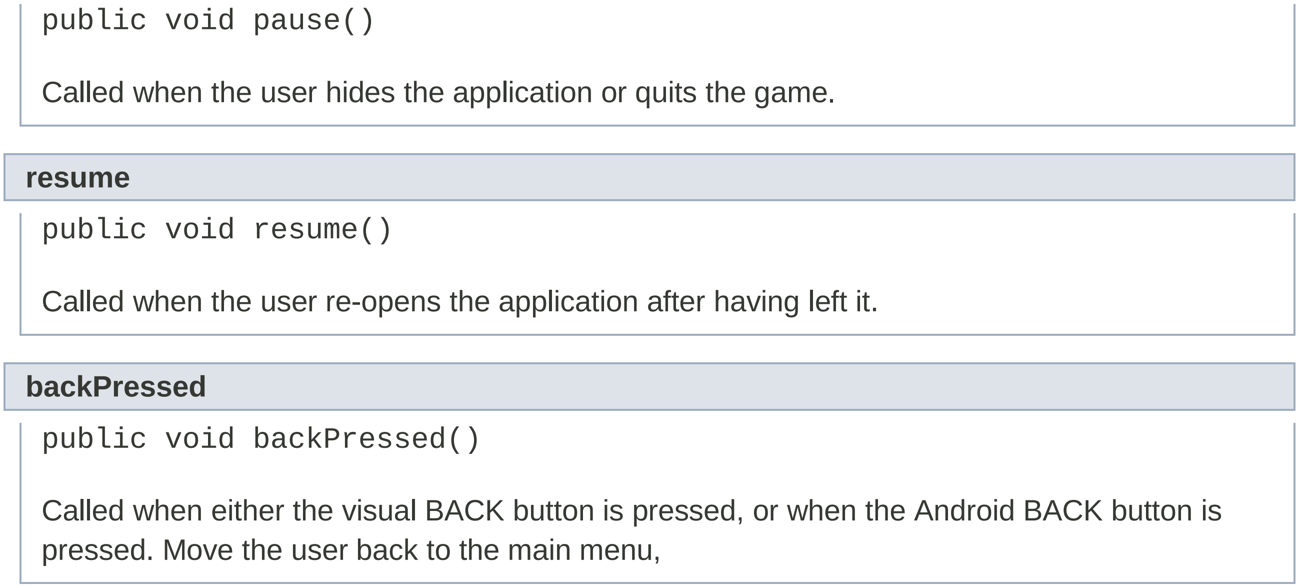
**Specified by:**

`dispose` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`dispose` in class `Screen`

### pause



```
public void pause()
```

Called when the user hides the application or quits the game.

**resume**

```
public void resume()
```

Called when the user re-opens the application after having left it.

**backPressed**

```
public void backPressed()
```

Called when either the visual BACK button is pressed, or when the Android BACK button is pressed. Move the user back to the main menu,

### II.3.1.29 *GameScreen*

The *GameScreen* is the window where the user will spend the most time in *Free the Bob*. It controls all of the elements of the world, including input and rendering. First, it has a *gameState*. This variable holds an enumeration constant which indicates the state of the game. For instance, if the user is inside the backpack menu, the state will be set to *GameState.BACKPACK*. The *paused* boolean, on the other hand, simply holds true if the game is paused. The game will stop updating, in this case. Furthermore, the *GameScreen* holds an instance to the *World*, which oversees all game logic. Next, this class has a *worldRenderer*. This instance is used to render every graphical element in the world. For instance, it draws the player and the terrain to the screen. Next, the *GameScreen* has references to an *InputManager* and a *GestureManager*, which receive all input events pertinent to the world.

In terms of GUI, the screen holds a *hud* variable. This instance variable holds the current *Hud* instance which should be used to draw the GUI which overlays the world. For instance, if the game needs to draw the *Exploration HUD*, the *hud* variable will hold an instance of *ExplorationHud*. The next five member variables each hold an instance of a *Hud*  subclass. The screen switches between these heads-up displays in order to show the graphical user-interface which corresponds to the current state of the game. Finally, the *GameScreen* holds a reference to

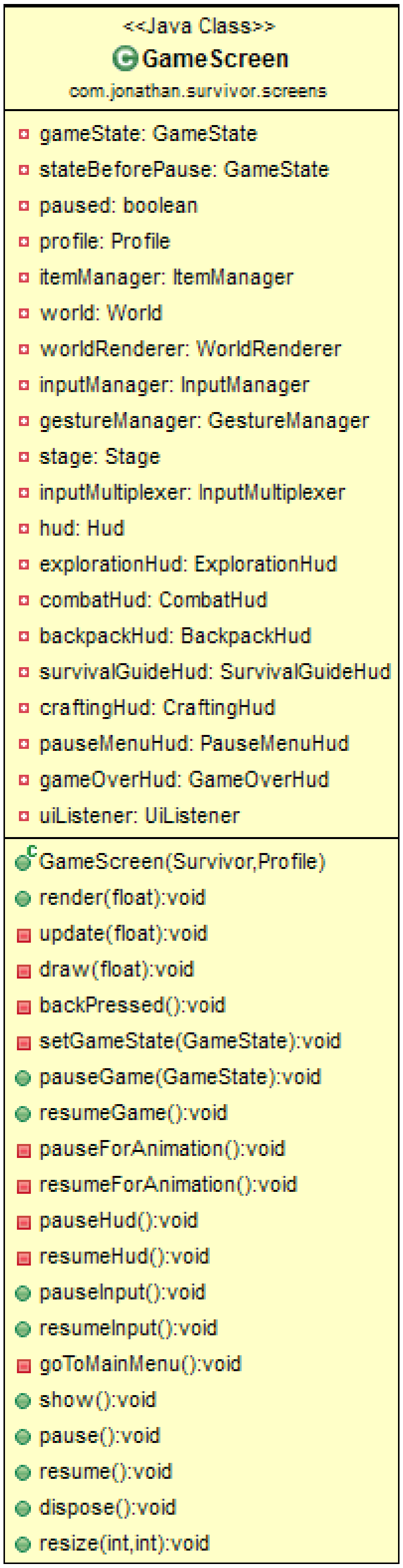


**Figure 48 : GameScreen class diagram**

a *UiListener*, an inner class which implements the *HudListener* interface. This listener is registered to every *Hud* instance, and listens for any events or button presses that have occurred inside a *Hud*. Depending on the event, the *GameScreen* can decide to delegate a method call to the *World*, or even switch from one state to another.

Next, the constructor of this screen accepts the *Survivor* instance used to create the screen. It also accepts the *Profile* instance from which the *World* will be instantiated. It allows the *GameScreen* to be aware of any save data pertinent to creating the world.

Furthermore, the *update(float)* and *render(float)* methods update world and render graphical elements respectively. They both accept the time elapsed between the previous and current frame. Note that the *setGameState(GameState)* method is private, as only the *GameScreen* should be allowed to update its state. Since it oversees all game logic, no other class should be able to modify its state. No getter for the *GameState* was included as no class needs to know about the game's state aside from the *GameScreen.*

Finally, the *pauseGame()* and *resumeGame()* methods pause and resume the game, respectively. The *pauseInput()* and *resumeInput()* methods are helper methods used to pause or resume any input handling.

```
public class GameScreen
extends Screen
```

## Field Detail

### gameState

```
private GameScreen.GameState gameState
```

Stores the state of the game, used to determine how to update the world, and how to draw the UI.

### stateBeforePause

```
private GameScreen.GameState stateBeforePause
```

Stores the game's state before it was paused. Allows the game to resume to its previous game state on resume.

### paused

```
private boolean paused
```

Holds true if the game is paused. Prevents the world and the graphics from being updated.

### profile

```
private Profile profile
```

Stores the profile used to create the world.

### itemManager

```
private ItemManager itemManager
```

Holds the ItemManager instance. Its purpose is to give access to Item instances used in the player's inventory, and sprites used inside menus.

### world

```
private World world
```

Stores the world, which controls all game logic.

### worldRenderer

```
private WorldRenderer worldRenderer
```

Stores the world renderer, which takes the objects in the world, and displays them.

### inputManager

```
private InputManager inputManager
```

Manages all simple input of the game such as "touch ups" and calls method of the world's GameObjects to match user input.

### gestureManager

```
private GestureManager gestureManager
```

Manages all gestures input of the game such as "swipes" and calls method of the world's GameObjects to match user input.

### stage

```
private com.badlogic.gdx.scenes.scene2d.Stage stage
```

Stores the stage instance where all hud elements will be placed and drawn.

### inputMultiplexer

```
private com.badlogic.gdx.InputMultiplexer inputMultiplexer
```

Class allowing us to set multiple instance of InputListeners to receive input events.

### hud

```
private Hud hud
```

Stores the currently active Hud which draws the UI to the screen.

### explorationHud

```
private ExplorationHud explorationHud
```

Stores the ExplorationHud instance which draws the UI when the user is in exploration mode.

### combatHud

```
private CombatHud combatHud
```

Holds the CombatHud instance which draws the UI when the user is in combat mode.

### backpackHud

```
private BackpackHud backpackHud
```

Stores the BackpackHud which displays the Backpack inventory screen.

### survivalGuideHud

```
private SurvivalGuideHud survivalGuideHud
```

Stores the SurvivalGuideHud which displays the survival guide menu.

### craftingHud

```
private CraftingHud craftingHud
```

Holds the CraftingHud instance which displays the crafting menu.

### pauseMenuHud

```
private PauseMenuHud pauseMenuHud
```

Holds the HUD which displays the pause menu.

### gameOverHud

```
private GameOverHud gameOverHud
```

Stores the HUD which displays the "Game Over" text when the player is dead.

### uiListener

```
private GameScreen.UiListener uiListener
```

Stores the UiListener which receives all events related to the UI or the HUD. Used to react appropriately to button presses.

### inputListener

```
private GameScreen.InputGestureListener inputListener
```

The listener which receives events fired from the InputManager class. For instance, the GameScreen is informed through this listener when the BACK key is pressed.

### sfxListener

```
private GameScreen.SfxListener sfxListener
```

Holds the listener which receives events whenever a particular sound needs to be played.

## Constructor Detail

### GameScreen

```
public GameScreen(Survivor game, Profile profile)
```

Creates a game screen. The profile used to create the screen must be specified to load the user's previous save information and update it.

**Parameters:**

`game` - the Game instance used to manage the screen

`profile` - the profile used to create the game and the world

## Method Detail

### render

`public void render(float deltaTime)`

Called every frame to update game logic and render all game entities.

**Specified by:**

`render` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`render` in class `Screen`

**Parameters:**

`deltaTime` - the execution time of the previous frame

### update

`private void update(float deltaTime)`

Updates the world and the world camera.

**Parameters:**

`deltaTime` - the execution time of the previous frame

### draw

`private void draw(float deltaTime)`

Draws the UI, along with the world and its contained GameObjects.

**Parameters:**

`deltaTime` - the execution time of the previous frame

### backPressed

`private void backPressed()`

Delegates when either the hardware back button is pressed, or the back button is pressed from the HUD.

### setGameState

`private void setGameState(GameScreen.GameState state)`

Sets the GameState. Updates the hudRenderer to draw the correct HUD.

### pauseGame

```
public void pauseGame(GameScreen.GameState newState)
```

Pauses the game whilst running. Called when transitioning to a menu. Accepts the game state the user is switching to.

**Parameters:**

`newState` - the new state to switch to when pausing

### resumeGame

```
public void resumeGame()
```

Resumes the game to its previous state before being paused.

### pauseForAnimation

```
private void pauseForAnimation()
```

Pauses the game when an animation plays. Allows the animation to finish without the player pressing anything.

### resumeForAnimation

```
private void resumeForAnimation()
```

Called when an screen overlay animation finishes playing. Resumes the game so that the user can continue playing.

### pauseHud

```
private void pauseHud()
```

Pauses the Hud so that the user can't press any button on the Hud.

### resumeHud

```
private void resumeHud()
```

Resumes the Hud so that the user can again press a button on the Hud.

### pauseInput

```
public void pauseInput()
```

Pauses the game by pausing all of the input handling.

### resumeInput

```
public void resumeInput()
```

Resumes the game by allowing the input managers to delegate method calls to the world.

### goToMainMenu

```
private void goToMainMenu()
```

Transitions to the MainMenuScreen.

### show

```
public void show()
```

Called when the screen becomes chosen as the screen for the game.

**See Also:**

`Screen.show()`

### pause

```
public void pause()
```

Called when the application is left on Android or when the game is exitted. Saves player information to the hard drive in case of application quit.

### resume

```
public void resume()
```

Called when the user re-opens the application after having left it.

**See Also:**

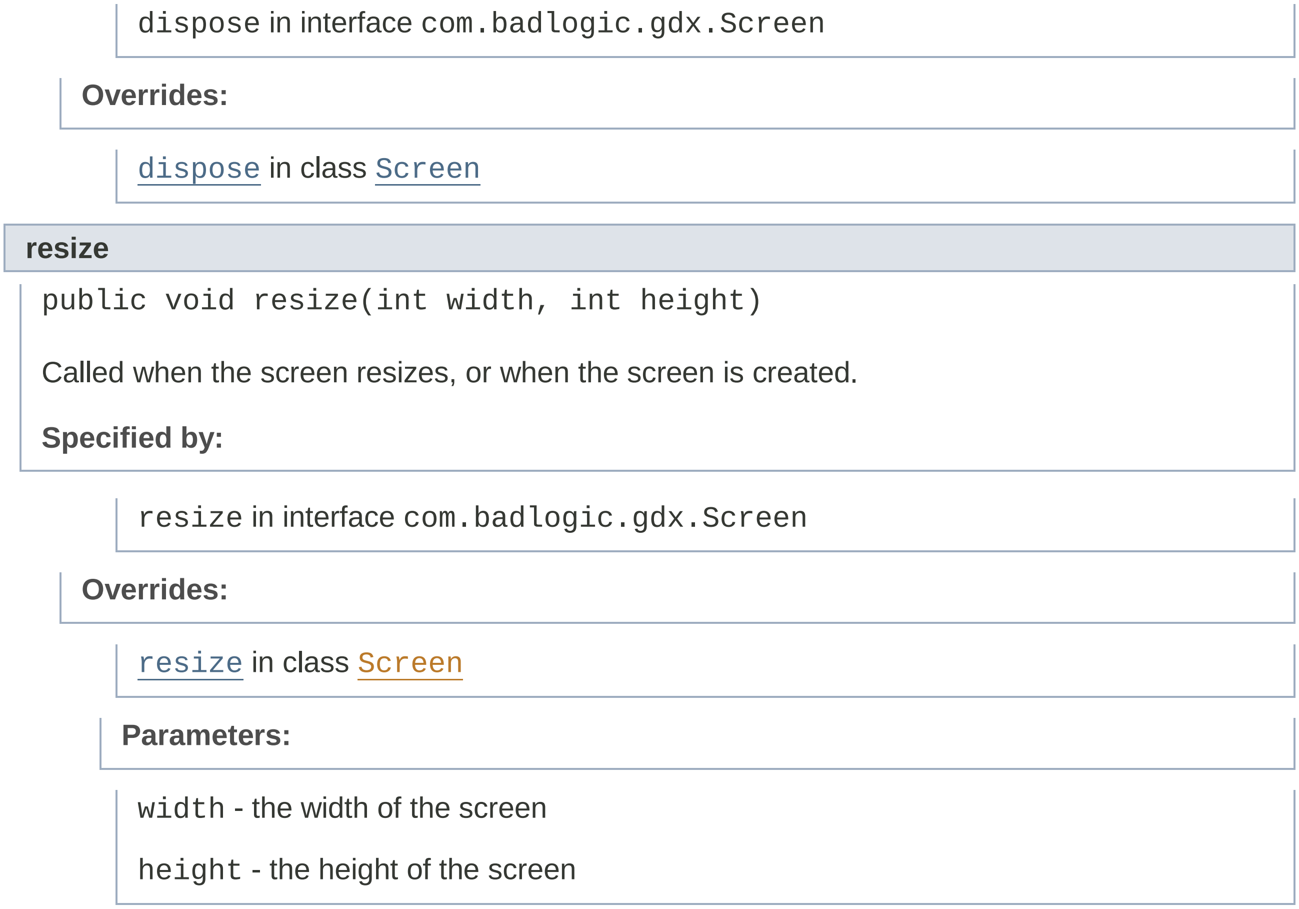
`Screen.resume()`

### dispose

```
public void dispose()
```

Called when the application closes, or when the user leaves the screen.

**Specified by:**



`dispose` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`dispose` in class `Screen`

### resize

`public void resize(int width, int height)`

Called when the screen resizes, or when the screen is created.

**Specified by:**

`resize` in interface `com.badlogic.gdx.Screen`

**Overrides:**

`resize` in class `Screen`

**Parameters:**

`width` - the width of the screen

`height` - the height of the screen

(See next page for *Hud* class*)*

II.3.1.30 *Hud*

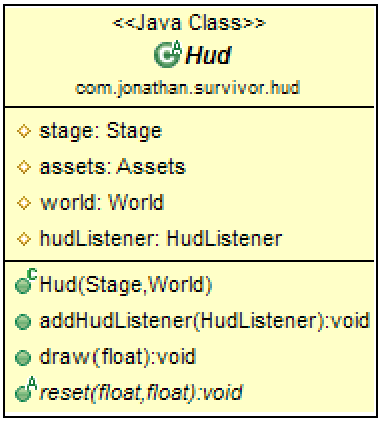


**Figure 49 : Hud class diagram**

The *Hud* abstract class is used to display a heads-up display while the player is inside the game world. It allows to overlay the world with a UI widgets which can be clicked to change the world's game logic appropriately.

The first member variable in this class is the very familiar *stage*. Much like in the *Screen* subclasses, it acts as a container for widgets, which it then draws to the screen. Then, the *Hud* class holds the *assets* singleton which allows all heads-up displays to have access to the visual assets needed to render the UI widgets.

The *Hud* also has a reference to the *World* instance, which controls game logic. Subclasses can call the world's methods according to certain button presses. For instance, if the left arrow button is pressed in the *ExplorationHud*, the *World.walk(Player, Direction)* method is called.

Most importantly, the *Hud* holds a *HudListener* instance. The *HudListener* registered to this class will receive methods pertinent to button presses in the HUD. In this program, the *GameScreen.uiListener* instance is registered to this class. Thus, the *GameScreen* receives all HUD events and acts correspondingly.

The only constructor in the *Hud* class accepts the *Stage* used to draw the HUD's widgets, and the *World* to which it can call methods.

The *draw(float)* method, on the other hand, accepts a *deltaTime* parameter, and renders all of the 2d widgets to the screen with the use of the *Stage* instance. Finally, the abstract *reset(float, float)* method is called whenever the screen is resized. Its purpose is to resize the stage used by the class. It accepts the width and height of the GUI in pixels in order to re-scale widgets accordingly.

```
public abstract class Hud
extends java.lang.Object
```

## Field Detail

### stage

```
protected com.badlogic.gdx.scenes.scene2d.Stage stage
```

Stores the stage where 2d widgets will be placed and drawn.

### assets

```
protected Assets assets
```

Stores the Assets singleton of the game used to fetch assets to draw the HUD.

### world

```
protected World world
```

Stores the world that any Hud elements can call methods from in case of a button press.

### hudListener

```
protected HudListener hudListener
```

Stores the Listener where Hud events are delegated.

## Constructor Detail

### Hud

```
public Hud(com.badlogic.gdx.scenes.scene2d.Stage stage, World world)
```

Accepts the stage where 2d widgets will be contained and drawn, and the world, where input events will be dispatched.

## Method Detail

### addHudListener

```
public void addHudListener(HudListener hudListener)
```

Registers the listener where Hud events will be delegated.

### draw

```
public void draw(float deltaTime)
```

Draws the Hud to the screen using the stage.

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

### reset

```
public abstract void reset(float guiWidth, float guiHeight)
```

Resets the widgets on the stage. Called when screen is resized. Given parameters are the size that the Hud should occupy in pixels.

**Parameters:**

`guiWidth` - The width in pixels that the gui should occupy.

`guiHeight` - the height in pixels that the gui should occupy.

(See next page for *CombatHud* & *ExplorationHud)*

### II.3.1.31 *ExplorationHud & CombatHud*

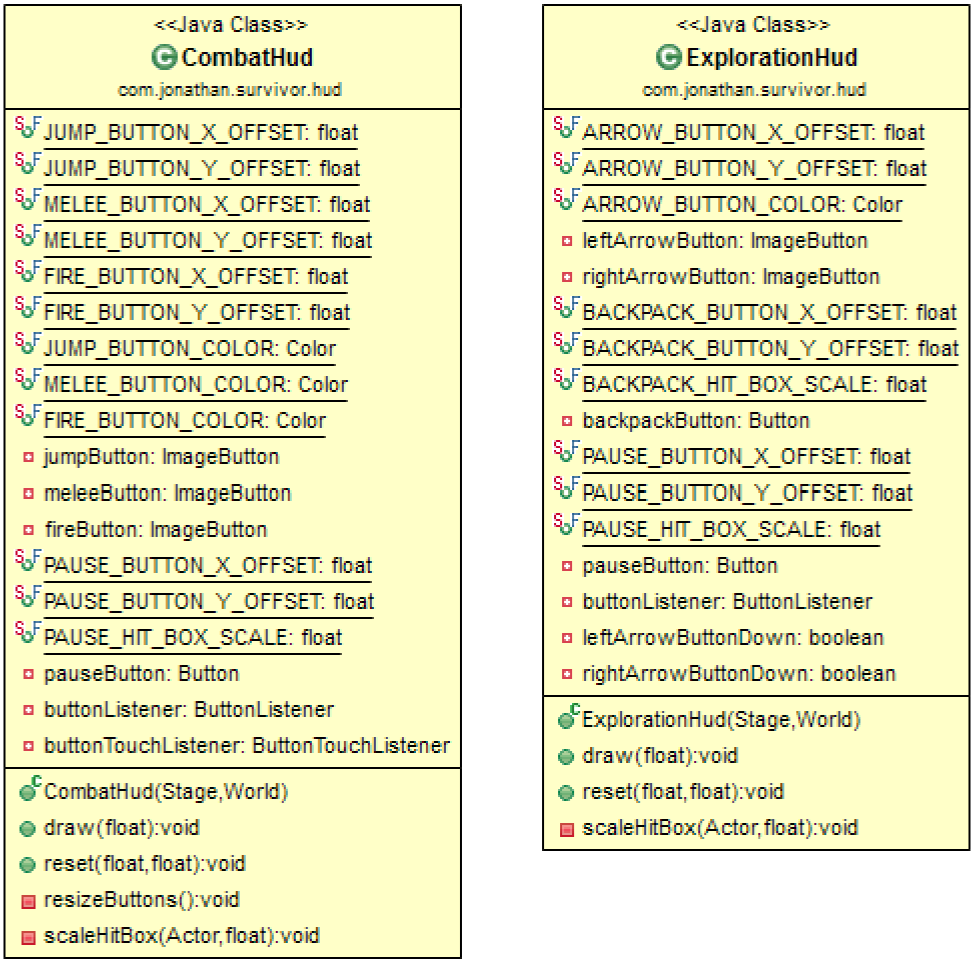

<<Java Class>>
CombatHud
com.jonathan.survivor.hud
JUMP_BUTTON_X_OFFSET: float
JUMP_BUTTON_Y_OFFSET: float
MELEE_BUTTON_X_OFFSET: float
MELEE_BUTTON_Y_OFFSET: float
FIRE_BUTTON_X_OFFSET: float
FIRE_BUTTON_Y_OFFSET: float
JUMP_BUTTON_COLOR: Color
MELEE_BUTTON_COLOR: Color
FIRE_BUTTON_COLOR: Color
jumpButton: ImageButton
meleeButton: ImageButton
fireButton: ImageButton
PAUSE_BUTTON_X_OFFSET: float
PAUSE_BUTTON_Y_OFFSET: float
PAUSE_HIT_BOX_SCALE: float
pauseButton: Button
buttonListener: ButtonListener
buttonTouchListener: ButtonTouchListener
CombatHud(Stage,World)
draw(float):void
reset(float,float):void
resizeButtons():void
scaleHitBox(Actor,float):void
<<Java Class>>
ExplorationHud
com.jonathan.survivor.hud
ARROW_BUTTON_X_OFFSET: float
ARROW_BUTTON_Y_OFFSET: float
ARROW_BUTTON_COLOR: Color
leftArrowButton: ImageButton
rightArrowButton: ImageButton
BACKPACK_BUTTON_X_OFFSET: float
BACKPACK_BUTTON_Y_OFFSET: float
BACKPACK_HIT_BOX_SCALE: float
backpackButton: Button
PAUSE_BUTTON_X_OFFSET: float
PAUSE_BUTTON_Y_OFFSET: float
PAUSE_HIT_BOX_SCALE: float
pauseButton: Button
buttonListener: ButtonListener
leftArrowButtonDown: boolean
rightArrowButtonDown: boolean
ExplorationHud(Stage,World)
draw(float):void
reset(float,float):void
scaleHitBox(Actor,float):void

**Figure 50 : ExplorationHud and CombatHud class diagrams**

These HUDs are displayed whenever the player is either exploring the world or fighting a zombie. Their appearance is shown in *section II.3 GUI*. The variable of these classes are simply the widgets used to display the HUD.

The constructors of these two classes accept the *Stage* instance used to draw its widgets, and the *World* to which it will delegate method calls. For instance, when the player presses the *rightArrowButton*, the *ExplorationHud* will call the *World.walk (Player, Direction)* method.

```
public class CombatHud
extends Hud
```

## Field Detail

### JUMP_BUTTON_X_OFFSET

```
public static final float JUMP_BUTTON_X_OFFSET
```

Stores the x-offset of the jump button. Used to anchor the button relative to the bottom-left of the screen.

### JUMP_BUTTON_Y_OFFSET

```
public static final float JUMP_BUTTON_Y_OFFSET
```

Stores the y-offset of the jump button. Used to anchor the button relative to the bottom-left of the screen.

### MELEE_BUTTON_X_OFFSET

```
public static final float MELEE_BUTTON_X_OFFSET
```

Stores the x-offset of the melee button. Used to anchor the button relative to the bottom-right of the screen.

### MELEE_BUTTON_Y_OFFSET

```
public static final float MELEE_BUTTON_Y_OFFSET
```

Stores the y-offset of the melee button. Used to anchor the button relative to the bottom-right of the screen.

### FIRE_BUTTON_X_OFFSET

```
public static final float FIRE_BUTTON_X_OFFSET
```

Stores the x-offset of the fire button. Used to anchor the button relative to the bottom-right of the screen.

### FIRE_BUTTON_Y_OFFSET

```
public static final float FIRE_BUTTON_Y_OFFSET
```

Stores the y-offset of the fire button. Used to anchor the button relative to the bottom-right of the screen.

### JUMP_BUTTON_COLOR

```
public static final com.badlogic.gdx.graphics.Color JUMP_BUTTON_COLOR
```

Stores the color of the jump button.

### MELEE_BUTTON_COLOR

```
public static final com.badlogic.gdx.graphics.Color MELEE_BUTTON_COLOR
```

Stores the color of the melee button.

### FIRE_BUTTON_COLOR

```
public static final com.badlogic.gdx.graphics.Color FIRE_BUTTON_COLOR
```

Stores the color of the fire button.

### jumpButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton jumpButton
```

Stores the button used to make the player jump.

### meleeButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton meleeButton
```

Holds the button used to make the player melee with his weapon.

### fireButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton fireButton
```

Holds the button used to make the player fire his ranged weapon.

### PAUSE_BUTTON_X_OFFSET

```
public static final float PAUSE_BUTTON_X_OFFSET
```

Stores the x-offset of the pause button. Used to anchor the button to the top-right corner of the screen with a given offset.

### PAUSE_BUTTON_Y_OFFSET

```
public static final float PAUSE_BUTTON_Y_OFFSET
```

Stores the y-offset of the pause button. Used to anchor the button to the top-right corner of the screen with a given offset.

### PAUSE_HIT_BOX_SCALE

```
public static final float PAUSE_HIT_BOX_SCALE
```

Holds the scale of the pause button's hit box. Allows for easier clicking.

### pauseButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button pauseButton
```

Stores the Pause Button, used to pause the game.

### buttonListener

```
private CombatHud.ButtonListener buttonListener
```

Stores the listener used to listen for events from the arrow buttons.

### buttonTouchListener

```
private CombatHud.ButtonTouchListener buttonTouchListener
```

Holds the ButtonTouchListener, used to recognize the button up and down events coming from the fire button.

## Constructor Detail

### CombatHud

```
public CombatHud(com.badlogic.gdx.scenes.scene2d.Stage stage,
                 World world)
```

Accepts the stage where 2d widgets will be drawn and placed, and the world, which will receive information about button presses.

**Parameters:**

`stage` - the stage where the HUD widgets are placed

`world` - the world, whose method are called whenever the HUD widgets need to interact with the world

## Method Detail

### draw

```
public void draw(float deltaTime)
```

**Description copied from class: Hud**

Draws the Hud to the screen using the stage.

**Overrides:**

`draw` in class `Hud`

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

**See Also:**

`Hud.draw(float)`

### reset

```
public void reset(float guiWidth, float guiHeight)
```

Called when the stage must be reset to draw the widgets contained in this class. Used when the stage needs to be re-purposed. Also called when the screen is resized to re-place the widgets.

**Specified by:**

`reset` in class `Hud`

**Parameters:**

`guiWidth` - The width in pixels that the gui should occupy.

`guiHeight` - the height in pixels that the gui should occupy.

### disableUselessButtons

```
private void disableUselessButtons()
```

Disable any buttons which the user cannot press, such as the meleeButton, if the user has no melee weapon equipped.

### resizeButtons

```
private void resizeButtons()
```

Resizes all of the buttons that need resizing. Ensures that the buttons' contents are all well scaled with no deformities.

### scaleHitBox

```
private void scaleHitBox(
                      com.badlogic.gdx.scenes.scene2d.Actor actor,
                      float scale)
```

Scales the bounds of the actor by the given amount. Allows to re-scale the bounding boxes of a button. Note that the re-scaled bounds are centered on the actor.

**Parameters:**

`actor` - the actor whose hit box will be scaled

`scale` - the multiplier by which to scale the actor's hit box

```
public class ExplorationHud
```

```
extends Hud
```

## Field Detail

### ARROW_BUTTON_X_OFFSET

```
public static final float ARROW_BUTTON_X_OFFSET
```

Stores the x-offset of both arrow buttons. Used to anchor the buttons to the corners of the screen.

### ARROW_BUTTON_Y_OFFSET

```
public static final float ARROW_BUTTON_Y_OFFSET
```

Stores the y-offset of both arrow buttons. Used to anchor the buttons to the corners of the screen.

### ARROW_BUTTON_COLOR

```
public static final com.badlogic.gdx.graphics.Color ARROW_BUTTON_COLOR
```

Stores the color of the arrow buttons.

### leftArrowButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton leftArrowButton
```

Stores the left and right arrow buttons to make the player move left and right.

### rightArrowButton

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageButton
rightArrowButton
```

Stores the left and right arrow buttons to make the player move left and right.

### BACKPACK_BUTTON_X_OFFSET

```
public static final float BACKPACK_BUTTON_X_OFFSET
```

Stores the x-offset of the backpack button. Used to anchor the button to the corner of the screen with a given offset.

### BACKPACK_BUTTON_Y_OFFSET

```
public static final float BACKPACK_BUTTON_Y_OFFSET
```

Stores the y-offset of the backpack button. Used to anchor the button to the corner of the screen with a given offset.

### BACKPACK_HIT_BOX_SCALE

```
public static final float BACKPACK_HIT_BOX_SCALE
```

Holds the scale of the backpack button's hit box. Allows for easier clicking.

### backpackButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button backpackButton
```

Stores the Backpack button.

### PAUSE_BUTTON_X_OFFSET

```
public static final float PAUSE_BUTTON_X_OFFSET
```

Stores the x-offset of the pause button. Used to anchor the button to the top-right corner of the screen with a given offset.

### PAUSE_BUTTON_Y_OFFSET

```
public static final float PAUSE_BUTTON_Y_OFFSET
```

Stores the y-offset of the pause button. Used to anchor the button to the top-right corner of the screen with a given offset.

### PAUSE_HIT_BOX_SCALE

```
public static final float PAUSE_HIT_BOX_SCALE
```

Holds the scale of the pause button's hit box. Allows for easier clicking.

### pauseButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button pauseButton
```

Stores the Pause Button, used to pause the game.

### buttonListener

```
private ExplorationHud.ButtonListener buttonListener
```

Stores the listener used to listen for events from the arrow buttons.

### leftArrowButtonDown

```
private boolean leftArrowButtonDown
```

Stores the buttons displaying the left arrow to move the player.

### rightArrowButtonDown

```
private boolean rightArrowButtonDown
```

Stores the buttons displaying the right arrow to move the player.

## Constructor Detail

### ExplorationHud

```
public ExplorationHud(com.badlogic.gdx.scenes.scene2d.Stage stage,
                                      World world)
```

Accepts the stage where 2d widgets will be drawn and placed, and the world, which will receive information about button presses.

**Parameters:**

`stage` - the stage where the HUD widgets are drawn `world` - the world, whose methods are called when the HUD should change world data

## Method Detail

### draw

```
public void draw(float deltaTime)
```

**Description copied from class: Hud**

Draws the Hud to the screen using the stage.

**Overrides:**

`draw` in class `Hud`

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

**See Also:**

`Hud.draw(float)`

### reset

```
public void reset(float guiWidth, float guiHeight)
```

Called when the stage must be reset to draw the widgets contained in this class. Used when the stage needs to be re-purposed. Also called when the screen is resized to re-place the widgets.

**Specified by:**

`reset` in class `Hud`

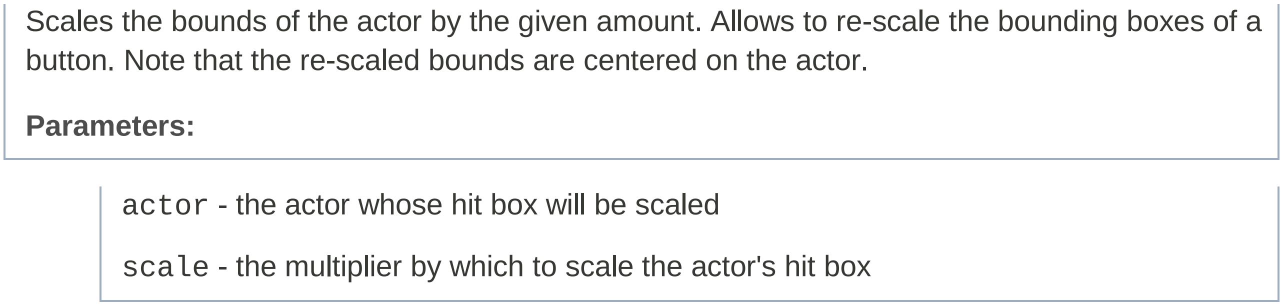
**Parameters:**

`guiWidth` - The width in pixels that the gui should occupy.

`guiHeight` - the height in pixels that the gui should occupy.

### scaleHitBox

```
private void scaleHitBox(com.badlogic.gdx.scenes.scene2d.Actor actor,
                                        float scale)
```



Scales the bounds of the actor by the given amount. Allows to re-scale the bounding boxes of a button. Note that the re-scaled bounds are centered on the actor.

**Parameters:**

`actor` - the actor whose hit box will be scaled

`scale` - the multiplier by which to scale the actor's hit box

II.3.1.32 *BackpackHud & CraftingHud*

The *BackpackHud* whose UML diagram is shown in the figure in the next page is displayed when the player presses the backpack button in the *Exploration HUD.* From the backpack HUD, the player can access the crafting HUD by pressing on the crafting button. The appearance of the *BackpackHud* is shown in the *Backpack Menu* entry in *section II.2 GUI.* On the other hand, the *CraftingHud's* appearance can be reffered to inside the *Crafting Menu* entry in the same section.

In terms of member variables, both classes first hold the *backpackBg* Image, which is used to display the background behind the widgets. Next, both classes, have a *header* Label, which shows the title of the menu at the top of the screen. Next, the *BackpackHud* holds the *survivalGuideButton*, which transitions to the *Survival Guide Menu*, and the *craftingButton*, which transitions to the *Crafting Menu* when pressed. Note that the two buttons are proceeded by two labels. The first, *survivalGuideLabel*, displays the text under the *survivalGuideButton*. The second, *craftingLabel*, displays the text under the *craftingButton*. The buttons and their labels were separated so that they could be positioned more easily. Next, both classes have the *backButton*, which is used to switch to the previous HUD where the user was located before switching to the current HUD. Then, both classes also have a *Table* instance. This *table* variable allows every button and label to be arranged neatly inside the HUD. It acts essentially like a *GridLayout* in Java's Swing framework.

The *CraftingHud* contains slightly more data fields. For instance, it holds a reference to the player's *inventory*. This allows the *CraftingHud* to extract the information for every item in the player's inventory.

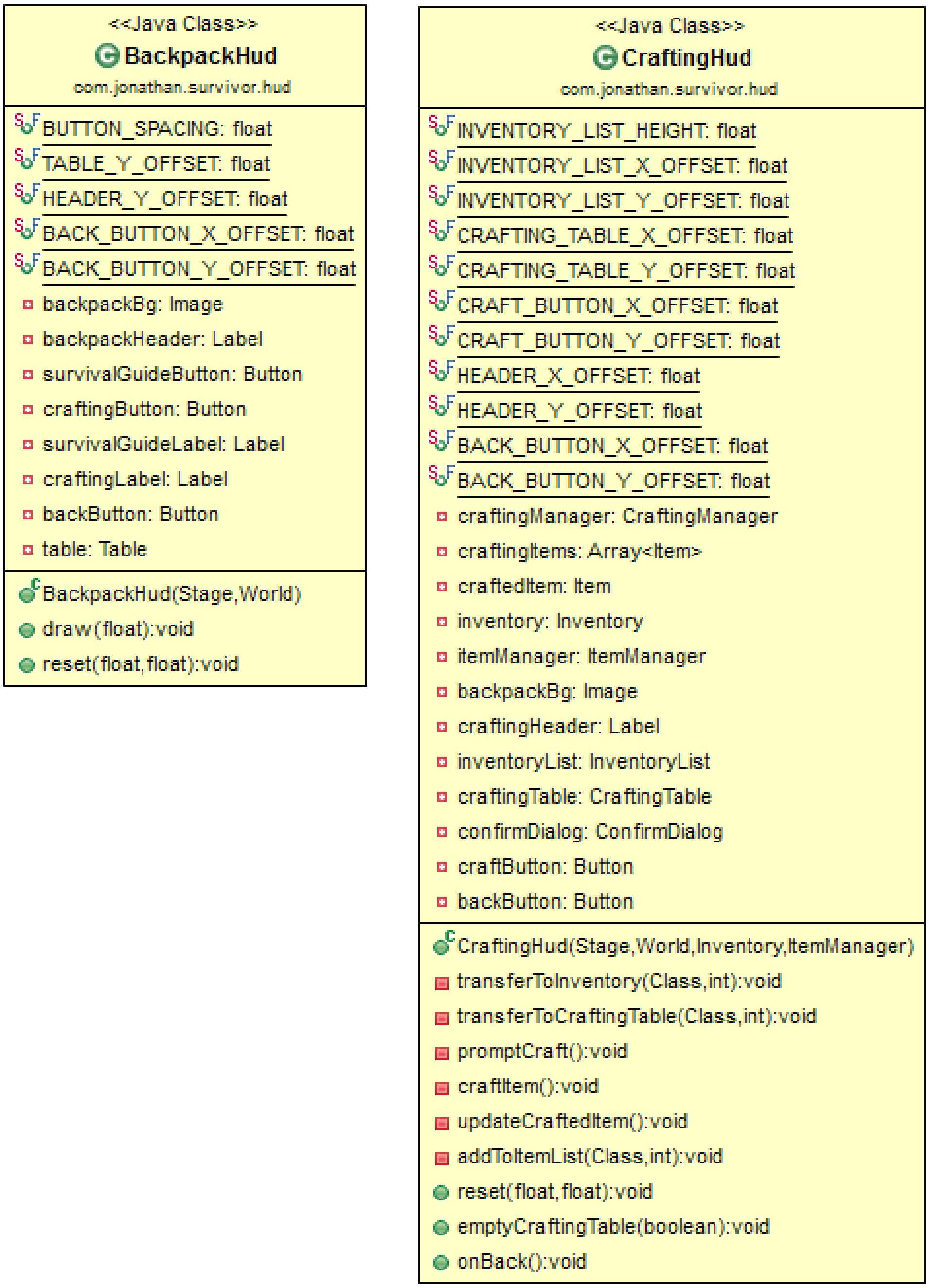

<<Java Class>>
BackpackHud
com.jonathan.survivor.hud
BUTTON_SPACING: float
TABLE_Y_OFFSET: float
HEADER_Y_OFFSET: float
BACK_BUTTON_X_OFFSET: float
BACK_BUTTON_Y_OFFSET: float
backpackBg: Image
backpackHeader: Label
survivalGuideButton: Button
craftingButton: Button
survivalGuideLabel: Label
craftingLabel: Label
backButton: Button
table: Table
BackpackHud(Stage,World)
draw(float):void
reset(float,float):void
<<Java Class>>
CraftingHud
com.jonathan.survivor.hud
INVENTORY_LIST_HEIGHT: float
INVENTORY_LIST_X_OFFSET: float
INVENTORY_LIST_Y_OFFSET: float
CRAFTING_TABLE_X_OFFSET: float
CRAFTING_TABLE_Y_OFFSET: float
CRAFT_BUTTON_X_OFFSET: float
CRAFT_BUTTON_Y_OFFSET: float
HEADER_X_OFFSET: float
HEADER_Y_OFFSET: float
BACK_BUTTON_X_OFFSET: float
BACK_BUTTON_Y_OFFSET: float
craftingManager: CraftingManager
craftingItems: Array<Item>
craftedItem: Item
inventory: Inventory
itemManager: ItemManager
backpackBg: Image
craftingHeader: Label
inventoryList: InventoryList
craftingTable: CraftingTable
confirmDialog: ConfirmDialog
craftButton: Button
backButton: Button
CraftingHud(Stage,World,Inventory,ItemManager)
transferToInventory(Class,int):void
transferToCraftingTable(Class,int):void
promptCraft():void
craftItem():void
updateCraftedItem():void
addToItemList(Class,int):void
reset(float,float):void
emptyCraftingTable(boolean):void
onBack():void

**Figure 51 : BackPackHud and CraftingHud class diagrams**

The constructors of these two classes accept the *Stage* instance used to draw its widgets, and the *World* to which it will delegate method calls.

```
public class BackpackHud
extends Hud
```

## Field Detail

### BUTTON_SPACING

```
public static final float BUTTON_SPACING
```

Stores the spacing between the buttons in the middle of the backpack.

### TABLE_Y_OFFSET

```
public static final float TABLE_Y_OFFSET
```

Stores the y position of the table nudge the table up, so that the header is at the right position on the backpack.

**See Also:**

### HEADER_Y_OFFSET

```
public static final float HEADER_Y_OFFSET
```

Stores the offset between the bottom of the "Backpack" header and the top of the buttons. Adds spacing between the header and the buttons.

**See Also:**

### BACK_BUTTON_X_OFFSET

```
public static final float BACK_BUTTON_X_OFFSET
```

Stores the x-offset used to anchor the back button to the bottom-right of the screen with a certain padding.

**See Also:**

### BACK_BUTTON_Y_OFFSET

```
public static final float BACK_BUTTON_Y_OFFSET
```

Stores the y-offset used to anchor the back button to the bottom-right of the screen with a certain padding.

### backpackBg

```
private com.badlogic.gdx.scenes.scene2d.ui.Image backpackBg
```

Stores the image of the backpack background.

### backpackHeader

```
private com.badlogic.gdx.scenes.scene2d.ui.Label backpackHeader
```

Stores the header displaying "Backpack" on top of the Hud.

### survivalGuideButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button survivalGuideButton
```

Stores the buttons displayed on the Backpack Hud, including the SurvivalGuide, and Crafting buttons.

### craftingButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button craftingButton
```

### survivalGuideLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label survivalGuideLabel
```

Stores the labels displaying the names of each main button.

### craftingLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label craftingLabel
```

### backButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button backButton
```

Stores the back button, used to exit out of the backpack hud.

### table

```
private com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Stores the Table instance where buttons are organized in a grid-like fashion.

## Constructor Detail

### BackpackHud

```
public BackpackHud(com.badlogic.gdx.scenes.scene2d.Stage stage,
                   World world)
```

Accepts the Stage instance where widgets are drawn, and the world, used to manipulate the world according to button presses.

**Parameters:**

`stage` - the stage where the HUD widgets are placed

`world` - the world, whose method are called whenever the HUD widgets need to interact with the world

## Method Detail

### draw

```
public void draw(float deltaTime)
```

**Description copied from class: Hud**

Draws the Hud to the screen using the stage.

**Overrides:**

`draw` in class `Hud`

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

### reset

```
public void reset(float guiWidth,
                  float guiHeight)
```

Called whenever the current Hud of the game switches to the backpack Hud. Used to reset the stage to hold the widgets of the Backpack Hud. Also called when the screen is resized. Thus, any widgets can be repositioned or rescaled accordingly.

**Specified by:**

`reset` in class `Hud`

**Parameters:**

`guiWidth` - The width in pixels that the gui should occupy.

`guiHeight` - the height in pixels that the gui should occupy.

The *CraftingHud* allows the user to craft resources using the items in his inventory. It contains a *CraftingTable* and an *InventoryList*, which control most of its functionalities.

```
public class CraftingHud
extends Hud
```

## Field Detail

### INVENTORY_LIST_HEIGHT

```
public static final float INVENTORY_LIST_HEIGHT
```

Holds the height of the inventory list holding all of the item buttons.

### INVENTORY_LIST_X_OFFSET

```
public static final float INVENTORY_LIST_X_OFFSET
```

Holds the offset used to anchor the inventory list to the left of the backpack background. Offsets the list relative to the center of the screen.

### INVENTORY_LIST_Y_OFFSET

```
public static final float INVENTORY_LIST_Y_OFFSET
```

Holds the offset used to anchor the inventory list to the left of the backpack background. Offsets the list relative to the center of the screen.

### CRAFTING_TABLE_X_OFFSET

```
public static final float CRAFTING_TABLE_X_OFFSET
```

Holds the x-offset used to anchor the crafting table to the left of the backpack background. Offsets the table relative to the center of the screen.

### CRAFTING_TABLE_Y_OFFSET

```
public static final float CRAFTING_TABLE_Y_OFFSET
```

Holds the y-offset used to anchor the crafting table to the left of the backpack background. Offsets the table relative to the center of the screen.

### CRAFT_BUTTON_X_OFFSET

```
public static final float CRAFT_BUTTON_X_OFFSET
```

Stores the x-offset used to anchor the craftButton to the bottom-right of the backpack background. Offsets the button relative to the bottom-right of the background

### CRAFT_BUTTON_Y_OFFSET

```
public static final float CRAFT_BUTTON_Y_OFFSET
```

Stores the offset used to anchor the craftButton to the bottom-right of the backpack background. Offsets the button relative to the bottom-right of the background

### HEADER_X_OFFSET

```
public static final float HEADER_X_OFFSET
```

Holds the x-offset used to anchor the header to the top-center of the screen. Offsets the header relative to the center of the screen.

### HEADER_Y_OFFSET

```
public static final float HEADER_Y_OFFSET
```

Holds the y-offset used to anchor the header to the top-center of the screen. Offsets the header relative to the center of the screen.

### BACK_BUTTON_X_OFFSET

```
public static final float BACK_BUTTON_X_OFFSET
```

Stores the offset used to anchor the back button to the bottom-right of the backpack background with a certain padding.

### BACK_BUTTON_Y_OFFSET

```
public static final float BACK_BUTTON_Y_OFFSET
```

Stores the offset used to anchor the back button to the bottom-right of the backpack background with a certain padding.

### craftingManager

```
private CraftingManager craftingManager
```

Holds the CraftingManager singleton which dictates whether or not an item combination forms a certain item.

### craftingItems

```
private com.badlogic.gdx.utils.Array<CraftingManager.Item>
craftingItems
```

Holds an array of each item and their quantity inside the crafting table. Used to dictate if the item combination forms another item.

### craftedItem

```
private CraftingManager.Item craftedItem
```

Stores the item crafted from the items currently inside the crafting table.

### inventory

```
private Inventory inventory
```

Stores the player's inventory in order to populate the items in the inventory list.

### itemManager

```
private ItemManager itemManager
```

Holds the ItemManager instance. Its purpose is to give access to Item instances, which give an item's information and sprite.

### backpackBg

```
private com.badlogic.gdx.scenes.scene2d.ui.Image backpackBg
```

Stores the image of the backpack background.

### craftingHeader

```
private com.badlogic.gdx.scenes.scene2d.ui.Label craftingHeader
```

Stores the header displaying "Crafting" on top of the Hud.

### inventoryList

```
private InventoryList inventoryList
```

Stores the list containing all the items in the player's inventory.

### craftingTable

```
private CraftingTable craftingTable
```

Holds the crafting table which holds a grid of the items being crafted.

### confirmDialog

```
private ConfirmDialog confirmDialog
```

Holds the ConfirmDialog which prompts the user when he wants to craft an item.

### craftButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button craftButton
```

Stores the button displaying "Craft". When pressed, the items in the crafting table combine to create a new item.

### backButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button backButton
```

Stores the back button, used to exit out of the crafting menu.

## Constructor Detail

### CraftingHud

```
public CraftingHud(com.badlogic.gdx.scenes.scene2d.Stage stage,
                   World world, Inventory inventory,
                   ItemManager itemManager)
```

Accepts the Stage instance where widgets are drawn, and the world, used to manipulate the world according to button presses. The ItemManager is accepted in order to fetch internally-pooled Item instances in order to draw the items in the player's inventory.

**Parameters**

`stage` - the stage where the HUD widgets are drawn

`world` - the world, whose methods are called when the HUD should change world data

`inventory` - the inventory, used to populate the list of items in the left-hand list

`itemManager` - the item manager, used to accesses Item instances and sprites, in order to display certain items in the HUD

## Method Detail

### transferToInventory

```
private void transferToInventory(java.lang.Class itemClass,
                                 int quantity)
```

Transfers the given quantity of the item from the crafting table to the inventory list.

**Parameters:**

`itemClass` - the item type to transfer to the inventory list

`quantity` - the quantity of that item to place in the inventory list

### transferToCraftingTable

```
private void transferToCraftingTable(java.lang.Class itemClass,
                                     int quantity)
```

Transfers the given quantity of the given item from the inventory list to the crafting table.

**Parameters:**

`itemClass` - the item type to transfer to the crafting table

`quantity` - the quantity of that item to place in the crafting table

### promptCraft

```
private void promptCraft()
```

Asks the user if he wants to craft the item in the crafting table. Opens up the confirm dialog to ensure of the player's choice.

### craftItem

```
private void craftItem()
```

Crafts the item formed by the items in the crafting table and adds it to the inventory, deleting all the other items that were used to form the item.

### updateCraftedItem

```
private void updateCraftedItem()
```

Updates the item shown in the cell below the item grid. Computes if any items can be crafted using the items in the crafting table. If so, the crafting box below the item grid is updated with the appropriate picture.

### addToItemList

```
private void addToItemList(java.lang.Class itemClass,
                 int quantity)
```

Adds the given item to the array of items in the crafting table. Allows class to determine if items can form another item.

**Parameters:**

`itemClass` - the item type to add to the list of items that the user has dragged to the crafting table

`quantity` - the quantity of that item added to the crafting table

### reset

```
public void reset(float guiWidth,
                      float guiHeight)
```

Called whenever the current Hud of the game switches to the backpack Hud. Used to reset the stage to hold the widgets of the Backpack Hud. Also called when the screen is resized. Thus, any widgets can be repositioned or rescaled accordingly.

**Specified by:**

`reset` in class `Hud`

**Parameters:**

`guiWidth` - The width in pixels that the gui should occupy.

`guiHeight` - the height in pixels that the gui should occupy.

### emptyCraftingTable

```
public void emptyCraftingTable(boolean transferToInventory)
```

Removes the items from the crafting table. If the argument is true, the items are put back into the player's inventory.

### onBack

```
public void onBack()
```

Called when the back button is pressed, or when the game exits. Prompts the gameScreen to return to the backpack menu, and removes the elements in the crafting table back into the inventory.

### II.3.1.33 *SurvivalGuideHud & PauseMenuHud*

The *SurvivalGuideHud* displays the *Survival Guide Menu*, and the *PauseMenuHud* displays the *Pause Menu*, which are both detailed in *section II.3 GUI*.

First, as seen in the class diagram on the next page, both classes have a *table* instance. Much like other *Hud* subclasses, this *table* is used to arrange the UI widgets in a grid-like fashion. Then, the *SurvivalGuideHud* has the *survivalGuideBg*, an image which shows the background of the backpack. Next, both classes have a *header* label, used to display the name of the HUD at the top of the screen.

The *PauseMenuHud* is more straight-forward. In terms of extra widgets, it simply has a *resumeButton*, used to resume the game, and a *mainMenuButton*, used to return to the main menu.

The constructors of these two classes accept the *Stage* instance used to draw its widgets, and the *World* to which it will delegate method calls.

(See next page for class diagrams)

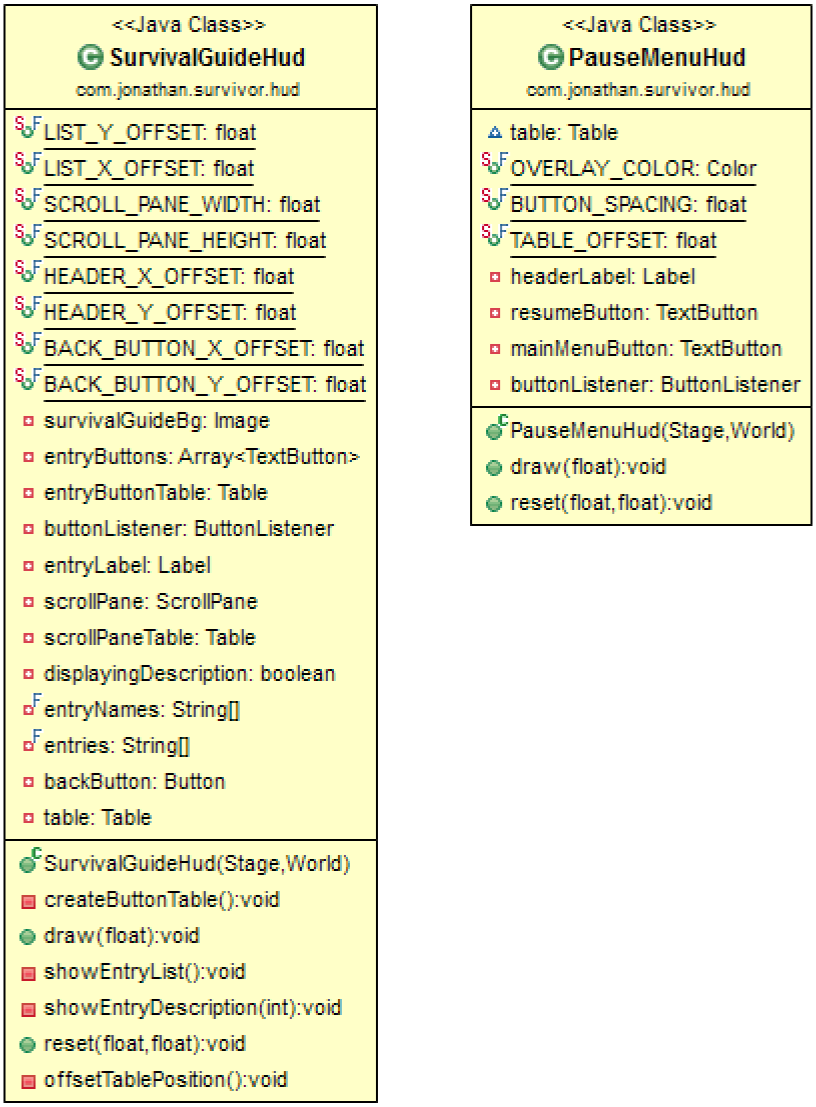


**Figure 52 : SurvivalGuideHud and PauseMenuHud class diagrams**

```
public class SurvivalGuideHud
extends Hud
```

## Field Detail

### LIST_Y_OFFSET

```
public static final float LIST_Y_OFFSET
```

Stores the amount the list is offset up, relative to the center of the screen. This is the top-most y-position of the text, where 0 will place it at the center of the screen.

### LIST_X_OFFSET

```
public static final float LIST_X_OFFSET
```

Holds the amount the list is nudged to the left. This allows the entries in the list to look left-aligned.

### SCROLL_PANE_WIDTH

```
public static final float SCROLL_PANE_WIDTH
```

Holds the width of the scroll pane where entries are displayed in the survival guide.

### SCROLL_PANE_HEIGHT

```
public static final float SCROLL_PANE_HEIGHT
```

Holds the height of the scroll pane where entries are displayed in the survival guide.

### HEADER_X_OFFSET

```
public static final float HEADER_X_OFFSET
```

Stores the x-offset of the "Guide" header relative to the center of the screen.

### HEADER_Y_OFFSET

```
public static final float HEADER_Y_OFFSET
```

Stores the offset between the bottom of the "Guide" header and the top of the listof entries. Adds spacing between the header and the buttons.

### BACK_BUTTON_X_OFFSET

```
public static final float BACK_BUTTON_X_OFFSET
```

Stores the x-offset used to anchor the back button to the bottom-right of the screen with a certain padding.

### BACK_BUTTON_Y_OFFSET

```
public static final float BACK_BUTTON_Y_OFFSET
```

Stores the y-offset used to anchor the back button to the bottom-right of the screen with a certain padding.

### survivalGuideBg

```
private com.badlogic.gdx.scenes.scene2d.ui.Image survivalGuideBg
```

Stores the image of the survival guide's background.

### entryButtons

```
private com.badlogic.gdx.utils.Array<com.badlogic.gdx.scenes.scene2d.u
i.TextButton> entryButtons
```

Stores the list of buttons which the user can press to access an entry in the survival guide.

### entryButtonTable

```
private com.badlogic.gdx.scenes.scene2d.ui.Table entryButtonTable
```

Stores a table containing all of the entry buttons, arranged in a vertical list. The user can scroll through it in a scroll pane and select an entry.

### buttonListener

```
private SurvivalGuideHud.ButtonListener buttonListener
```

Holds the Listener which registers any button clicks. If an entry button is pressed, the correct description for that pressed entry is shown.

### entryLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label entryLabel
```

Holds the label displaying the description for the entry the user clicked.

### scrollPane

```
private com.badlogic.gdx.scenes.scene2d.ui.ScrollPane scrollPane
```

Stores the ScrollPane which allows the items in the survival guide to be scollable.

### scrollPaneTable

```
private com.badlogic.gdx.scenes.scene2d.ui.Table scrollPaneTable
```

Holds the table where the scroll pane is contained. This is the high-level container for the list.

### displayingDescription

```
private boolean displayingDescription
```

True if the description for an entry is currently being shown. On back, revert to the entry name list.

### entryNames

```
private final java.lang.String[] entryNames
```

Holds the list of entry names that the user can choose from the list.

### entries

```
private final java.lang.String[] entries
```

Holds the description of every entry in the survival guide.

### backButton

```
private com.badlogic.gdx.scenes.scene2d.ui.Button backButton
```

Stores the back button, used to exit out of the backpack hud.

### table

```
private com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Stores the Table instance where buttons are organized in a grid-like fashion.

## Constructor Detail

### SurvivalGuideHud

```
public SurvivalGuideHud(com.badlogic.gdx.scenes.scene2d.Stage stage,
                        World world)
```

Accepts the stage where widgets are placed. The passed world is unused for this HUD.

**Parameters:**

`stage` - the stage where the HUD widgets are drawn

`world` - the world, whose methods are called when the HUD should change world data

## Method Detail

### createButtonTable

```
private void createButtonTable()
```

Called upon instantiation to create the table which holds buttons. The user can press these buttons to access entries in the survival guide.

### draw

```
public void draw(float deltaTime)
```

**Description copied from class: Hud**

Draws the Hud to the screen using the stage.

**Overrides:**

`draw` in class `Hud`

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

**See Also:**

`Hud.draw(float)`

### showEntryList

```
private void showEntryList()
```

Displays the list consisting of entry names.

### showEntryDescription

```
private void showEntryDescription(int index)
```

Displays the description for the entry with the given index in the entryNames:String[] array.

**Parameters:**

`index` - the index of the entryNames:String[] array, which determines the string which will be displayed in the survival guide

### reset

```
public void reset(float guiWidth,
                  float guiHeight)
```

Called either when this pause menu is supposed to be displayed, or when the screen is resized. Parameters indicate the size that the HUD should occupy.

**Specified by:**

reset in class Hud

**Parameters:**

guiWidth - The width in pixels that the gui should occupy.

guiHeight - the height in pixels that the gui should occupy.

### offsetTablePosition

```
private void offsetTablePosition()
```

Offsets the position of the table so that the header is at the right position

### backPressed

```
public boolean backPressed()
```

Called by the GameScreen when the BACK key is pressed. If the survival guide is showing the description to an entry, the guide reverts back to the entry list.

**Returns:**

Returns true if the survival guide was displaying the list of entries. If so, upon pressing the back button, the GameScreen is told that the user should be reverted back to the Backpack HUD.

```
public class PauseMenuHud
extends Hud
```

## Field Detail

### table

```
com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Holds the table used to arrange the buttons in a grid-like fashion.

### OVERLAY_COLOR

```
public static final com.badlogic.gdx.graphics.Color OVERLAY_COLOR
```

Holds the color which overlays the screen below the pause menu.

### BUTTON_SPACING

```
public static final float BUTTON_SPACING
```

Stores the spacing between the buttons displayed in a list.

### TABLE_OFFSET

```
public static final float TABLE_OFFSET
```

Stores the amount the table is offset upwards so that the "Paused" label is shown higher up the screen.

### headerLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label headerLabel
```

Holds the header for the pause menu.

### resumeButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton resumeButton
```

Stores the resume button displayed on the pause menu.

### saveButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton saveButton
```

Stores the resume button displayed on the pause menu.

### mainMenuButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton mainMenuButton
```

Stores the quit button displayed on the pause menu.

### saveDialog

```
private ConfirmDialog saveDialog
```

Holds the ConfirmDialog which prompts the user and makes sure he wants to save his profile.

### quitDialog

```
private ConfirmDialog quitDialog
```

Holds the ConfirmDialog which informs the user that his progress will be lost if he quits the game.

### buttonListener

```
private PauseMenuHud.ButtonListener buttonListener
```

Holds the ButtonListener which receives events when one of the pause menu's buttons are pressed.

## Constructor Detail

### PauseMenuHud

```
public PauseMenuHud(com.badlogic.gdx.scenes.scene2d.Stage stage,
                    World world)
```

Accepts the stage where widgets are placed. The passed world is unused for this HUD.

**Parameters:**

`stage` - the stage where the HUD widgets are drawn

`world` - the world, whose methods are called when the HUD should change world data

## Method Detail

### draw

```
public void draw(float deltaTime)
```

**Description copied from class: Hud**

Draws the Hud to the screen using the stage.

**Overrides:**

`draw` in class `Hud`

**Parameters:**

`deltaTime` - the amount of time elapsed since the last render call

**See Also:**

`Hud.draw(float)`

### reset

```
public void reset(float guiWidth,
                  float guiHeight)
```

Called either when this pause menu is supposed to be displayed, or when the screen is resized. Parameters indicate the size that the HUD should occupy.

**Specified by:**

`reset` in class `Hud`

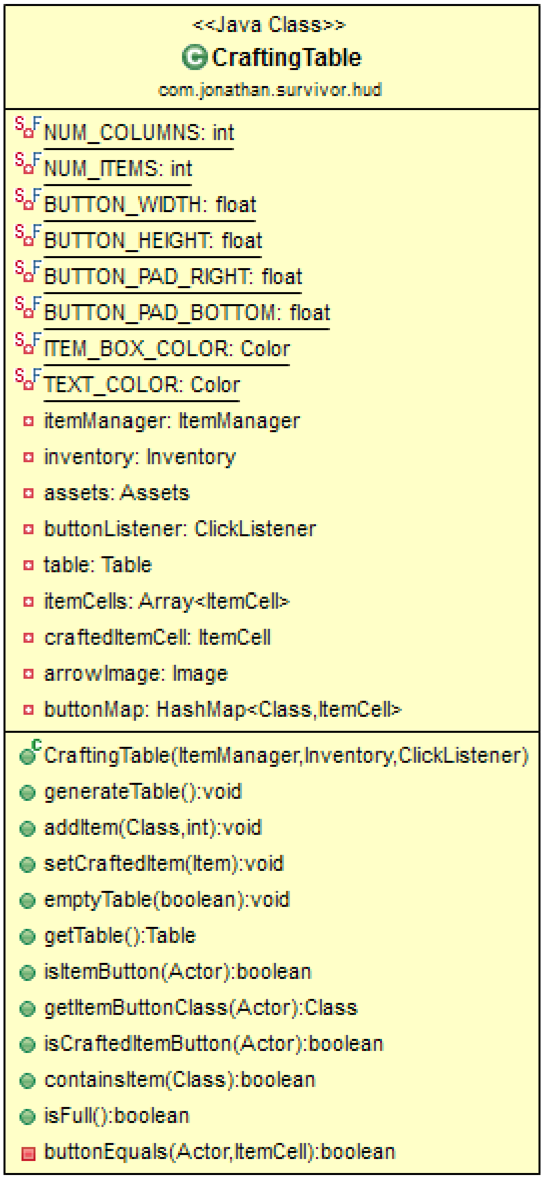
**Parameters:**

`guiWidth` - The width in pixels that the gui should occupy.

`guiHeight` - the height in pixels that the gui should occupy.

(See next page for *CraftingTable* class)

### II.3.1.34 *CraftingTable*



<<Java Class>>
CraftingTable
com.jonathan.survivor.hud

NUM_COLUMNS: int
NUM_ITEMS: int
BUTTON_WIDTH: float
BUTTON_HEIGHT: float
BUTTON_PAD_RIGHT: float
BUTTON_PAD_BOTTOM: float
ITEM_BOX_COLOR: Color
TEXT_COLOR: Color
itemManager: ItemManager
inventory: Inventory
assets: Assets
buttonListener: ClickListener
table: Table
itemCells: Array<ItemCell>
craftedItemCell: ItemCell
arrowImage: Image
buttonMap: HashMap<Class,ItemCell>

CraftingTable(ItemManager,Inventory,ClickListener)
generateTable():void
addItem(Class,int):void
setCraftedItem(Item):void
emptyTable(boolean):void
getTable():Table
isItemButton(Actor):boolean
getItemButtonClass(Actor):Class
isCraftedItemButton(Actor):boolean
containsItem(Class):boolean
isFull():boolean
buttonEquals(Actor,ItemCell):boolean

**Figure 53: CraftingTable class diagram**

The *CraftingTable* displays a table with six item slots and one crafted item slot. This is the class the *CraftingHud* uses to show the grid of six items to the right of the screen.

```
public class CraftingTable
extends java.lang.Object
```

## Field Detail

### NUM_COLUMNS

```
private static final int NUM_COLUMNS
```

Stores the amount of columns of ItemCells in the table.

### NUM_ITEMS

```
private static final int NUM_ITEMS
```

Holds the number of items that can be placed inside the crafting table.

### BUTTON_WIDTH

```
private static final float BUTTON_WIDTH
```

Stores the width of each item button. Note that this is the size of the ItemCells' backgrounds, not of the item's image itself.

### BUTTON_HEIGHT

```
private static final float BUTTON_HEIGHT
```

Stores the height of each item button. Note that this is the size of the ItemCells' backgrounds, not of the item's image itself.

### BUTTON_PAD_RIGHT

```
private static final float BUTTON_PAD_RIGHT
```

Holds the horizontal distance between each item button.

### BUTTON_PAD_BOTTOM

```
private static final float BUTTON_PAD_BOTTOM
```

Holds the vertical distance between each item button.

### ITEM_BOX_COLOR

```
private static final com.badlogic.gdx.graphics.Color ITEM_BOX_COLOR
```

Holds the color of the item box which acts as the small box behind each item sprite.

### TEXT_COLOR

```
private static final com.badlogic.gdx.graphics.Color TEXT_COLOR
```

Stores the color of the text displaying the quantity of each item in the crafting table.

### itemManager

```
private ItemManager itemManager
```

Holds the ItemManager instance from which the Sprites for each item is retrieved.

### inventory

```
private Inventory inventory
```

Holds the inventory from which the list of player items is retrieved.

### assets

```
private Assets assets
```

Holds the universal Assets singleton used to retrieve the visual assets needed to create the inventory list.

### buttonListener

```
private com.badlogic.gdx.scenes.scene2d.utils.ClickListener
buttonListener
```

Stores the ClickListener used by the CraftingHud. All button clicks in the table are delegated to this listener to be handled by the CraftingHud.

### table

```
private com.badlogic.gdx.scenes.scene2d.ui.Table table
```

Stores the table containing all of the items in the crafting table.

### itemCells

```
private com.badlogic.gdx.utils.Array<CraftingTable.ItemCell> itemCells
```

Holds an array containing the six cells which hold an item in the table.

### craftedItemCell

```
private CraftingTable.ItemCell craftedItemCell
```

Stores the ItemCell displaying the item that is crafted as a result of the items in the crafting table.

### arrowImage

```
private com.badlogic.gdx.scenes.scene2d.ui.Image arrowImage
```

Holds the image displaying the arrow below the grid of items.

### buttonMap

```
private java.util.HashMap<java.lang.Class,CraftingTable.ItemCell>
buttonMap
```

Maps an Item subclass with a item cell displaying this item.

## Constructor Detail

### CraftingTable

```
public CraftingTable(ItemManager itemManager, Inventory inventory,

com.badlogic.gdx.scenes.scene2d.utils.ClickListener buttonListener)
```

Accepts the itemManager from which to retrieve the items' sprites, the inventory from which to retrieve the player's items, the ClickListener to which button clicks will be delegated, and the height of the list.

**Parameters:**

`itemManager` - the item manager, used to retrieve item sprites so that they can be displayed on the crafting table

`inventory` - the player's inventory, where items are transfered once they leave the crafting table.

`buttonListener` - the button listener, which receives events once the crafting table's buttons are pressed.

## Method Detail

### generateTable

```
public void generateTable()
```

Called when the crafting table's widgets and its table must be created.

### addItem

```
public void addItem(java.lang.Class itemClass,int quantity)
```

Adds the given amount of the item inside a cell in the crafting table.

**Parameters:**

`itemClass` - the item type to add to the table

`quantity` - the amount to add to the crafting table

### setCraftedItem

```
public void setCraftedItem(CraftingManager.Item craftedItem)
```

Sets the item displayed in craftedItem slot. This is the ItemCell where the crafted item is shown. If null, the slot is emptied.

**Parameters:**

`craftedItem` - the item which will be shown on the slot below the arrow of the crafting table.

### emptyTable

```
public void emptyTable(boolean transferToInventory)
```

Called when the user leaves the crafting menu. Removes all of the items in the crafting table, and places them back into the player's inventory. If the boolean argument is true, the items inside the crafting table are put back in the player's inventory. If not, they are lost.

**Parameters:**

`transferToInventory` - if true, transfer to the items from the crafting table back to the player's inventory

### getTable

```
public com.badlogic.gdx.scenes.scene2d.ui.Table getTable()
```

Returns the table containing all the widgets in the crafting table.

### isItemButton

```
public boolean isItemButton(
                           com.badlogic.gdx.scenes.scene2d.Actor actor)
```

Returns true if the given actor is an item button inside the crafting table. An item button is every button in the grid of six buttons the crafting table.

**Parameters:**

`actor` - the button which is tested to be a item button

**Returns:**

true, if the given button is contained inside the crafting table

### getItemButtonClass

```
public java.lang.Class getItemButtonClass(com.badlogic.gdx.scenes.scen
e2d.Actor actor)
```

Returns the class belonging to the given button. That is, each itemButton in the crafting table represents an item class. This method returns the class represented by the button.

**Parameters:**

`actor` - the itemButton whose item type is returned

**Returns:**

the item class held by the given button

### isCraftedItemButton

```
public boolean isCraftedItemButton(com.badlogic.gdx.scenes.scene2d.Act
or actor)
```

Returns true if the given Actor is the craftedItemButton. That is, the button which contains the preview of the item crafted from the items in the crafting table.

**Parameters:**

`actor` - the itemButton who's tested to be the craftedItemButton

**Returns:**

true, if the given button is the crafted item button

### containsItem

```
public boolean containsItem(java.lang.Class itemClass)
```

Returns true if the given item is contained inside the crafting table.

**Parameters:**

`itemClass` - the item class which is checked to be contained in the table

**Returns:**

true, if the crafting table holds an item of the given type

### isFull

```
public boolean isFull()
```

Returns true if the crafting table cannot take any more items.

### buttonEquals

```
private boolean buttonEquals(
                    com.badlogic.gdx.scenes.scene2d.Actor actor,
                    CraftingTable.ItemCell itemCell)
```

Returns true if the actor corresponds to the itemCell. That is, if the actor is contained inside the itemCell's button, the actor is technically equal to the itemCell. Used to verify whether the actor which delegated a button click corresponds to a certain ItemCell.

**Parameters:**

`actor` - the actor used to check for equivalence

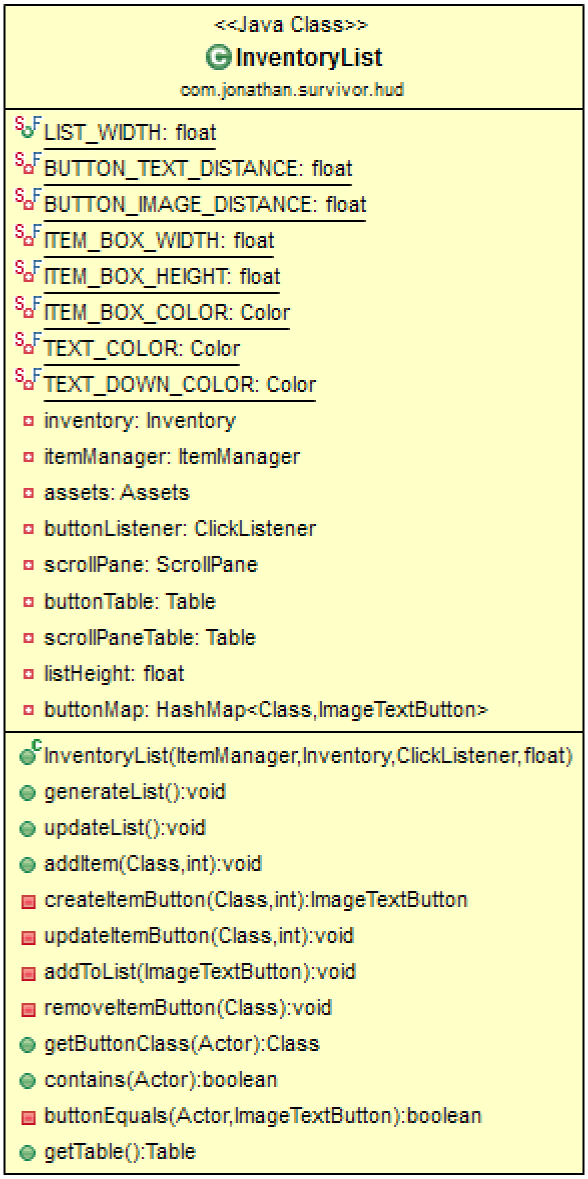
`itemCell` - the item cell used to check for equivalence

**Returns:**

true, if the actor belongs to the item cell

(See next page for *InventoryList* class)

### II.3.1.35 *InventoryList*



```
<<Java Class>>
InventoryList
com.jonathan.survivor.hud

LIST_WIDTH: float
BUTTON_TEXT_DISTANCE: float
BUTTON_IMAGE_DISTANCE: float
ITEM_BOX_WIDTH: float
ITEM_BOX_HEIGHT: float
ITEM_BOX_COLOR: Color
TEXT_COLOR: Color
TEXT_DOWN_COLOR: Color
inventory: Inventory
itemManager: ItemManager
assets: Assets
buttonListener: ClickListener
scrollPane: ScrollPane
buttonTable: Table
scrollPaneTable: Table
listHeight: float
buttonMap: HashMap<Class,ImageTextButton>

InventoryList(ItemManager,Inventory,ClickListener,float)
generateList():void
updateList():void
addItem(Class,int):void
createItemButton(Class,int):ImageTextButton
updateItemButton(Class,int):void
addToList(ImageTextButton):void
removeItemButton(Class):void
getButtonClass(Actor):Class
contains(Actor):boolean
buttonEquals(Actor,ImageTextButton):boolean
getTable():Table
```

**Figure 54: InventoryList class diagram**

The *InventoryList* represents a list displaying the items in the player's inventory. This list is contained inside the *CraftingHud* class, and is used to access the player's inventory, and to add items into the *CraftingTable*.

```
public class InventoryList
extends java.lang.Object
```

## Field Detail

### LIST_WIDTH

```
public static final float LIST_WIDTH
```

Stores the width of the list in pixels for the target (480x320) resolution.

### BUTTON_TEXT_DISTANCE

```
private static final float BUTTON_TEXT_DISTANCE
```

Holds the distance between the left of the button and the left starting point of the text.

### BUTTON_IMAGE_DISTANCE

```
private static final float BUTTON_IMAGE_DISTANCE
```

Holds the distance between the left of the button and the center of each item image.

### ITEM_BOX_WIDTH

```
private static final float ITEM_BOX_WIDTH
```

Stores the width of the item box which acts as a background behind each item image.

### ITEM_BOX_HEIGHT

```
private static final float ITEM_BOX_HEIGHT
```

Stores the height of the item box which acts as a background behind each item image.

### ITEM_BOX_COLOR

```
private static final com.badlogic.gdx.graphics.Color ITEM_BOX_COLOR
```

Holds the color of the item box which acts as the small box behind each item sprite.

### TEXT_COLOR

```
private static final com.badlogic.gdx.graphics.Color TEXT_COLOR
```

Stores the color of the name of each item.

### TEXT_DOWN_COLOR

```
private static final com.badlogic.gdx.graphics.Color TEXT_DOWN_COLOR
```

Stores the color of the name of each item when the item is pressed.

### inventory

```
private Inventory inventory
```

Holds the inventory from which the list of player items is retrieved.

### itemManager

```
private ItemManager itemManager
```

Holds the ItemManager instance from which the Sprites for each item is retrieved.

### assets

```
private Assets assets
```

Holds the universal Assets singleton used to retrieve the visual assets needed to create the inventory list.

### buttonListener

```
private com.badlogic.gdx.scenes.scene2d.utils.ClickListener
buttonListener
```

Stores the ClickListener used by the CraftingHud. All button clicks in the table are delegated to this listener to be handled by the CraftingHud.

### scrollPane

```
private com.badlogic.gdx.scenes.scene2d.ui.ScrollPane scrollPane
```

Stores the ScrollPane which allows the item list stored inside the itemTable to be scrollable.

### buttonTable

```
private com.badlogic.gdx.scenes.scene2d.ui.Table buttonTable
```

Stores the table where all item buttons are placed.

### scrollPaneTable

```
private com.badlogic.gdx.scenes.scene2d.ui.Table scrollPaneTable
```

Holds the table where the scroll pane is contained. This is the high-level container for the list.

### listHeight

```
private float listHeight
```

Holds the height of the inventory list.

### buttonMap

```
private java.util.HashMap<java.lang.Class,com.badlogic.gdx.scenes.scen
e2d.ui.ImageTextButton> buttonMap
```

Maps an Item subclass with a button displaying this item.

## Constructor Detail

### InventoryList

```
public InventoryList(ItemManager itemManager, Inventory inventory,
com.badlogic.gdx.scenes.scene2d.utils.ClickListener buttonListener,
                                                        float height)
```

Accepts the itemManager from which to retrieve the items' sprites, the inventory from which to retrieve the player's items, the ClickListener to which button clicks will be delegated, and the height of the list.

**Parameters:**

`itemManager` - the item manager

`inventory` - the inventory used to populate the list

`buttonListener` - the button listener which receives events whenever a button in the list is pressed

`height` - the height of the list in pixels (relative to the target 480x320 resolution).

## Method Detail

### generateList

```
public void generateList()
```

Populates the list with buttons corresponding to all the items in the player's inventory.

### updateList

```
public void updateList()
```

Called when the contents of the inventory list must be updated. Updates the buttons inside the list, along with their quantities.

### addItem

```
public void addItem(java.lang.Class itemClass,
           int quantity)
```

Adds the given amount of items to the inventory. If a button already exists for the item, the number shown is updated. If the quantity is negative and makes the amount in the inventory zero, the button is deleted.

**Parameters:**

`itemClass` - the item type to add to the list

`quantity` - the amount to add to the list

### createItemButton

```
private
com.badlogic.gdx.scenes.scene2d.ui.ImageTextButton createItemButton(ja
va.lang.Class itemClass, int quantity)
```

Creates a button for the given item, with the given quantity specified in the button's text. Stores it inside the buttonMap HashMap.

**Parameters:**

`itemClass` - the item that this button will display

`quantity` - the number displayed on the button

**Returns:**

a button displaying the given item in the given quantity, which can then be placed in the inventory list

### updateItemButton

```
private void updateItemButton(java.lang.Class itemClass,
                                   int quantity)
```

Updates the quantity displayed on the item button for the given class.

**Parameters:**

`itemClass` - the item whose quantity needs to be changed `quantity` - the new quantity to display on the item button

### addToList

```
private void addToList(com.badlogic.gdx.scenes.scene2d.ui.ImageTextBut
ton itemButton)
```

Adds the given button to the ScrollPane. Note that this method must be called AFTER generateTable() is called.

**Parameters:**

`itemButton` - the item button to place on the inventory list

### removeItemButton

```
private void removeItemButton(java.lang.Class itemClass)
```

Removes the item button from the inventory list. Note that items are refered to by their corresponding class.

**Parameters:**

`itemClass` - the item to remove from the inventory list

### getButtonClass

```
public java.lang.Class getButtonClass(com.badlogic.gdx.scenes.scene2d.
Actor actor)
```

Returns the class of the item which corresponds to the given button in the inventory list.

**Parameters:**

`actor` - the button whose item is returned

**Returns:**

the class of the item which corresponds to the given button

### contains

```
public boolean contains(com.badlogic.gdx.scenes.scene2d.Actor actor)
```

Returns true if the given actor is a button contained inside the inventory. Used by CraftingHud to dictate if a button from the inventory was pressed.

**Parameters:**

`actor` - the actor tested to be in the inventory list

**Returns:**

true, if the given actor is a button in the inventory list

### buttonEquals

```
private boolean buttonEquals(
                     com.badlogic.gdx.scenes.scene2d.Actor actor,
       com.badlogic.gdx.scenes.scene2d.ui.ImageTextButton button)
```

Returns true if the actor corresponds to the button. That is, if the actor is a widget contained inside the button, the actor is considered to be equal to the given button. Used to verify whether the actor which delegated a button click corresponds to a certain button in the inventory.

**Parameters:**

`actor` - the actor used to check for equivalence

`button` - the button used to check for equivalence

**Returns:**

true, if the given actor is a widget inside the given button

### getTable

```
public com.badlogic.gdx.scenes.scene2d.ui.Table getTable()
```

Returns the table containing all of the buttons in the inventory list.

(See next page for *Assets* class)

### II.3.1.36 *Assets*

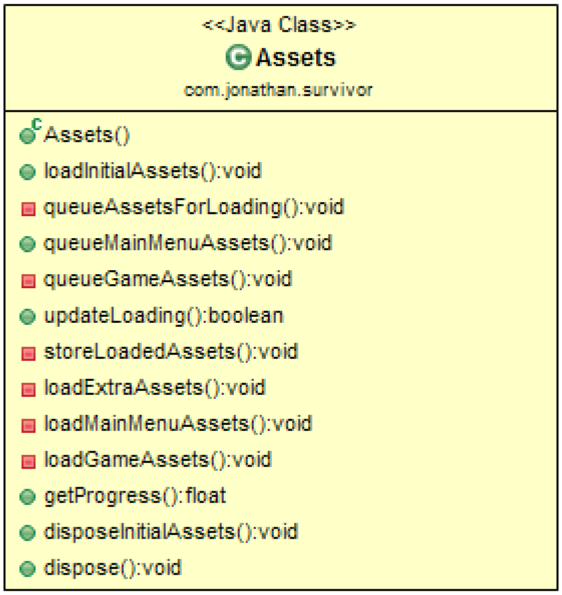


**Figure 55 : Assets class diagram (constructor and methods)**

The *Assets* class is a singleton. That is, only once instance of the class is used inside the program. This instance is accessed using the static *instance:Assets* variable. This allows every class in the program to have access to the visual assets of the game.

Next, the class holds a reference to all visual assets used by the game. They are of the type *BitmapFont* if the variable holds a font, *TextureAtlas* if the variable holds a sprite sheet, or *Sprite* if the variable is holding the reference to an image. Note that the member variables are not listed in the UML diagram nor in the explanations below, as they would be too long to list, and they are irrelevant to the understanding of the source code.

The default constructor is called only once, and simply creates an empty *Assets* object which is held inside the *instance* variable. Before the splash screen is loaded, the *loadInitialAssets()* method is called. This allows the game to load the splash screen's image, along with all images needed for the loading screen. Like this, the game loads as little assets as possible to show the first two screens, preventing the user from having to sit in front of a black screen. On the other hand, the *updateLoading()* method is called on every game tick inside the loading screen. This method uses the *assetManager* to load the visual assets needed for the game. Further, the *getProgress():float* method returns an integer between 0 and 1, giving the loading progress of the assets. When *getProgress()* is equal to 1, the assets are done loading. At this time, the user leaves the loading screen and enters the main menu. Subsequently, the *storeLoadedAssets()* method is called. This allows all the assets that have been loaded inside *updateLoading()* to be stored inside the member variables of the *Assets* class.

Finally *disposeInitialAssets()* simply frees the resources allocated to the images used inside the splash screen and the loading screen. The *dispose()* method is called when the user exits the game, and frees all memory allocated to the game's visual assets.

```
public class Assets
extends java.lang.Object
```

## Constructor Detail

### Assets

```
public Assets()
```

Creates an Assets instance responsible for the loading and visual and audio resources needed for the application.

## Method Detail

### loadInitialAssets

```
public void loadInitialAssets()
```

Loads the assets used by the company splash screen and the loading screen. We need to load their assets first before going to the loading screen and loading the all of the assets.

### loadSplashScreenAssets

```
public void loadSplashScreenAssets()
```

Loads the assets which are needed for the loading screen. They are loaded when the splash screen is shown.

### queueAssetsForLoading

```
private void queueAssetsForLoading()
```

Queues all assets for loading. Loading is performed every time the updateLoading() method is called. Before calling updateLoading(), the AssetManager must know which assets to load. This method puts all assets to queue inside the AssetManager instance.

### queueGeneralAssets

```
private void queueGeneralAssets()
```

Queues general assets for loading. Loading is performed every time the updateLoading() method is called. Before calling updateLoading(), the AssetManager needs to know which assets to load. This method puts all the general assets which persist throughout the game's entire life-cycle to queue inside the AssetManager instance.

### queueMainMenuAssets

```
public void queueMainMenuAssets()
```

Queues the main menu's assets for loading. Loading is performed every time the updateLoading() method is called. Before calling updateLoading(), the AssetManager which assets to load. This method puts all main menu assets assets to queue inside the AssetManager instance.

### queueGameAssets

```
private void queueGameAssets()
```

Queues the game's assets for loading. Loading is performed every time the updateLoading() method is called. Before calling updateLoading(), the AssetManager which assets to load. This method puts all game assets assets to queue inside the AssetManager instance.

### updateLoading

```
public boolean updateLoading()
```

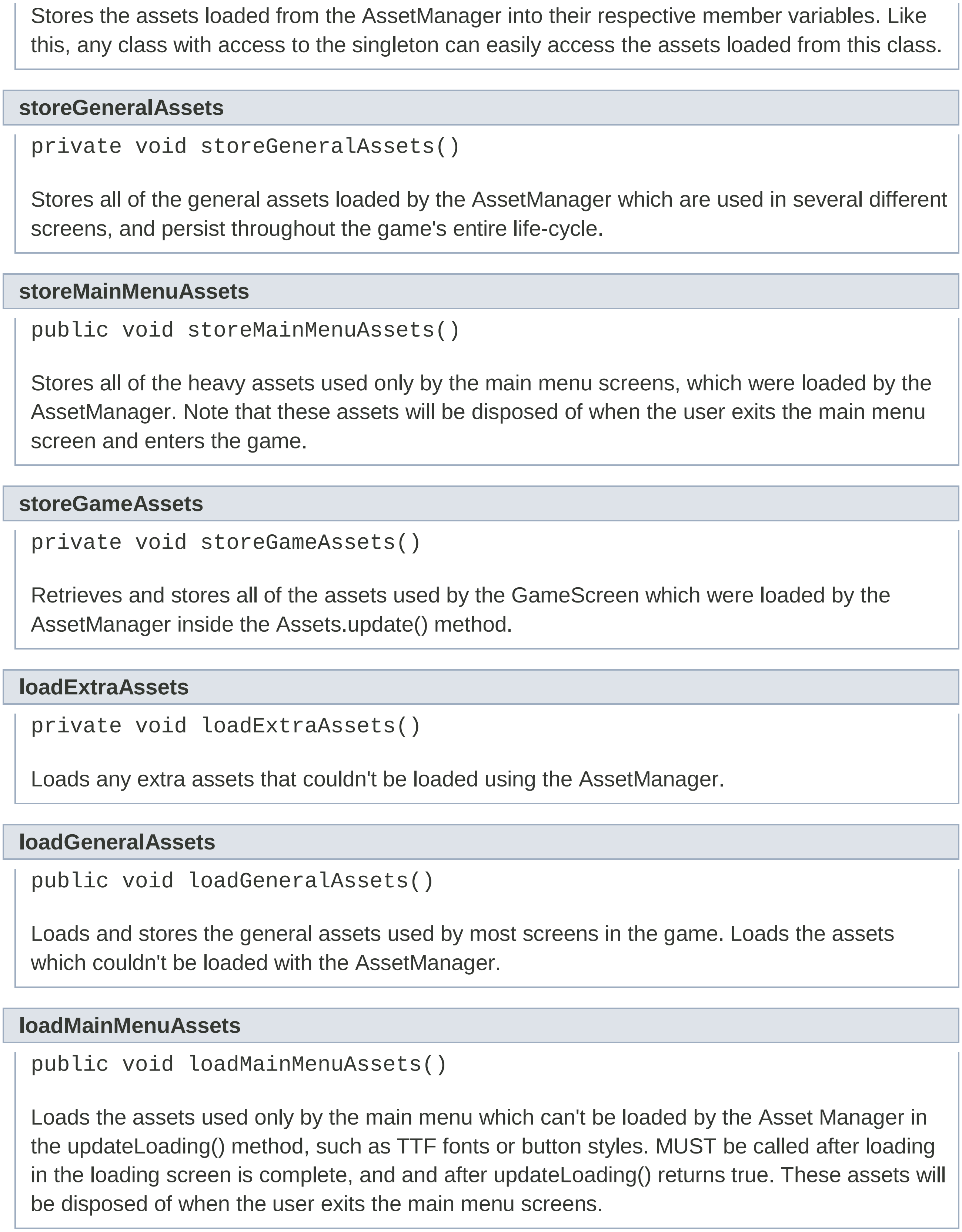
Call every frame inside a render() method to load the assets stored inside the AssetManager instance. This performs loading on a separate thread so that the render() method which calls this method can also update its appearance.

**Returns:**

Returns true if the loading is complete.

### storeLoadedAssets

```
public void storeLoadedAssets()
```



Stores the assets loaded from the AssetManager into their respective member variables. Like this, any class with access to the singleton can easily access the assets loaded from this class.

### storeGeneralAssets

```
private void storeGeneralAssets()
```

Stores all of the general assets loaded by the AssetManager which are used in several different screens, and persist throughout the game's entire life-cycle.

### storeMainMenuAssets

```
public void storeMainMenuAssets()
```

Stores all of the heavy assets used only by the main menu screens, which were loaded by the AssetManager. Note that these assets will be disposed of when the user exits the main menu screen and enters the game.

### storeGameAssets

```
private void storeGameAssets()
```

Retrieves and stores all of the assets used by the GameScreen which were loaded by the AssetManager inside the Assets.update() method.

### loadExtraAssets

```
private void loadExtraAssets()
```

Loads any extra assets that couldn't be loaded using the AssetManager.

### loadGeneralAssets

```
public void loadGeneralAssets()
```

Loads and stores the general assets used by most screens in the game. Loads the assets which couldn't be loaded with the AssetManager.

### loadMainMenuAssets

```
public void loadMainMenuAssets()
```

Loads the assets used only by the main menu which can't be loaded by the Asset Manager in the updateLoading() method, such as TTF fonts or button styles. MUST be called after loading in the loading screen is complete, and and after updateLoading() returns true. These assets will be disposed of when the user exits the main menu screens.

### loadGameAssets

```
public void loadGameAssets()
```

Loads the assets used in-game which couldn't be loaded by the Asset Manager in the updateLoading() method, such as SkeletonJson files. MUST be called after loading in the loading screen is complete, and and after updateLoading() returns true. Otherwise, there will be certain atlases and assets that won't be loaded that will cause NullPointerExceptions.getProgress

```
public float getProgress()
```

Returns the loading progress for the assets. Note that this method will return zero if the updateLoading() method has not been called yet.

**Returns:**

A value between 0.0f and 1.0f indicating the progress of the loading.

### disposeInitialAssets

```
public void disposeInitialAssets()
```

Disposes of any assets used by the loading screen or the company splash screen to free up system resources.

### disposeMainMenuAssets

```
public void disposeMainMenuAssets()
```

Dispose all of the heavy assets needed only by the main menu and all its associated screens. Called when the user leaves the main menu in order to free up resources for the GameScreen.

### dispose

```
public void dispose()
```

Called on application quit inside the Survivor class. Frees any audio/visual resources used by the application.

### II.3.1.37 *CraftingManager*

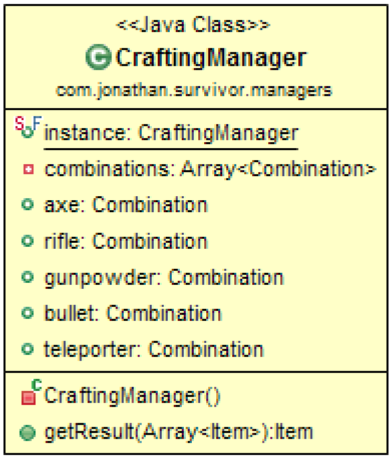


**Figure 56: CraftingManager class diagram**

The *CraftingManager* essentially dictates whether or not a given list of items forms a combination. This singleton is used by the *CraftingHud* whenever the user places items on the *CraftingTable*. When this happens, the list of items is passed to the *getResult(Array<Item>):Item* method. If the items produce a result, the result is returned.

```
public class CraftingManager
extends java.lang.Object
```

## Field Detail

### instance

```
public static final CraftingManager instance
```

Holds the singleton instance to the CraftingManager.

### combinations

```
private com.badlogic.gdx.utils.Array<CraftingManager.Combination>
combinations
```

Stores an array of all possible crafting combinations.

### axe

```
public CraftingManager.Combination axe
```

Possible crafting combination.

### rifle

```
public CraftingManager.Combination rifle
```

Possible crafting combination.

### gunpowder

```
public CraftingManager.Combination gunpowder
```

Possible crafting combination.

### bullet

```
public CraftingManager.Combination bullet
```

Possible crafting combination.

### teleporter

```
public CraftingManager.Combination teleporter
```

Possible crafting combination.

## Constructor Detail

### CraftingManager

```
private CraftingManager()
```

Instantiates a new crafting manager.

## Method Detail

### getResult

```
public CraftingManager.Item getResult(
              com.badlogic.gdx.utils.Array<CraftingManager.Item> items)
```

Returns the resulting item crafted using the given array of items. If null, no result is formed using the given list of items.

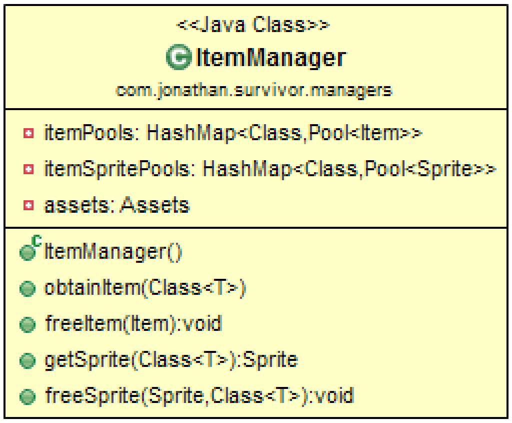
**Parameters:**

`items` - a list of items. If the given combination can craft an item, that item is returned

**Returns:**

returns the item that can be crafted by the given list of items. Null, if the combination cannot craft anything.

### II.3.1.37b *ItemManager* and *ZombieManager*



<<Java Class>>
ItemManager
com.jonathan.survivor.managers
itemPools: HashMap<Class,Pool<Item>>
itemSpritePools: HashMap<Class,Pool<Sprite>>
assets: Assets
ItemManager()
obtainItem(Class<T>)
freeItem(Item):void
getSprite(Class<T>):Sprite
freeSprite(Sprite,Class<T>):void

**Figure 57: ItemManager class diagram**

The *ItemManager* class is responsible for pooling *Item* instances and sprites. In fact, this is a crucial optimization for the game. Instead of creating *Item* instances whenever the user collects an item, that item is simply refered to by its class (e.g., *Wood.class*). When the *CraftingTable* or the *InventoryList* needs to access the data fields for an item, *ItemManager.obtainItem()* is called, and an *Item* instance is returned. The same thing happens with regards to item sprites. This prevents the application from constantly creating and destroying *Item* instances and sprites, and is thus an important optimization.

```
public class ItemManager
extends java.lang.Object
```

#### Field Detail

#### itemPools

```
private java.util.HashMap<java.lang.Class,com.badlogic.gdx.utils.Pool<
Item>> itemPools
```

Holds a HashMap where every item subclass is a key, and its value is a pool of items of that class.

#### itemSpritePools

```
private java.util.HashMap<java.lang.Class,com.badlogic.gdx.utils.Pool<
com.badlogic.gdx.graphics.g2d.Sprite>> itemSpritePools
```

Holds a HashMap where every item subclass is a key, and its value is a pool of inventory sprites for that item.

### assets

```
private Assets assets
```

Stores the universal assets singleton used to retrieve Sprite templates for each item.

## Constructor Detail

### ItemManager

```
public ItemManager()
```

Creates an itemManager which provides access to Item instances. Efficient, as it uses pools.

## Method Detail

### obtainItem

```
public <T extends Item> T obtainItem(java.lang.Class<T> itemClass)
```

Obtains an Item instance of the given class.

**Type Parameters:**

`T` - the generic type of the Item class

**Parameters:**

`itemClass` - the item class whose object will be returned

**Returns:**

an instance of the given Item class.

### freeItem

```
public void freeItem(Item item)
```

Frees an Item instance back into the manager's internal pools for later reuse.

**Parameters:**

`item` - the item to free back into a pool

### getSprite

```
public <T extends
Item> com.badlogic.gdx.graphics.g2d.Sprite getSprite(java.lang.Class<T
> itemClass)
```

Obtains an Sprite instance of the given class to display in an inventory.

**Type Parameters:**

`T` - the generic type of the item

**Parameters:**

`itemClass` - the item class to obtain a sprite for

**Returns:**

a sprite instance for the given item type.

### freeSprite

```
public <T extends Item> void freeSprite
                     (com.badlogic.gdx.graphics.g2d.Sprite sprite,
                                          java.lang.Class<T> itemClass)
```

Frees the specified sprite instance back inside an internal pool for later reuse.

**Type Parameters:**

`T` - the generic type of the item

**Parameters:**

`sprite` - the sprite to free back into a pool

`itemClass` - the item class displayed by the sprite

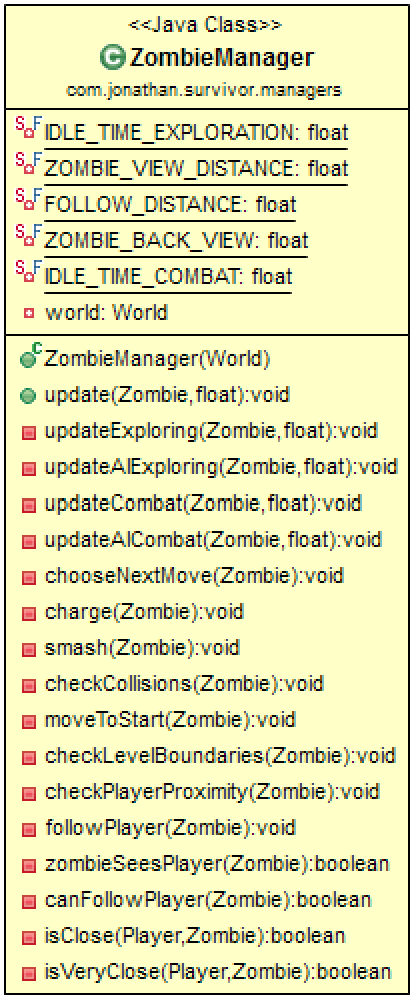

<<Java Class>>
ZombieManager
com.jonathan.survivor.managers
IDLE_TIME_EXPLORATION: float
ZOMBIE_VIEW_DISTANCE: float
FOLLOW_DISTANCE: float
ZOMBIE_BACK_VIEW: float
IDLE_TIME_COMBAT: float
world: World
ZombieManager(World)
update(Zombie,float):void
updateExploring(Zombie,float):void
updateAIExploring(Zombie,float):void
updateCombat(Zombie,float):void
updateAICombat(Zombie,float):void
chooseNextMove(Zombie):void
charge(Zombie):void
smash(Zombie):void
checkCollisions(Zombie):void
moveToStart(Zombie):void
checkLevelBoundaries(Zombie):void
checkPlayerProximity(Zombie):void
followPlayer(Zombie):void
zombieSeesPlayer(Zombie):boolean
canFollowPlayer(Zombie):boolean
isClose(Player,Zombie):boolean
isVeryClose(Player,Zombie):boolean

**Figure 58: ZombieManager class diagram**

The *ZombieManager* is a helper class to the *World* class. Instead of having the *World* class update every zombie's game logic, the *Zombie* instance is passed to the *ZombieManager*, and the zombie is updated there. This avoids cluttering the *World* class with the zombies' functionalities.

```
public class ZombieManager
extends java.lang.Object
```

## Field Detail

### IDLE_TIME_EXPLORATION

```
private static final float IDLE_TIME_EXPLORATION
```

Stores the amount of time the zombie stays idle before starting to move again (in EXPLORATION mode).

### ZOMBIE_VIEW_DISTANCE

```
private static final float ZOMBIE_VIEW_DISTANCE
```

Holds the x-distance between the zombie and the player so that the zombie can first see the player.

### FOLLOW_DISTANCE

```
private static final float FOLLOW_DISTANCE
```

Stores the minimum x-distance between the zombie and the player so that the zombie will continue to follow him.

### ZOMBIE_BACK_VIEW

```
private static final float ZOMBIE_BACK_VIEW
```

Stores the x-distance between the zombie and the player so that the zombie can see the player even when he's in back of the zombie.

### IDLE_TIME_COMBAT

```
private static final float IDLE_TIME_COMBAT
```

Stores the amount of time the zombie stays idle before choosing a move to attack the player (in COMBAT mode).

### world

```
private World world
```

Holds the World instance that the zombies are a part of. Used to access the convenience methods created for the player and needed for the zombies.

## Constructor Detail

### ZombieManager

```
public ZombieManager(World world)
```

Creates a ZombieManager which updates zombies and their AI. Accepts the World instance used by the game in order to access the convenience methods used by the player and similarly needed to update zombies.

**Parameters:**

`world` - the world, whose methods are used to control zombies and get access to the player's information

## Method Detail

### update

```
public void update(Zombie zombie, float deltaTime)
```

Updates the game logic for the given zombie.

**Parameters:**

`zombie` - the zombie to update

`deltaTime` - the execution time of the previous frame.

### updateExploring

```
private void updateExploring(Zombie zombie, float deltaTime)
```

Updates the game logic of the given zombie when he is in EXPLORING state.

**Parameters:**

`zombie` - the zombie to update

`deltaTime` - the execution time of the previous frame.

### updateAIExploring

```
private void updateAIExploring(Zombie zombie, float deltaTime)
```

Updates the zombie's state according to the principles of his artifical intelligence. Called when the zombie is in Exploration mode.

**Parameters:**

`zombie` - the zombie to update

`deltaTime` - the execution time of the previous frame.

### updateCombat

```
private void updateCombat(Zombie zombie, float deltaTime)
```

Updates the zombie when he is in COMBAT mode and fighting the player.

**Parameters:**

`zombie` - the zombie to update

`deltaTime` - the execution time of the previous frame.

### updateAICombat

```
private void updateAICombat(Zombie zombie, float deltaTime)
```

Updates the zombie's state according to the principles of his artifical intelligence. Called when the zombie is in COMBAT mode.

**Parameters:**

`zombie` - the zombie to update

`deltaTime` - the execution time of the previous frame.

### chooseNextMove

```
private void chooseNextMove(Zombie zombie)
```

Makes the given zombie choose the next move to attack the player with.

**Parameters:**

`zombie` - the zombie whose next move is chosen

### charge

```
private void charge(Zombie zombie)
```

Make the zombie charge towards the player.

**Parameters:**

`zombie` - the zombie that will charge

### smash

```
private void smash(Zombie zombie)
```

Makes the zombie perform a smash.

**Parameters:**

`zombie` - the zombie that will perform a smash and ultimately an earthquake.

### checkCollisions

```
private void checkCollisions(Zombie zombie)
```

Checks if the zombie has intersected with any GameObject which is pertinent to the zombie, such as the player.

**Parameters:**

`zombie` - the zombie which is tested for collisions with other GameObjects.

### moveToStart

```
private void moveToStart(Zombie zombie)
```

Moves the zombie back to his starting position.

**Parameters:**

`zombie` - the zombie to move to his starting position

### checkLevelBoundaries

```
private void checkLevelBoundaries(Zombie zombie)
```

Checks if the zombie is within the confines of the combat level. If not, the zombie is respawned at the right place.

**Parameters:**

`zombie` - the zombie to check level boundaries with

### checkPlayerProximity

```
private void checkPlayerProximity(Zombie zombie)
```

Sets the zombie to ALERT state if the zombie is not already alerted.

**Parameters:**

`zombie` - the zombie to check player proximity with

### followPlayer

```
private void followPlayer(Zombie zombie)
```

Makes the zombie walk towards the player to essentially follow him.

**Parameters:**

`zombie` - the zombie which is meant to follow the player.

### zombieSeesPlayer

```
private boolean zombieSeesPlayer(Zombie zombie)
```

Returns true if the zombie sees the player in the world.

**Parameters:**

`zombie` - the zombie who is tested to see the player

**Returns:**

true, if the given zombie can see the world's player.

### canFollowPlayer

```
private boolean canFollowPlayer(Zombie zombie)
```

Returns true if the given zombie is already following the player, and can continue to follow him. The player has to be within a given distance from the zombie.

**Parameters:**

`zombie` - the zombie which is tested to be able to follow the player.

**Returns:**

true, if the zombie can follow the player

### isClose

```
private boolean isClose(Player player, Zombie zombie)
```

Returns true if the player is close to the given zombie.

**Parameters:**

`player` - the player which is tested for proximity with the given zombie

`zombie` - the zombie which is tested for proximity with the given player.

**Returns:**

true, if the player and the zombie are close in distance

### isVeryClose

```
private boolean isVeryClose(Player player, Zombie zombie)
```

Returns true if the player is very close to the zombie, and the zombie can even see the player from in back of him.

**Parameters:**

`player` - the player which is tested for proximity with the given zombie

`zombie` - the zombie which is tested for proximity with the given player.

**Returns:**

true, if the player and the zombie are very close in distance

II.3.1.38 *GestureManager*

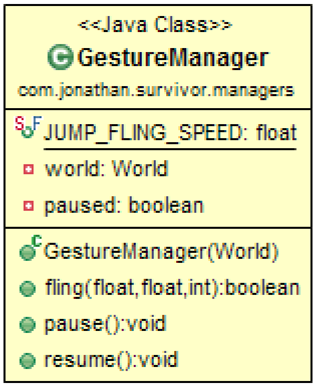


**Figure 59 : GestureManager class diagram**

The role of the *GestureManager* is to detect touch gestures from the user's finger, and to update the world's logic accordingly.

The first constant in the *GestureManager* class is the *JUMP_FLING_SPEED* floating-point. It dictates the velocity in the y-axis in pixels per second that a swipe must have in order to make the player jump to a higher layer or fall to a lower layer. Also, the *GestureManager* holds a reference to the *World*. This is because the manager needs a reference to the player in the world. In fact, when the manager receives a swipe event, it will call the player's *jump()* method or *fall()* method depending on whether the swipe went upwards or downwards. Furthermore, the *GestureManager* has the *paused* boolean. If *true*, the *GestureManager* does not receive any gesture events.

The constructor of the class accepts the *World* instance used to control game logic. In terms of behaviour, the *fling(float, float, int):boolean* method is called when the user performs a swipe on the screen. The first two parameters indicate the x and y velocity of the swipe in pixels per second. The method returns *true*, telling LibGDX that the swipe gesture has been handled. Thus, the *fling()* method will not be called again on the same frame. Next, the *pause()* and *resume()* methods change the *paused* field accordingly, and either pause or resume gestures handling in the class.

```
public class GestureManager
extends com.badlogic.gdx.input.GestureDetector.GestureAdapter
```

### Field Detail

#### JUMP_FLING_SPEED

```
public static final float JUMP_FLING_SPEED
```

Stores the minimum speed the user has to fling to make the player jump or fall.

#### world

```
private World world
```

Stores the world that the manager will modify according to user input.

#### paused

```
private boolean paused
```

True if the game is paused, so that no gestures should be registered nor handled.

### Constructor Detail

#### GestureManager

```
public GestureManager(World world)
```

Creates an InputManager with the given world. This manager receives all touch events and reacts by calling the appropriate methods for the World.

**Parameters:**

`world` - the world whose methods are called when gestures should control player movement.

## Method Detail

### fling

```
public boolean fling(float velocityX,float velocityY, int button)
```

Called when the user flicks the screen. Velocity is in pixels/second. This method is also called on Desktop.

**Specified by:**

`fling` in interface `com.badlogic.gdx.input.GestureDetector.GestureListener`

**Overrides:**

`fling` in class `com.badlogic.gdx.input.GestureDetector.GestureAdapter`

**Parameters:**

`velocityX` - the x velocity of the swipe

`velocityY` - the y velocity of the swipe

`button` - the button which was pressed while flinging (unimportant for Android)

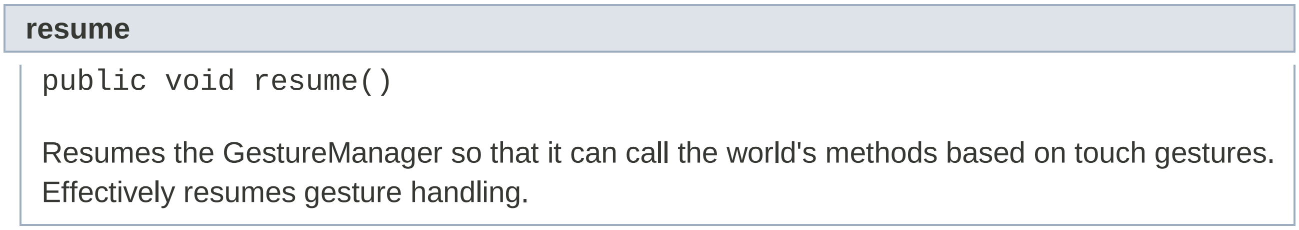
**Returns:**

true if the fling event can be sent to other GestureAdapters

### pause

```
public void pause()
```

Pauses the GestureManager so that it doesn't call any of the world's methods. Effectively pauses gesture handling.



**resume**

```
public void resume()
```

Resumes the GestureManager so that it can call the world's methods based on touch gestures. Effectively resumes gesture handling.

II.3.1.38 *InputManager*

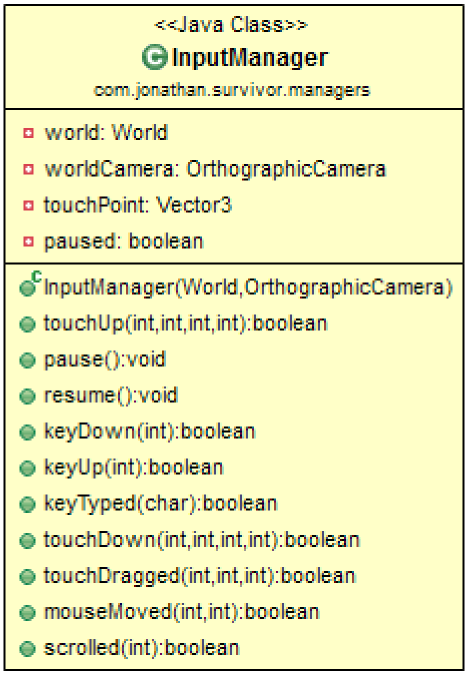

<<Java Class>>
InputManager
com.jonathan.survivor.managers
world: World
worldCamera: OrthographicCamera
touchPoint: Vector3
paused: boolean
InputManager(World,OrthographicCamera)
touchUp(int,int,int,int):boolean
pause():void
resume():void
keyDown(int):boolean
keyUp(int):boolean
keyTyped(char):boolean
touchDown(int,int,int,int):boolean
touchDragged(int,int,int):boolean
mouseMoved(int,int):boolean
scrolled(int):boolean

**Figure 60 : InputManager class diagram**

The *InputManager* is similar to the *GestureManager*. The main difference, however, is that the *InputManager* detects taps instead of gestures.

First, the *InputManager* holds a reference to the *World*. This allows the manager to call the *World.touchUp(x:float, y:float)* method. However, before calling this method, the *InputManager* must first convert touch coordinates into world coordinates. To do this, the manager must have an instance of the *worldCamera*, which is the camera used to render the world and its surrounding *GameObjects*. Much like the *GestureManager,* this class has a *paused* flag. If true, the manager stops processing input.

The constructor of the class accepts the *World* instance used to control game logic. Also, it accepts the *worldCamera* used to render *GameObjects*. This allows the manager's member variables to be populated with the constructor's parameters.

Next, *touchUp(x:int, y:int, pointer:int, button:int):boolean* method is called by LibGDX when the user touches the screen. The touch point is given in screen coordinates, which is why it must be converted into world coordinates using the *worldCamera.unproject()* method. Moreover, the *pointer* and *button* parameters are irrelevant, as these parameters are only useful if the target platform is Windows or Mac. Finally, the *pause()* and *resume()* methods change the *paused* field accordingly, and either pause or resume gestures handling in the class.

```
public class InputManager
extends java.lang.Object
implements com.badlogic.gdx.InputProcessor
```

### Field Detail

#### world

```
private World world
```

Stores the world that the manager will modify according to user input.

#### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the worldCamera, used to convert touch points to world coordinates.

#### inputListener

```
private InputManager.InputListener inputListener
```

The InputListener which delegates events to the GameScreen. Allows the GameScreen to know about certain input events.

#### touchPoint

```
private com.badlogic.gdx.math.Vector3 touchPoint
```

Helper Vector3 used to store the latest touch point.

#### paused

```
private boolean paused
```

Holds true if the game is paused, and no input events should be delegated to the world.

## Constructor Detail

### InputManager

```
public InputManager(World world,
            com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Creates an InputManager with the given world. This manager receives all touch events and reacts by calling the appropriate methods for the World.

**Parameters:**

`world` - the world, whose methods are called when the user taps the screen

`worldCamera` - the world camera, used to convert screen coordinates to world coordinates.

## Method Detail

### touchUp

```
public boolean touchUp(int screenX, int screenY, int pointer,
              int button)
```

Called when the user releases a press anywhere on the screen.

**Specified by:**

`touchUp` in interface `com.badlogic.gdx.InputProcessor`

**Parameters:**

`screenX` - the screen x-position of the tap

`screenY` - the screen y-position of the tap

`pointer` - the pointer of the tap

`button` - the button of the tap (irrelevant for Android devices)

**Returns:**

true, if the event should be delegated to other touchUp() methods.

### pause

```
public void pause()
```

Pauses the InputManager so that it doesn't call any of the world's methods. Effectively pauses input.

### resume

```
public void resume()
```

Resumes the InputManager so that it can call the world's methods based on touch events. Effectively resumes input handling.

### keyDown

```
public boolean keyDown(int keycode)
```

Called when a key has been pressed.

**Specified by:**

`keyDown` in interface `com.badlogic.gdx.InputProcessor`

**Parameters:**

`keycode` - the keycode of the pressed key

**Returns:**

true, if the keyDown() method of other InputProcessors should be delegated

### setInputListener

```
public void setInputListener(InputManager.InputListener listener)
```

Registers the given InputListener to this InputManager instance. The InputManager will then call method from the given listener.

### getInputListener

```
public InputManager.InputListener getInputListener()
```

Returns the InputListener instance whose methods are delegated by this InputManager instance.

### keyUp

```
public boolean keyUp(int keycode)
```

**Specified by:**

keyUp in interface com.badlogic.gdx.InputProcessor

**See Also:**

InputProcessor.keyUp(int)

### keyTyped

```
public boolean keyTyped(char character)
```

**Specified by:**

keyTyped in interface com.badlogic.gdx.InputProcessor

**See Also:**

InputProcessor.keyTyped(char)

### touchDown

```
public boolean touchDown(int screenX, int screenY, int pointer,
                            int button)
```

**Specified by:**

touchDown in interface com.badlogic.gdx.InputProcessor

**See Also:**

InputProcessor.touchDown(int, int, int, int)

### touchDragged

```
public boolean touchDragged(int screenX, int screenY,int pointer)
```

**Specified by:**

touchDragged in interface com.badlogic.gdx.InputProcessor

**See Also:**

InputProcessor.touchDragged(int, int, int)

### mouseMoved

```
public boolean mouseMoved(int screenX,
                          int screenY)
```

**Specified by:**

`mouseMoved` in interface `com.badlogic.gdx.InputProcessor`

**See Also:**

`InputProcessor.mouseMoved(int, int)`

### scrolled

```
public boolean scrolled(int amount)
```

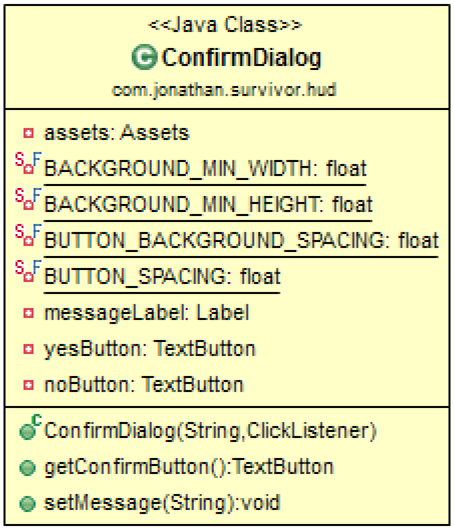
**Specified by:**

`scrolled` in interface `com.badlogic.gdx.InputProcessor`

**See Also:**

`InputProcessor.scrolled(int)`

### II.3.1.39 *ConfirmDialog*



<<Java Class>>
ConfirmDialog
com.jonathan.survivor.hud
assets: Assets
BACKGROUND_MIN_WIDTH: float
BACKGROUND_MIN_HEIGHT: float
BUTTON_BACKGROUND_SPACING: float
BUTTON_SPACING: float
messageLabel: Label
yesButton: TextButton
noButton: TextButton
ConfirmDialog(String,ClickListener)
getConfirmButton():TextButton
setMessage(String):void

**Figure 61 : ConfirmDialog class diagram**

The *ConfirmDialog* is a dialog which prompts the user with a message, which is usually a question, followed by a *Yes* and a *No* button. It allows the program to prompt the user to confirm a choice.

```
public class ConfirmDialog
extends com.badlogic.gdx.scenes.scene2d.ui.Dialog
```

## Field Detail

### assets

```
private Assets assets
```

Holds the Assets singleton used to retrieve pre-defined ui styles.

### BACKGROUND_MIN_WIDTH

```
private static final float BACKGROUND_MIN_WIDTH
```

Holds the width of the background. Background is scaled up if it is too small for the dialog's widgets.

### BACKGROUND_MIN_HEIGHT

```
private static final float BACKGROUND_MIN_HEIGHT
```

Holds the height of the background. Background is scaled up if it is too small for the dialog's widgets.

### BUTTON_BACKGROUND_SPACING

```
private static final float BUTTON_BACKGROUND_SPACING
```

Stores the spacing between the bottom of the buttons and the bottom of the nine-patch background.

### BUTTON_SPACING

```
private static final float BUTTON_SPACING
```

Stores the horizontal spacing between the "Yes" and "No" buttons.

### messageLabel

```
private com.badlogic.gdx.scenes.scene2d.ui.Label messageLabel
```

Holds the label displaying the confirm dialog's message.

### yesButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton yesButton
```

Stores the 'Yes' and the 'No' buttons which prompt the user for confirmation.

### noButton

```
private com.badlogic.gdx.scenes.scene2d.ui.TextButton noButton
```

Stores the 'Yes' and the 'No' buttons which prompt the user for confirmation.

## Constructor Detail

### ConfirmDialog

```
public ConfirmDialog(java.lang.String message,
   com.badlogic.gdx.scenes.scene2d.utils.ClickListener clickListener)
```

Creates the confirm dialog with the given confirmation message as a title. The clickListener is registered to the 'Yes' button. Thus, its clicked() method will be called when the yesButton is pressed.

**Parameters:**

`message` - the message shown on the dialog

`clickListener` - the listener which receives an event whenever the "Yes" button is pressed on the dialog.

## Method Detail

### getConfirmButton

```
public
com.badlogic.gdx.scenes.scene2d.ui.TextButton getConfirmButton()
```

Returns the "Yes" button clicked when the user confirms his choice.

### setMessage

```
public void setMessage(java.lang.String message)
```

Sets the message shown on the confirm dialog.

**Parameters:**

`message` - the new message to display

### II.3.1.40 *SpriteUtils*

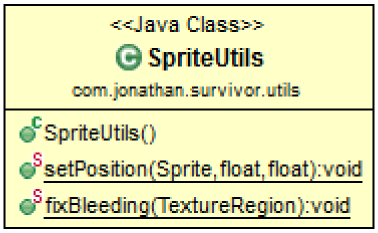

<<Java Class>>
SpriteUtils
com.jonathan.survivor.utils
SpriteUtils()
setPosition(Sprite,float,float):void
fixBleeding(TextureRegion):void

**Figure 62 : SpriteUtils class diagram**

The *SpriteUtils* class is a short utility class meant to perform basic operations with textures and sprites.

```
public class SpriteUtils
extends java.lang.Object
```

**Constructor Detail**

**SpriteUtils**

```
public SpriteUtils()
```

**Method Detail**

**setPosition**

```
public static void setPosition(
com.badlogic.gdx.graphics.g2d.Sprite sprite, float x, float y)
```

Positions the center of the sprite at the given (x,y) coordinates in world coordinates.

**Parameters:**

`sprite` - the sprite whose position to alter

x - the new center x-position of the sprite

y - the new center y-position of the sprite

### fixBleeding

```
public
static void fixBleeding(com.badlogic.gdx.graphics.g2d.TextureRegion re
gion)
```

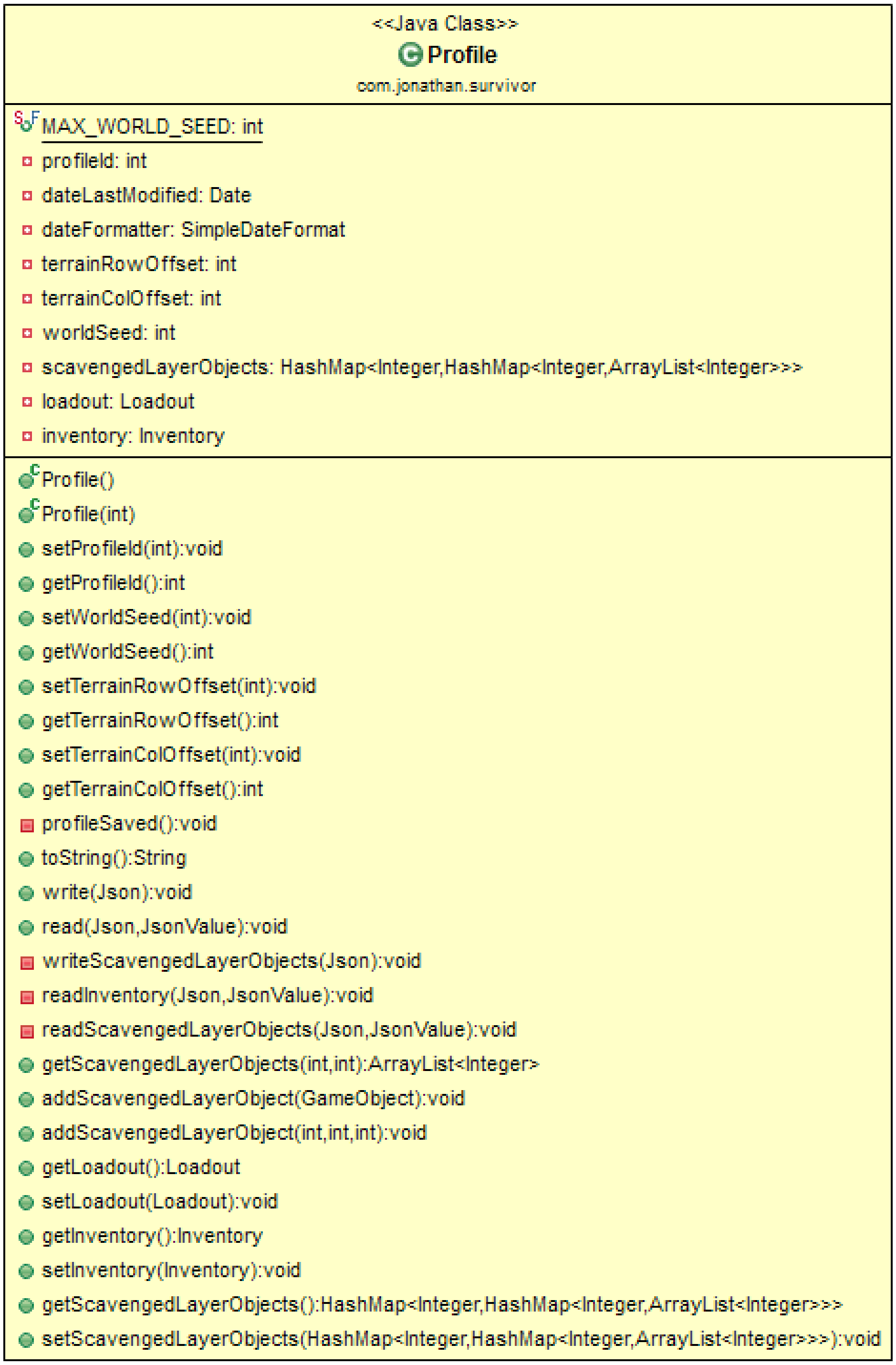
Fixes bleeding on a TextureRegion, removing the edge pixels that can bleed into the edges of the texture.

**Parameters:**

`region` - the region whose bleeding will be fixed

(See next page for *Profile* class)

## II.3.1.41 *Profile*



```
<<Java Class>>
Profile
com.jonathan.survivor

MAX_WORLD_SEED: int
profileId: int
dateLastModified: Date
dateFormatter: SimpleDateFormat
terrainRowOffset: int
terrainColOffset: int
worldSeed: int
scavengedLayerObjects: HashMap<Integer,HashMap<Integer,ArrayList<Integer>>>
loadout: Loadout
inventory: Inventory

Profile()
Profile(int)
setProfileId(int):void
getProfileId():int
setWorldSeed(int):void
getWorldSeed():int
setTerrainRowOffset(int):void
getTerrainRowOffset():int
setTerrainColOffset(int):void
getTerrainColOffset():int
profileSaved():void
toString():String
write(Json):void
read(Json,JsonValue):void
writeScavengedLayerObjects(Json):void
readInventory(Json,JsonValue):void
readScavengedLayerObjects(Json,JsonValue):void
getScavengedLayerObjects(int,int):ArrayList<Integer>
addScavengedLayerObject(GameObject):void
addScavengedLayerObject(int,int,int):void
getLoadout():Loadout
setLoadout(Loadout):void
getInventory():Inventory
setInventory(Inventory):void
getScavengedLayerObjects():HashMap<Integer,HashMap<Integer,ArrayList<Integer>>>
setScavengedLayerObjects(HashMap<Integer,HashMap<Integer,ArrayList<Integer>>>):void
```

**Figure 63 : Profile class diagram**

A *Profile* is used to hold the user's save data. It allows the player to load an old save file and continue in the game where he left off.

The first variable in the class is the *profileId* integer, a number used to identify each profile. These profiles are numbered from zero and on, sequentially. The id corresponds to the order of the profiles in the *World Select Menu*. That is, the profile with *id* equal to zero will appear first on the profile list. Next, the *dateLastModified* variable simply holds the date where the profile was last saved. It allows each profile to display a time stamp in the *World Select Menu*.

Next is the more interesting *terrainRowOffset* and *terrainColOffset*. These integers hold the cell coordinates of the bottom-left layer in the *terrainLayers* matrix inside the *TerrainLevel* class. Its purpose is to retain where the player was last located in the world. In fact, when a profile is loaded, the *TerrainLevel* will know that the *TerrainLayer* in *terrainLayers[0][0]* should have row *terrainRowOffset* and column *terrainColOffset.* As such, the world will be generated with the right data such that the player will be dropped at the same place he left off in when last quitting the game.

On a different note, the *worldSeed* integer holds the seed used to procedurally generate the world. Since each *Profile* has a *worldSeed*, the world will always look the same, since all terrain geometry and objects are generated using this random number generator seed.

One of the most complex data fields of the class is the *scavengedLayerObjects* HashMap. This HashMap holds an integer key, to which another HashMap is assigned as a value. This inner HashMap has an integer key which holds an *ArrayList* of Integers. To start, the first, outer integer key denotes the row of a *TerrainLayer*. Thus, *scavengedLayerObjects.get(0)* returns a HashMap for each *TerrainLayer* in row zero. The inner integer key denotes a layer's column. Thus, accessing *scavengedLayerObjects.get(0).get(0)* returns an *ArrayList* of integers for row zero and column zero. This *ArrayList* contains a list of *objectIds.* These are the *ids* for the *GameObjects* which have been scavenged by the player. For instance, if the player chops down a tree in row zero, column zero, and the tree's id was one, then *scavengedLayerObjects.get(0).get(0)* will return an *ArrayList* with the integer one inside. Afterwards, whenever the *TerrainLayer* with row zero and column zero has to be populated with

*GameObjects*, the following happens. Say that the *TerrainLayer* randomly generates a number, and from that number dictates that a tree should be placed on the layer. If that tree is the second object to be placed on the layer, its *objectId* will be one. Thus, since one is part of the *ArrayList* of scavenged objects, the tree will not be placed on the layer. In this case, the layer knows that the player has chopped down the tree. Thus, the layer will skip to placing the third object in the level. This is how trees and zombie are never spawned again once they are killed or scavenged.

Finally, the *Profile* holds a reference to the *Loadout* and the *Inventory* which belongs to the player. This allows the player's items and weapons to be saved to the hard drive and retrieved once the profile is loaded.

Next, the *Profile* constructor simply creates a profile with default values for its member variables. Only once the *read(Json, JsonValue):void* is called do the data fields get populated with hard drive information. In fact, the purpose of this *read(Json, JsonValue):void* method is to read a JSON file and to populate the member variables with its information. Conversely, the *write(Json):void* method is called when the profile needs to be saved to the hard drive. In this case, every data field of the class is parsed into a string and written to a JSON file.

On a separate note, the *getScavengedObjects(int, int):ArrayList<Integer>* accepts a row and a column as parameters. The method will then return an *ArrayList* containing the objectIds of the objects that have been scavenged in the layer denoted by the given row and the column.

Next, the *addScavengedLayerObject(GameObject)* method accepts a *GameObject* which was scavenged by the player. The way the method works is the following

```
//Pseudocode start

scavengedLayerObjects.get(go.terrainCell.row).get(go.terrainCell.col).add(go.objectId);

//Pseudocode end
```

As such, the *GameObject* will be registered inside the *scavengedLayerObjects* HashMap and thus will never respawn.

Similarly, the *addScavengedLayerObject(row, col, objectId)* performs the same functionality, except that it accepts the row and the column of the *TerrainLayer* to which to add the objectId. It further accepts the *objectId* which to add to the ArrayList in the HashMap.

```
public class Profile
extends java.lang.Object
implements com.badlogic.gdx.utils.Json.Serializable
```

### Field Detail

#### MAX_WORLD_SEED

```
public static final int MAX_WORLD_SEED
```

Stores the max world seed that will be used to create the world. Possibly the higher the value, the more probability in world diversity.

#### firstTimeCreate

```
private boolean firstTimeCreate
```

True if this profile was just created, and the player has not saved the game since creating the level.

#### profileId

```
private int profileId
```

Stores the id of the profile, where 0 is the first profile shown in the world selection list.

#### dateLastModified

```
private java.util.Date dateLastModified
```

Stores the date the profile was last modified.

#### dateFormatter

```
private transient java.text.SimpleDateFormat dateFormatter
```

Helper object used to convert the date last modified into a string.

#### terrainRowOffset

```
private int terrainRowOffset
```

Stores the row offset to use for the TerrainLayers of the level. These cell coordinates are the coordinates of the bottom-left- most layer of the TerrainLevel when the game was saved. Thus, if this offset is specified, the TerrainLevel can choose to define the bottom-left- most layer to have these cell coordinates, and the player will be dropped in the same cell he left off in the TerrainLevel.

### terrainColOffset

```
private int terrainColOffset
```

Stores the column offset to should use for the TerrainLayers of the level. These cell coordinates are the coordinates of the bottom-left- most layer of the TerrainLevel when the game was saved. Thus, if this offset is specified, the TerrainLevel can choose to define the bottom-left-most layer to have these cell coordinates, and the player will be dropped in the same cell he left off in the TerrainLevel.

### lastXPos

```
private float lastXPos
```

The player's last x-position when he saved the profile. Allows the player to spawn in exactly the same place. Note that this position is relative to the left-most x-position of the layer where the player resided on game save.

### worldSeed

```
private int worldSeed
```

Stores the world seed. Each profile has a different seed. The same seed creates the same world.

### scavengedLayerObjects

```
private java.util.HashMap<java.lang.Integer,java.util.HashMap<java.lan
g.Integer,java.util.ArrayList<java.lang.Integer>>>
scavengedLayerObjects
```

Stores a HashMap containing lists of scavenged objects in each TerrainLayer. First key is the layer's row, second is the layer's column. The array stores the list of objectIds for all GameObjects that have been scavenged on that layer.

### loadout

```
private Loadout loadout
```

Stores the player's loadout so that it stays constants when re-entering the game.

### inventory

```
private Inventory inventory
```

Holds the player's inventory, which contains all of the player's collected items.

## Constructor Detail

### Profile

```
public Profile()
```

Creates a default profile with profileId = 0. This constructor will be called when a Profile object is read from a JSON file.

### Profile

```
public Profile(int id)
```

Creates a new profile starting from the beginning of the game. Called when the user creates a new world.

**Parameters:**

`id` - ID of the profile we want to create. The first profile has ID 0, and is the first shown in the world selection list.

## Method Detail

### getDateLastModified

```
public java.util.Date getDateLastModified()
```

Returns the date at which the profile was last modified and saved to the hard drive. Note that the Date object's time is mutable.

### setProfileId

```
public void setProfileId(int profileId)
```

Sets the profile Id of the profile.

### getProfileId

```
public int getProfileId()
```

Returns the if of the profile, where 0 is the first profile shown in the world selection list

### setWorldSeed

```
public void setWorldSeed(int worldSeed)
```

Sets the world seed of the profile. Changing it changes the entire world. Should not be changed after profile creation, or will break save file.

### getWorldSeed

```
public int getWorldSeed()
```

Returns the world seed used to procedurally generate the world. Same seed = same world.

### setTerrainRowOffset

```
public void setTerrainRowOffset(int offset)
```

Sets the terrain row offset to be used the next time the game is loaded. Set this to the row of the bottom-left-most layer of the level to resume the game where the player left off last.

### getTerrainRowOffset

```
public int getTerrainRowOffset()
```

Gets the terrain row offset which was saved to profile. Specify this as the rowOffset of the TerrainLevel to start the game at the same place the user left off.

### setTerrainColOffset

```
public void setTerrainColOffset(int offset)
```

Sets the terrain column offset to be used the next time the game is loaded. Set this to the column of the bottom-left-most layer of the level to resume the game where the player left off last.

### getTerrainColOffset

```
public int getTerrainColOffset()
```

Gets the terrain column offset which was saved to profile. Specify this as the colOffset of the TerrainLevel to start the game at the same place the user left off.

### setLastXPos

```
public void setLastXPos(float x)
```

Updates the last x-position where the player was upon saving the profile. Note that this position is relative to the left-most x-position of the layer where the player resided on profile save.

### getLastXPos

```
public float getLastXPos()
```

Returns the last x-position where the player was upon saving the profile. Note that this position is relative to the left-most x-position of the layer where the player resided on profile save.

### profileSaved

```
private void profileSaved()
```

Called when the profile has been saved to the hard drive.

### toString

```
public java.lang.String toString()
```

Returns a string representation for the profile, used for each item of the world selection list from the world select screen.

**Overrides:**

`toString` in class `java.lang.Object`

### write

```
public void write(com.badlogic.gdx.utils.Json json)
```

Indicates how a Profile object is converted to a JSON file.

**Specified by:**

`write` in interface `com.badlogic.gdx.utils.Json.Serializable`

**See Also:**

`Json.Serializable.write(com.badlogic.gdx.utils.Json)`

### read

```
public void read(com.badlogic.gdx.utils.Json json,
                 com.badlogic.gdx.utils.JsonValue jsonData)
```

Indicates how a JSON file is read to be converted into a Profile object. Note that the default Profile constructor is called before this method.

**Specified by:**

`read` in interface `com.badlogic.gdx.utils.Json.Serializable`

**See Also:**

`Json.Serializable.read(com.badlogic.gdx.utils.Json, com.badlogic.gdx.utils.JsonValue)`

### writeScavengedLayerObjects

```
private void writeScavengedLayerObjects(com.badlogic.gdx.utils.Json js
on)
```

Converts the scavengedLayerObjects HashMap into a String and writes it to the Profile's JSON file.

**Parameters:**

`json` - Json object where the HashMap is written

### readInventory

```
private void readInventory(com.badlogic.gdx.utils.Json json,
                            com.badlogic.gdx.utils.JsonValue jsonData)
```

Reads the inventory from the Profile's JSON file and converts it into an Inventory instance, so that the user can have his saved Inventory back.

**Parameters:**

`json` - Json object where the inventory HashMap is read from.

`jsonData` - JsonValue object used to read data

### readScavengedLayerObjects

```
private void readScavengedLayerObjects(com.badlogic.gdx.utils.Json
                     json,com.badlogic.gdx.utils.JsonValue jsonData)
```

Reads the String stored inside the JSON file and converts it into a HashMap for the scavengedLayerObjects variable.

### getScavengedLayerObjects

```
public java.util.ArrayList<java.lang.Integer> getScavengedLayerObjects
                                                (int row,int col)
```

Returns a list of all of the objectIds that have been scavenged on the given TerrainLayer, denoted by its row and column.

**Parameters:**

`row` - The row of the TerrainLayer

`col` - The column of the TerrainLayer

**Returns:**

A list of objectIds of each object scavenged on the given TerrainLayer

### addScavengedLayerObject

```
public void addScavengedLayerObject(GameObject gameObject)
```

Adds the given GameObject as a scavenged GameObject. It is added as a scavenged GameObject at the TerrainLayer where it resides, so that the GameObject is never instantiated there again.

**Parameters:**

`gameObject` - The GameObject which was scavenged

### addScavengedLayerObject

```
public void addScavengedLayerObject(int row, int col,int objectId)
```

Adds a scavenged object to the TerrainLayer, specified with the layer's cell coordinates. Accepts the objectId of the scavenged GameObject. Makes it so that the GameObject won't respawn the next time the layer is displayed.

**Parameters:**

`row` - The TerrainLayer's row

`col` - The TerrainLayer's column

`objectId` - The objectId of the scavenged GameObject

### getLoadout

```
public Loadout getLoadout()
```

Gets the loadout used by the player.

### setLoadout

```
public void setLoadout(Loadout loadout)
```

Sets the loadout used by the player.

### getInventory

```
public Inventory getInventory()
```

Retrieves the player's inventory, which contains all of the items collected by the player.

### setInventory

```
public void setInventory(Inventory inventory)
```

Sets the player's inventory, which contains all of the items collected by the player.

### getScavengedLayerObjects

```
public java.util.HashMap<java.lang.Integer,java.util.HashMap<java.lang
.Integer,java.util.ArrayList<java.lang.Integer>>> getScavengedLayerObj
ects()
```

Returns a HashMap containing an array for each TerrainLayer. This array specifies the objectIds of the GameObjects that have been scavenged on that layer. First key is the TerrainLayer's row, and second is the TerrainLayer's column.

### setScavengedLayerObjects

```
public void setScavengedLayerObjects(java.util.HashMap<java.lang.Integ
er,java.util.HashMap<java.lang.Integer,java.util.ArrayList<java.lang.I
nteger>>> scavengedLayerObjects)
```

Sets the HashMap containing objectId arrays for each TerrainLayer. These array specify the objectIds of the GameObjects that have been scavenged on a certain layer.

### isFirstTimeCreate

```
public boolean isFirstTimeCreate()
```

Returns true if the profile was just created. In fact, if the player has not saved the profile since creating it, this method returns true.

### setFirstTimeCreate

```
public void setFirstTimeCreate(boolean firstTimeCreate)
```

Sets whether or not this is the first time the profile is created. In fact, if the player has not saved the profile since creating it, firstTimeCreate should be true.

### II.3.1.42 *SoundManager* and *MusicManager*

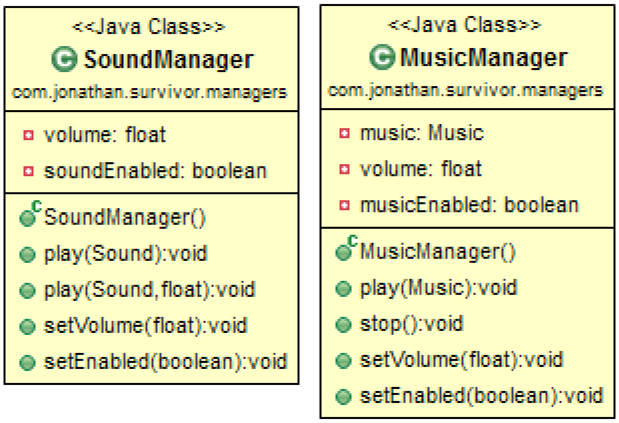


**Figure 64 : SoundManager and MusicManager class diagrams**

The *SoundManager* is responsible for playing all of the sounds in the game, whereas the *MusicManager* is responsible for playing the music. Note that music and sound was separated to ensure that, if ever we wanted music to play at a different volume than the sound effects, we could easily control their volumes independently, without any necessary workarounds.

```
public class SoundManager
extends java.lang.Object
```

## Field Detail

### volume

```
private float volume
```

Stores the volume to play all sound clips at by default

### soundEnabled

```
private boolean soundEnabled
```

Stores whether or not sound is enabled.

## Constructor Detail

### SoundManager

```
public SoundManager()
```

Instantiates a new sound manager.

## Method Detail

### play

```
public void play(com.badlogic.gdx.audio.Sound sound)
```

Plays a sound instance at maximum volume relative to the sound manager's volume. If music is disabled, the music is put on standby until music is enabled.

**Parameters:**

`sound` - the sound to play

### play

```
public void play(com.badlogic.gdx.audio.Sound sound,
        float volume)
```

Plays a sound instance at a custom volume. If sound is disabled, the sound is not played.

**Parameters:**

`sound` - the sound to play

`volume` - the volume at which to play the sound

### setVolume

```
public void setVolume(float volume)
```

Set the volume of the manager. Any subsequent sound will be played at this volume by default. The volume has range zero (quiet) to 1 (loudest).

**Parameters:**

`volume` - the new volume at which the sounds will all be played (between 0 and 1)

### setEnabled

```
public void setEnabled(boolean enabled)
```

Set whether the music is enabled or not. If disabled, music does not play.

**Parameters:**

`enabled` - if true, sound effects are allowed to play.

```
public class MusicManager
extends java.lang.Object
```

## Field Detail

### music

```
private com.badlogic.gdx.audio.Music music
```

Stores the music which is currently being played by the manager. Note that, if music is disabled, this variable is populated with the music that will be played once the music is re-enabled

### volume

```
private float volume
```

Stores the volume to play all music clips at

### musicEnabled

```
private boolean musicEnabled
```

Stores whether or not music is enabled.

## Constructor Detail

### MusicManager

```
public MusicManager()
```

Instantiates a new music manager.

## Method Detail

### play

```
public void play(com.badlogic.gdx.audio.Music music)
```

Plays a music instance at the managers volume setting. If music is disabled, the music is put on standby until music is enabled.

**Parameters:**

`music` - the music to play

### stop

```
public void stop()
```

Stop the current playing music.

### setVolume

```
public void setVolume(float volume)
```

Set the volume of the manager. Any subsequent music will be played at this volume, range zero (quiet) to 1 (loudest).

**Parameters:**

`volume` - the new volume of the music, between 0 and 1.

### setEnabled

```
public void setEnabled(boolean enabled)
```

Set whether the music is enabled or not. If disabled, music does not play.

**Parameters:**

`enabled` - if true, the music is enabled

### II3.1.43 *PreferencesManager*

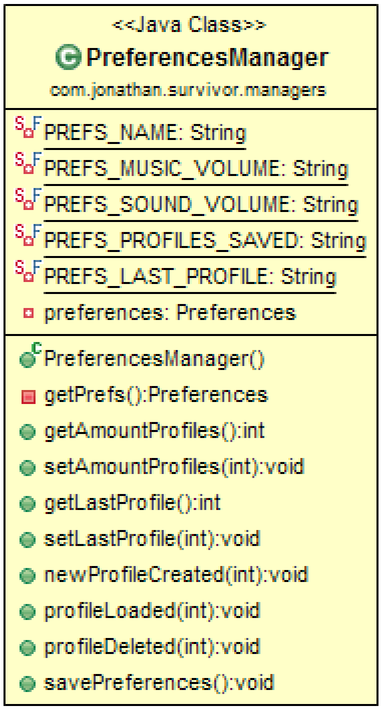


**Figure 65: PreferencesManager class diagram**

The *PreferencesManager* is responsible for handling the player's preferences. The player's preferences persist throughout the application. That is, the are saved onto the hard drive through the *Preferences* instance, which allow the user to keep the same settings, even upon application quit. Note that the preferences are universal. That is, no matter which profile the user selects, his preferences will remain the same. Thus, anything which should persist through application quit, and that should be independent of the profile the user selects is considered a preference.

```
public class PreferencesManager
extends java.lang.Object
```

## Field Detail

### PREFS_NAME

```
private static final java.lang.String PREFS_NAME
```

The name of the Preferences instance saved on the device.

### PREFS_MUSIC_VOLUME

```
private static final java.lang.String PREFS_MUSIC_VOLUME
```

The key for the music volume preference.

### PREFS_SOUND_VOLUME

```
private static final java.lang.String PREFS_SOUND_VOLUME
```

The key for the SFX volume preference.

### PREFS_PROFILES_SAVED

```
private static final java.lang.String PREFS_PROFILES_SAVED
```

The key for the amount of profiles saved by the player.

### PREFS_LAST_PROFILE

```
private static final java.lang.String PREFS_LAST_PROFILE
```

The key for the preference containing the last profile loaded by the player.

### preferences

```
private com.badlogic.gdx.Preferences preferences
```

Stores the Preferences instance used to save and retrieve player save information.

## Constructor Detail

### PreferencesManager

```
public PreferencesManager()
```

Creates a default PreferencesManager.

## Method Detail

### getPrefs

```
private com.badlogic.gdx.Preferences getPrefs()
```

Returns the Preferences instance used to modify and access the player's preferences.

### getAmountProfiles

```
public int getAmountProfiles()
```

Returns the amount of profiles that the user has saved on the hard drive.

### setAmountProfiles

```
public void setAmountProfiles(int numProfiles)
```

Sets the amount of profiles that have been saved by the player on the hard drive.

### getLastProfile

```
public int getLastProfile()
```

Returns the profileId of the last profile used by the player.

### setLastProfile

```
public void setLastProfile(int profileId)
```

Saves the profileId of the last profile loaded by the player.

### newProfileCreated

```
public void newProfileCreated(int profileId)
```

Called when a new profile is created. Allows the PreferencesManager to update the amount of profiles saved by the player. Also allows the manager to record this profile as the last profile the user has loaded so that the user can continue from this profile the next time the game is loaded.

**Parameters:**

`profileId` - The ID of the profile that has been created

### profileLoaded

```
public void profileLoaded(int profileId)
```

Called when a profile is loaded. Accepts the id of the loaded profile. The given profile will be loaded the next time the user presses "Continue".

**Parameters:**

`profileId` - the profile id of the loaded profile

### profileDeleted

```
public void profileDeleted(int profileId)
```

Called when a profile is deleted. Decrements the amount of saved profiles by one to ensure that the manager keeps track of the amount of profiles the user has saved.

**Parameters:**

`profileId` - The ID of the profile that was deleted. If this profile is the one that is supposed to be loaded when "Continue" is pressed, the PreferencesManager changes the profile that will be loaded when "Continue" is pressed.

### savePreferences

```
public void savePreferences()
```

Saves the preferences to the hard drive.

### II.3.1.44 *ProfileManager*

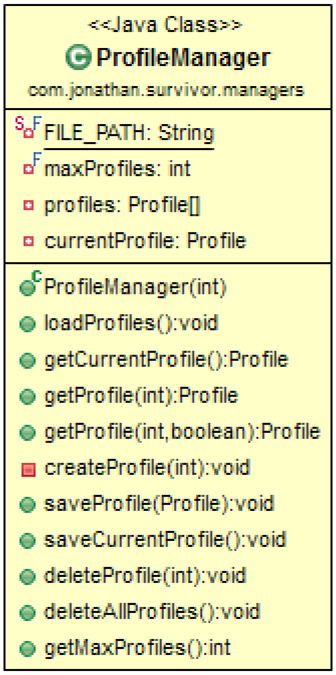


**Figure 66 : ProfileManager class diagram**

The *ProfileManager* is used to manage all of the player's profiles. It loads the profiles from the hard drive and stores them in an array.

The first variable in the class is a constant named *FILE_PATH*. It simply holds the file path where the profiles are saved to the hard drive. Next, the *maxProfiles* integer holds the maximum amount of profiles the user can have. Additionally, the *profiles* array holds a reference to every profile that has been created by the user. Conversely, what the *currentProfile* variable does is hold a reference to the profile that is currently being used by the player. It is the profile that the user selected in the *World Select Menu*.

In terms of constructors, the *ProfileManager* only has one constructor, which accepts the maximum amount of profiles the user can have. The constructor then calls *loadProfiles()*, which reads all of the profiles saved on the hard drive, and converts them into *Profile* objects. These objects are then stored inside the *profiles* array.

Next, the class has a *getProfile(index:int):Profile* method. It accepts the index for the profile that will be returned by the method. Conversely, the *createProfile(int)* method creates a new *Profile* object, saves it in the hard drive, and stores it inside the *profiles* array in the index passed as an argument. The role of the *saveProfile(Profile)* method is to convert a profile object into a string, and save it into a file of JSON type. Similarly, the *saveCurrentProfile()* does the same thing, except it does not accept any parameters, and instead saves the player's current profile to the hard drive.

The *deleteProfile(index:int)* and the *deleteAllProfiles()* methods are self-explanatory. The former deletes the *Profile* stored in the given index in the *profiles* array, and the latter deletes all of the player's profiles, including the JSON files saved on the hard drive.

```
public class ProfileManager
extends java.lang.Object
```

### Field Detail

#### FILE_PATH

```
private static final java.lang.String FILE_PATH
```

Stores the local file path for the profiles. Note that it ends with an underscore as it will be proceeded by "[id].json"

#### numProfiles

```
private int numProfiles
```

Stores the amount of profiles created by the player in order to determine how many should be loaded from the hard drive.

### profiles

```
private com.badlogic.gdx.utils.Array<Profile> profiles
```

Stores an array of every profile that has been read by the ProfileManager to avoid re-reading JSON files. Note that the index into the profiles array for a given profile is the same as to the profile's id.

### currentProfile

```
private Profile currentProfile
```

Stores the current profile being used by the user.

## Constructor Detail

### ProfileManager

```
public ProfileManager(int numProfiles)
```

Creates a profile manager, specifying the maximum amount of profiles the user can hold.

**Parameters:**

> `numProfiles` - Specifies how many profiles the manager will retrieve from the hard drive. Should correspond to amount of profiles the user has saved.

## Method Detail

### loadProfiles

```
public void loadProfiles()
```

Loads the profiles existing in the hard drive and populates the profiles:Profile[] array.

### loadProfile

```
public Profile loadProfile(int profileId)
```

Loads the profile with the given ID from the hard drive and returns it.

**Parameters:**

`profileId` - the id of the profile to load

**Returns:**

the profile loaded from the hard drive

### getCurrentProfile

```
public Profile getCurrentProfile()
```

Returns the current profile being used by the user. The current profile is the last one that was retrieved from getProfile(profileId):Profile. By default, if that method was never called, the last profile will be retrieved.

### getProfile

```
public Profile getProfile(int profileId)
```

Gets the profile with the given id, ranging from 0 to MAX_PROFILES-1. If the profile doesn't exist, it is not created nor saved to the hard drive.

**Parameters:**

`profileId` - the profile id of the profile that needs to be retrieved

**Returns:**

the profile with the given id

### getProfile

```
public Profile getProfile(int profileId, boolean createNew)
```

Gets the profile with the given id, ranging from 0 to MAX_PROFILES-1. Should be called after loadProfiles() to ensure that profiles have been loaded from the hard drive.

**Parameters:**

`profileId` - The id of the profile, ranging from 0 to ProfileManager.MAX_PROFILES-1.

`createNew` - If true, a new profile will be created if it doesn't exist on the hard drive.

**Returns:**

The profile with the given ID

### createProfile

```
public Profile createProfile(int profileId)
```

Creates a profile with the given profile ID, and saves it to the hard drive. Also sets the created profile to be the current user profile.

**Parameters:**

`profileId` - the profile id of the profile to create

**Returns:**

a new profile with with the given ID

### saveProfile

```
public void saveProfile(Profile profile)
```

Saves the profile to the hard drive. The file name depends on the ID of the profile passed as a parameter.

**Parameters:**

`profile` - the profile to save to the hard drive

### saveCurrentProfile

```
public void saveCurrentProfile()
```

Saves the current profile to the hard drive as a JSON file.

### deleteProfile

```
public void deleteProfile(int profileId)
```

Deletes a profile with the given ID from the hard drive.

**Parameters:**

`profileId` - the profile id of the profile to delete from the hard drive and RAM

### shiftProfiles

```
private void shiftProfiles(int profileId)
```

Shifts all the saved profiles from indices [profileId+1,numProfiles] to indices [profileId,numProfiles-1]. Called when the profile with the given id is deleted to ensure that the empty spot from the deleted profile is filled.

**Parameters:**

`profileId` - the profile id to which all the other profiles are shifted to

### unloadProfiles

```
public void unloadProfiles()
```

Unloads all of the profiles loaded by the ProfileManager. The player will lose any changes made to his current profile. This method essentially clears every Profile reference stored in the ProfileManager. Note that other classes can still hold references to the profiles. Therefore, pay attention to how those those references are used once this method is called.

### deleteAllProfiles

```
public void deleteAllProfiles()
```

Deletes all profiles from the hard drive, if they exist.

### isEmpty

```
public boolean isEmpty()
```

Returns true if the ProfileManager does not have any loaded profiles.

### getNumProfiles

```
public int getNumProfiles()
```

Gets the number of profiles the user has saved on the hard drive.

## II.3.1.45 *Settings*

The *Settings* class is used as a convenience class in order to save player data. First, the *Settings* holds the *profile* which it wants to save to the hard drive. Next, it holds the *profileManager*, used to save a profile to the hard drive. Finally, it holds the *world* instance. This is needed in order to read player information and save it to the *Profile* instance.

The default constructor creates a new *Settings* instance with null member variables. The second constructor accepts the *ProfileManager* from which it will save profiles to the hard drive. Next, the biggest constructor also accepts a *ProfileManager*, along with the *Profile* instance inside which to save player information, and a *World* instance, from which player data is extracted.

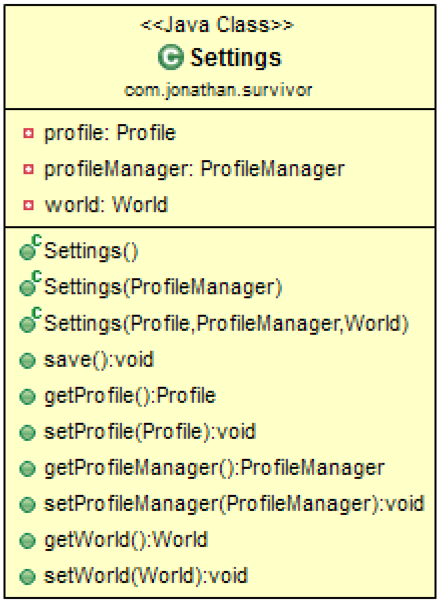


**Figure 67 : Settings class diagram**

The *Settings* class also has the *save()* method. When called inside the *GameScreen*, the *Profile* in the class's member variable is taken. Then, this profile's member variables are updated to hold the player's new information. Then, the *profileManager* is used to save the profile to the hard drive. The rest of the methods are simple getters and setters.

```
public class Settings
extends java.lang.Object
```

## Field Detail

### profile

```
private Profile profile
```

Stores the profile where data will be saved.

### profileManager

```
private ProfileManager profileManager
```

Used to save the profile held by the Settings instance.

### world

```
private World world
```

Stores the world from which we retrieve player data to save.

## Constructor Detail

### Settings

```
public Settings()
```

Creates an empty settings instance

### Settings

```
public Settings(ProfileManager profileManager)
```

Creates a Settings instance which uses the given ProfileManager to save the player profile.

### Settings

```
public Settings(Profile profile, ProfileManager profileManager,
                World world)
```

Accepts the profile to save, the ProfileManager used to save the profile, and world from which player data is retrieved.

**Parameters:**

`profile` - The Profile instance which is saved to the hard drive by this Settings instance

`profileManager` - The ProfileManager used to save the profile

`world` - The World instance, whose information is used to update the profile according to the changes in the world.

## Method Detail

### save

```
public void save()
```

Saves player information to the profile registered to this instance.

### saveLastXPos

```
private void saveLastXPos()
```

Saves the last x-position of the player before saving the profile. Note that this position is relative to the layer where he currently resides.

### getProfile

```
public Profile getProfile()
```

Retrieves the profile where the Settings instance saves player data.

### setProfile

```
public void setProfile(Profile profile)
```

Sets the profile where player data will be saved.

### getProfileManager

```
public ProfileManager getProfileManager()
```

Gets the ProfileManager used to save the profile to the hard drive.

### setProfileManager

```
public void setProfileManager(ProfileManager profileManager)
```

Sets the ProfileManager used to save the profile to the hard drive.

### getWorld

```
public World getWorld()
```

Gets the World where player information is read and saved to the hard drive.

### setWorld

```
public void setWorld(World world)
```

Gets the World where player information is read and saved to the hard drive.

### II.3.1.46 *GameObjectManager*

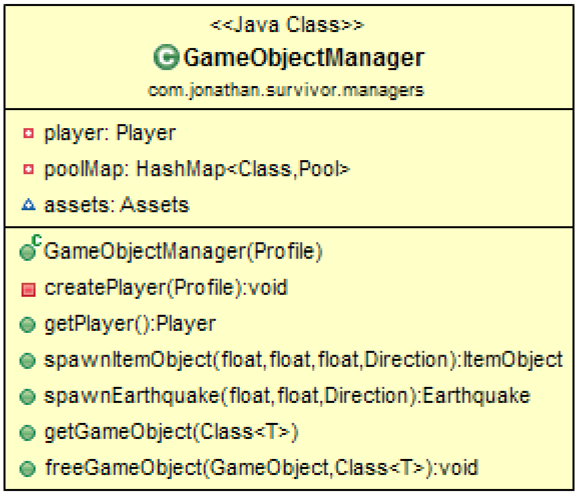

<<Java Class>>
GameObjectManager
com.jonathan.survivor.managers
player: Player
poolMap: HashMap<Class,Pool>
assets: Assets
GameObjectManager(Profile)
createPlayer(Profile):void
getPlayer():Player
spawnItemObject(float,float,float,Direction):ItemObject
spawnEarthquake(float,float,Direction):Earthquake
getGameObject(Class<T>)
freeGameObject(GameObject,Class<T>):void

**Figure 68 : GameObjectManager class diagram**

The *GameObjectManager* manages the instantiation and pooling of *GameObjects* in the game. For instance, if ever a tree needs to be placed in the world, this manager will be used to retrieve or instantiate a *Tree* object.

In terms of data fields, the *GameObjectManager* has a *player* variable, which holds the *Player* instance that the user controls. Next, the class has a *poolMap*. This HashMap has a *Class* as a key and a *Pool* instance as a value. The key will always be a *GameObject* subclass. Thus, each *GameObject* subclass has a pool where its objects are stored. For instance, the *Tree.class* key has a *TreePool* value. A *Pool* subclass allows objects to be stored inside an array when they are not needed, and retrieved when they are needed. When an object is no longer in use, it can be freed back into the pool for later re-use. This way, every *TerrainLayer* recycles and re-uses the same *GameObject* instances. This prevents the program from instantiating *GameObjects* often.

On a different note, the constructor of the class simply accepts a *Profile* instance. From this profile, the *player* will be created using the *createPlayer(Profile)* method. This method will create an instance of *Player* from the save data stored inside the profile. Ignoring the getters and setters for the *player*, the *spawnItemObject(Class, float, float, Direction):ItemObject* method first accepts a *Class*. This class has to be an *Item* subclass. The method will then retrieve an *ItemObject* from the *poolMap*, and spawn it in the position denoted by the second and third parameters. Note that the *Direction* parameter accepts either *Direction.LEFT* or *Direction.RIGHT*. This is the direction where the item will fly when spawned. The item is spawned using *ItemObject.spawn(…)*.

Alternatively, the *spawnZombie(x:float, y:float):Zombie* first retrieves a *Zombie* instance from the *poolMap*. Then, the zombie is placed at the (x,y) position given by the parameters of the method and returns this zombie instance.

The most important methods of the class are the *getGameObject(Class<T>):T* method and the *freeGameObject(GameObject,Class)* method. The former accepts a *Class*. A *GameObject* of class *T* is returned. For instance, if *GameObjectManager.getGameObject(Tree.class)* is called, a pooled *Tree* object is returned. This method allows classes to easily access pooled *GameObjects* of any class. Conversely, the *freeGameObject(GameObject, Class)* performs the reverse operation of the *getGameObject(…)* method. It inserts the *GameObject* parameter in the *poolMap.get(Class)* pool. Like this, when *GameObjects* are no longer needed, if, for instance, a zombie is killed, it can be freed back into the *ZombiePool* to be recycled.

```
public class GameObjectManager
extends java.lang.Object
```

### Field Detail

#### player

```
private Player player
```

Stores the player GameObject.

#### poolMap

```
private java.util.HashMap<java.lang.Class,com.badlogic.gdx.utils.Pool>
poolMap
```

Stores HashMap of GameObjectPools where every GameObject class is a key to a pool of its GameObjects. Used for easy management of pools.

#### assets

```
Assets assets
```

Stores the Assets singleton used to access the visual assets used by the game.

### Constructor Detail

#### GameObjectManager

```
public GameObjectManager(Profile profile)
```

Accepts the player's profile to re-create some GameObjects using save data.

### Method Detail

#### createPlayer

```
private void createPlayer(Profile profile)
```

Creates the player GameObject, along with his skeleton. Accepts profile to re-create the player with his old settings.

#### getPlayer

```
public Player getPlayer()
```

Returns the Player GameObject being managed by the manager.

#### spawnItemObject

```
public ItemObject spawnItemObject(float x, float y,
              float velocityMultiplier,Human.Direction direction)
```

Spawns an ItemObject at the given position. The (x,y) position is the bottom-center of the position where the ItemObject is spawned.

**Parameters:**

`velocityMultiplier` - Multiplier which allows certain items to fly further or closer when spawned.

`direction` - Specifies the direction in which the item will fly when spawned.

#### spawnEarthquake

```
public Earthquake spawnEarthquake(float x, float y,
                                  Human.Direction direction)
```

Spawns an Earthquake instance at the given bottom-center (x,y) position. The earthquake will move in the given direction passed as an argument.

### getGameObject

```
public <T> T getGameObject(java.lang.Class<T> goClass)
```

Gets a tree GameObject of the given class cached inside one of the Manager's pools. No same GameObject will be returned twice until it is freed using freeGameObject().

**Type Parameters:**

T - The type of GameObject that wants to be retrieved.

**Returns:**

A GameObject of the given class

### freeGameObject

```
public <T> void freeGameObject(GameObject gameObject,
                                   java.lang.Class<T> goClass)
```

Frees a gameObject back inside the manager's internal GameObject pools. Tells the manager that the GameObject is no longer in use, and that the it can be returned when getGameObject() is called.

### II.3.1.47 *Cell*

The *Cell* class represents a pair of cell coordinates. It holds a row and a column. The default constructor instantiates a cell at row and column zero. The second constructor accepts the row and column of the *Cell* as parameters.

The class also has the *moveLeft/Right/Up/Down()*. This shifts the cell left, right, up, and down, respectively. For instance, if the *Cell* holds coordinates (3,0), and *moveUp()* is called, the cell coordinates are changed to (4,0). The *set(row:int, col:int)* method changes the row and column of the cell. The rest of the methods are simple getters and setters.

(See next page for class diagram)

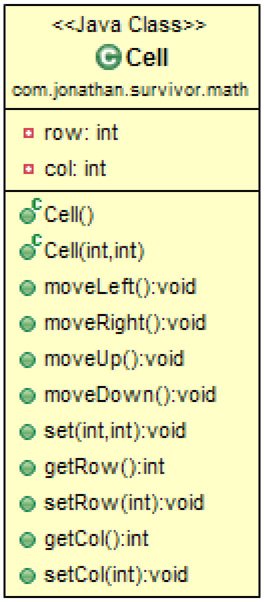


**Figure 69 : Cell class diagram**

```
public class Cell
extends java.lang.Object
```

## Field Detail

### row

```
private int row
```

Stores the row and column of the cell.

### col

```
private int col
```

Stores the row and column of the cell.

## Constructor Detail

### Cell

```
public Cell()
```

Creates a cell at (0,0)

### Cell

```
public Cell(int row,int col)
```

Creates a cell with the specified row and column.

**Parameters:**

`row` - the row of the cell

`col` - the col of the cell

## Method Detail

### moveLeft

```
public void moveLeft()
```

Moves the cell left by decrementing the column

### moveRight

```
public void moveRight()
```

Moves the cell right by incrementing the column

### moveUp

```
public void moveUp()
```

Moves the cell up by incrementing the row

### moveDown

```
public void moveDown()
```

Moves the cell down by decrementing the row

### set

```
public void set(int row,int col)
```

Sets the row and the column of the cell to the specified values.

### getRow

```
public int getRow()
```

Retrieves the cell's row

### setRow

```
public void setRow(int row)
```

Sets the cell's row

### getCol

```
public int getCol()
```

Returns the cell's column

### setCol

```
public void setCol(int col)
```

Sets the cell's column

II.3.1.48 *Line*

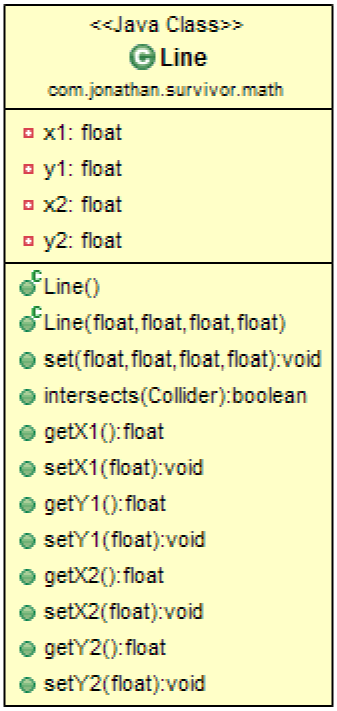


**Figure 70: Line class diagram**

A *Line* object simply represents a line going from one end-point to the other. It is used when the player is firing his ranged weapon. In fact, at this moment, a line object is used to model the trajectory line of the gun. It allows the user to know where his gun will fire.

```
public class Line
extends java.lang.Object
```

## Field Detail

### x1

```
private float x1
```

End-point of the line.

### y1

```
private float y1
```

End-point of the line.

### x2

```
private float x2
```

End-point of the line.

### y2

```
private float y2
```

End-point of the line.

## Constructor Detail

### Line

```
public Line()
```

Creates a line with both end points at (0,0).

### Line

```
public Line(float x1, float y1, float x2, float y2)
```

Creates a line with the given end-points.

## Method Detail

### set

```
public void set(float x1, float y1, float x2, float y2)
```

Changes the two end-points of the line.

### intersects

```
public boolean intersects(Collider collider)
```

Returns true if the line intersects with a given collider.

**Parameters:**

`collider` - the collider which is tested to intersect the line

**Returns:**

true, if the line intersects the collider.

### getX1

```
public float getX1()
```

Returns the x-position of the first end-point of the line.

### setX1

```
public void setX1(float x1)
```

Sets the x-position of the first end-point of the line.

### getY1

```
public float getY1()
```

Returns the y-position of the first end-point of the line.

### setY1

```
public void setY1(float y1)
```

Sets the y-position of the first end-point of the line.

### getX2

```
public float getX2()
```

Returns the x-position of the second end-point of the line.

**setX2**

```
public void setX2(float x2)
```

Sets the x-position of the second end-point of the line.

**getY2**

```
public float getY2()
```

Returns the y-position of the second end-point of the line.

**setY2**

```
public void setY2(float y2)
```

Sets the y-position of the second end-point of the line.

### II.3.1.49 *Item*

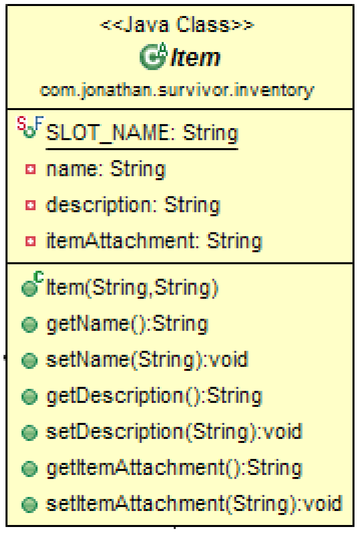


**Figure 71 : Item class diagram**

The *Item* class is a data holder for each item in the world. Whereas an *ItemObject* represents a physical entity, an *Item* is used to provide information on a specific item.

First, each item has a *name*. This is the name displayed for each item inside the *Crafting Menu*. Next, an item has an *inventorySprite*. This is an image which is used to represent the item inside the *Crafting Menu*. Finally, an item has an *itemAttachment* String. This String denotes the name of the attachment inside Spine used to display the item. An *ItemObject* needs this attachment name to represent the item as a physical object in the world. In fact, the attachment tells the *ItemObject* which image to display, since an *attachment* is essentially an image inside Spine. Note that the subclasses do not accept this String as an argument to keep the constructor's parameters easy to follow. The String, however, may be modified using its setter method.

```
public abstract class Item
extends java.lang.Object
```

## Field Detail

### SLOT_NAME

```
public static final java.lang.String SLOT_NAME
```

Holds the name of the slot where the item's image is attached for the Item skeleton in Spine. Allows to change an ItemObject's appearance.

### name

```
private java.lang.String name
```

Stores the name of the item.

### description

```
private java.lang.String description
```

Holds the description of the item.

### itemAttachment

```
private java.lang.String itemAttachment
```

Stores the name of the image which displays the item's image in Spine. This is the image that will be displayed on the ItemObject containing this item.

## Constructor Detail

### Item

```
public Item(java.lang.String name,java.lang.String description)
```

Creates an item with the given name and description.

**Parameters:**

name - the name if the item

`description` - the description of the item.

## Method Detail

### getName

```
public java.lang.String getName()
```

Gets the item's name.

### setName

```
public void setName(java.lang.String name)
```

Sets the item's name.

### getDescription

```
public java.lang.String getDescription()
```

Gets the item's description.

### setDescription

```
public void setDescription(java.lang.String description)
```

Sets the item's description.

### getItemAttachment

```
public java.lang.String getItemAttachment()
```

Retrieves the name of the attachment used in Spine to display this Item when it is an object in the world.

### setItemAttachment

```
public void setItemAttachment(java.lang.String itemAttachment)
```

Sets the name of the attachment used in Spine to display this Item when it is an object in the world.

### II.3.1.50 *Weapon*

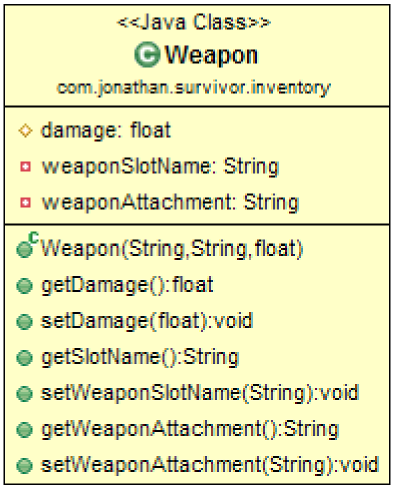

<<Java Class>>
Weapon
com.jonathan.survivor.inventory
damage: float
weaponSlotName: String
weaponAttachment: String
Weapon(String,String,float)
getDamage():float
setDamage(float):void
getSlotName():String
setWeaponSlotName(String):void
getWeaponAttachment():String
setWeaponAttachment(String):void

**Figure 72 : Weapon class diagram**

The abstract *Weapon* represents a weapon as both a physical entity in the world and as an item in the player's inventory.

First, a *Weapon* has a *damage* variable. This represents the amount of health points an entity loses when hit by the weapon. Next, the weapon has an attachment, which denotes the name of the image used to display the weapon when it is equipped by the player. In subclasses, the *weaponAttachment* data field is not populated by a constructor argument. In order to prevent the constructor from bloating with an excess of arguments, the *weaponAttachment* is changed using the data field's mutator.

```
public class Weapon
extends Item
```

**Field Detail**

**damage**

```
protected float damage
```

The amount of damage done by the weapon with one hit.

### weaponSlotName

```
private java.lang.String weaponSlotName
```

Stores the slot in Spine where the image of the gun is attached.

### weaponAttachment

```
private java.lang.String weaponAttachment
```

Stores the name of the attachment image used to display the weapon.

## Constructor Detail

### Weapon

```
public Weapon(java.lang.String name, java.lang.String description,
              float damage)
```

Creates a weapon with the given name, description, and damage.

**Parameters:**

`name` - the name of the weapon

`description` - the description of the weapon

`damage` - the damage dealt by the weapon

## Method Detail

### getDamage

```
public float getDamage()
```

Gets the amount of damage the weapon deals in one hit.

### setDamage

```
public void setDamage(float damage)
```

Sets the amount of damage the weapon deals in one hit.

### getSlotName

```
public java.lang.String getSlotName()
```

Returns the slot name on the player in Spine where the weapon's image is attached.

### setWeaponSlotName

```
public void setWeaponSlotName(java.lang.String slotName)
```

Sets the slot name on the player in Spine where the weapon's image is attached.

### getWeaponAttachment

```
public java.lang.String getWeaponAttachment()
```

Gets the name of the image (attachment) in Spine which displays the weapon.

### setWeaponAttachment

```
public void setWeaponAttachment(java.lang.String attachmentName)
```

Sets the name of the image (attachment) used in Spine to display the weapon.

## II.3.1.51 *RangedWeapon*

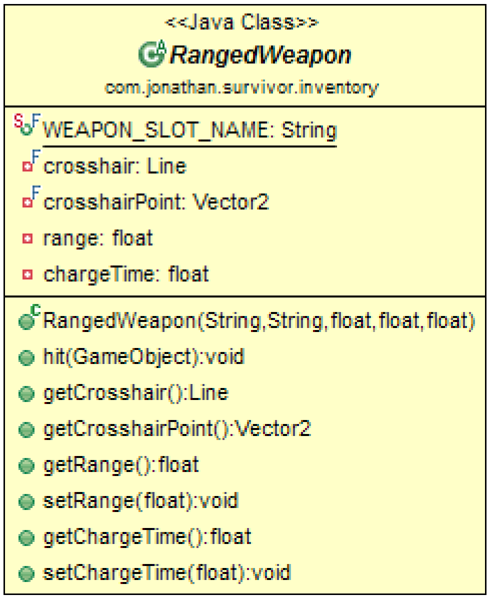


**Figure 73 : RangedWeapon class diagram**

A *RangedWeapon* acts as a superclass for every ranged weapon in the game.

```
public abstract class RangedWeapon
```

```
extends Weapon
```

## Field Detail

### WEAPON_SLOT_NAME

```
public static final java.lang.String WEAPON_SLOT_NAME
```

Stores the name of the slot on the player in Spine where ranged weapon images are stored.

### crosshair

```
private final Line crosshair
```

Holds a line which traces the trajectory of the gun's bullet. In essence, this is location where a zombie can get hit by the weapon.

### crosshairPoint

```
private final Vector2 crosshairPoint
```

Stores the position where the gun's crosshair should be placed. This is where the trajectory lines starts.

### range

```
private float range
```

Stores the range of the gun, which is the distance in world units that the gun's bullet can travel.

### chargeTime

```
private float chargeTime
```

Holds the amount of time it takes to charge the weapon completely.

## Constructor Detail

### RangedWeapon

```
public RangedWeapon(java.lang.String name,
                    java.lang.String description, float damage,
                    float range, float chargeTime)
```

Accepts the name, description, damage, and charge time of the melee weapon.

**Parameters:**

`name` - the name of the RangedWeapon

`description` - the description of the RangedWeapon

`damage` - the damage of the RangedWeapon

`range` - the range of the RangedWeapon

`chargeTime` - the charge time of the RangedWeapon

## Method Detail

### hit

```
public void hit(GameObject gameObject)
```

Called when the RangedWeapon has hit a GameObject and should deal damage to it.

**Parameters:**

`gameObject` - the game object to hit

### getCrosshair

```
public Line getCrosshair()
```

Returns a line depicting where the bullet will travel when the ranged weapon is shot.

### getCrosshairPoint

```
public Vector2 getCrosshairPoint()
```

Returns the position where the start of the crosshair should be placed on the weapon. This is usually the tip of the ranged weapon.

### getRange

```
public float getRange()
```

Gets the range of the ranged weapon in world units. This is the distance that a bullet can travel relative to the tip of the weapon.

### setRange

```
public void setRange(float range)
```

Sets the range of the ranged weapon in world units. This is the distance that a bullet can travel relative to the tip of the weapon.

### getChargeTime

```
public float getChargeTime()
```

Gets the amount of time it takes to charge the weapon.

### setChargeTime

```
public void setChargeTime(float chargeTime)
```

Gets the amount of time it takes to charge the weapon completely.

## II.3.1.52 *MeleeWeapon*

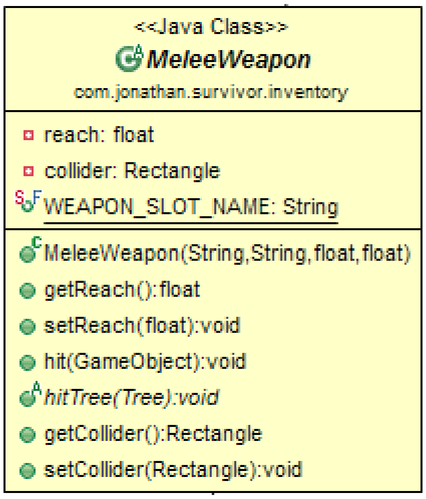


**Figure 74 : Visual representation of the MeleeWeapon class**

On the other hand, the *MeleeWeapon* class acts as a superclass for all melee weapons. Its primary role is to allow other classes to distinguish between a melee and a ranged weapon just by the weapon's supertype.

The main difference between the ranged and the melee weapon is that melee weapons have a *reach*. This data field is a floating-point value which denotes a distance in meters. It determine how far from the player the weapon can attack. The ranged weapon, on the other hand, does not have a range as these weapons always have infinite reach.

```
public abstract class MeleeWeapon
extends Weapon
```

## Field Detail

### reach

```
private float reach
```

Stores the horizontal reach of the melee weapon in world units.

### collider

```
private Rectangle collider
```

Holds the rectangle collider put around the melee weapon when it is equipped. Allows to test for hit detection.

### WEAPON_SLOT_NAME

```
public static final java.lang.String WEAPON_SLOT_NAME
```

Stores the name of the slot on the player in Spine where melee weapon images are stored.

## Constructor Detail

### MeleeWeapon

```
public MeleeWeapon(java.lang.String name,
              java.lang.String description, float damage,float reach)
```

Accepts the name, description, damage, and range of the melee weapon.

**Parameters:**

`name` - the name of the MeleeWeapon

`description` - the description of the MeleeWeapon

`damage` - the damage of the MeleeWeapon

`reach` - the reach of the MeleeWeapon

## Method Detail

### getReach

```
public float getReach()
```

Gets the horizontal reach in world units of the melee weapon.

### setReach

```
public void setReach(float range)
```

Sets the horizontal reach in world units of the melee weapon.

### hit

```
public void hit(GameObject gameObject)
```

Called when the MeleeWeapon has hit a GameObject and should deal damage to it.

**Parameters:**

`gameObject` - the game object to hit

### hitTree

```
public abstract void hitTree(Tree tree)
```

Called when the MeleeWeapon has hit a tree and should deal damage to it.

**Parameters:**

`tree` - the tree to hit

### getCollider

```
public Rectangle getCollider()
```

Returns the collider around the melee weapon used to detect when the weapon hits an enemy.

### setCollider

```
public void setCollider(Rectangle collider)
```

Sets the collider around the melee weapon used to detect when the weapon hits an enemy.

### II.3.1.53 *Axe & Rifle*

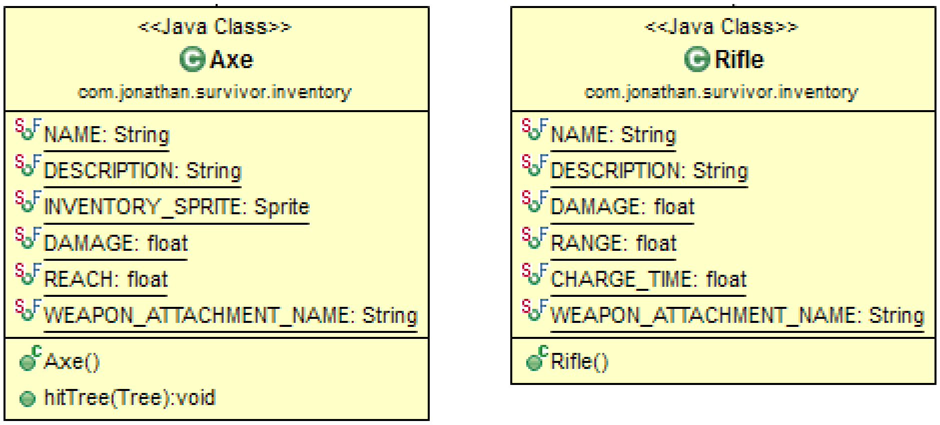


**Figure 75 : Axe and Rifle class diagrams**

The *Axe* and the *Rifle* classes represent the two weapons available in the world. Their constructors don't accept any arguments, as every instance of these classes have the same data fields. Thus, the constructors simply populate their member variables with the appropriate values.

```
public class Axe
extends MeleeWeapon
```

#### Field Detail

##### NAME

```
public static final java.lang.String NAME
```

The name of the axe.

##### DESCRIPTION

```
public static final java.lang.String DESCRIPTION
```

The description of the axe.

##### DAMAGE

```
public static final float DAMAGE
```

The damage inflicted by the axe.

### REACH

```
public static final float REACH
```

The horizontal range of the axe.

### WEAPON_ATTACHMENT_NAME

```
public static final java.lang.String WEAPON_ATTACHMENT_NAME
```

Stores the name of the image placed on the player in Spine which displays the Axe.

## Constructor Detail

### Axe

```
public Axe()
```

Creates an axe.

## Method Detail

### hitTree

```
public void hitTree(Tree tree)
```

Called when the MeleeWeapon has hit a tree and should deal damage to it.

**Specified by:**

`hitTree` in class `MeleeWeapon`

**Parameters:**

`tree` - the tree to hit

```
public class Rifle
extends RangedWeapon
```

## Field Detail

### NAME

```
public static final java.lang.String NAME
```

Stores the properties of the rifle.

### DESCRIPTION

```
public static final java.lang.String DESCRIPTION
```

Stores the description of the rifle.

### DAMAGE

```
public static final float DAMAGE
```

The damage of the rifle.

### RANGE

```
public static final float RANGE
```

The length of the crosshair of the rifle.

### CHARGE_TIME

```
public static final float CHARGE_TIME
```

The amount of time the rifle takes to charge.

### WEAPON_ATTACHMENT_NAME

```
public static final java.lang.String WEAPON_ATTACHMENT_NAME
```

Stores the name of the image placed on the player in Spine which displays the rifle.

## Constructor Detail

### Rifle

```
public Rifle()
```

Creates a rifle.

### II.3.1.53 *Craftable*

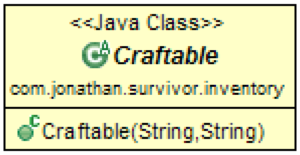


**Figure 76 : Craftable class diagram**

The abstract *Craftable* class simply denotes any items that are not weapons. Its constructor accepts the name of the item along with the *Sprite* which is used to display the item in the *Crafting Menu*.

```
public abstract class Craftable
extends Item
```

## Constructor Detail

### Craftable

```
public Craftable(java.lang.String name, java.lang.String description)
```

Instantiates a new craftable item.

**Parameters:**

`name` - the name of the craftable item

`description` - the description of the craftable item

(See next page for *Craftable* subclasses)

### II.3.1.55 *Craftable subclasses*

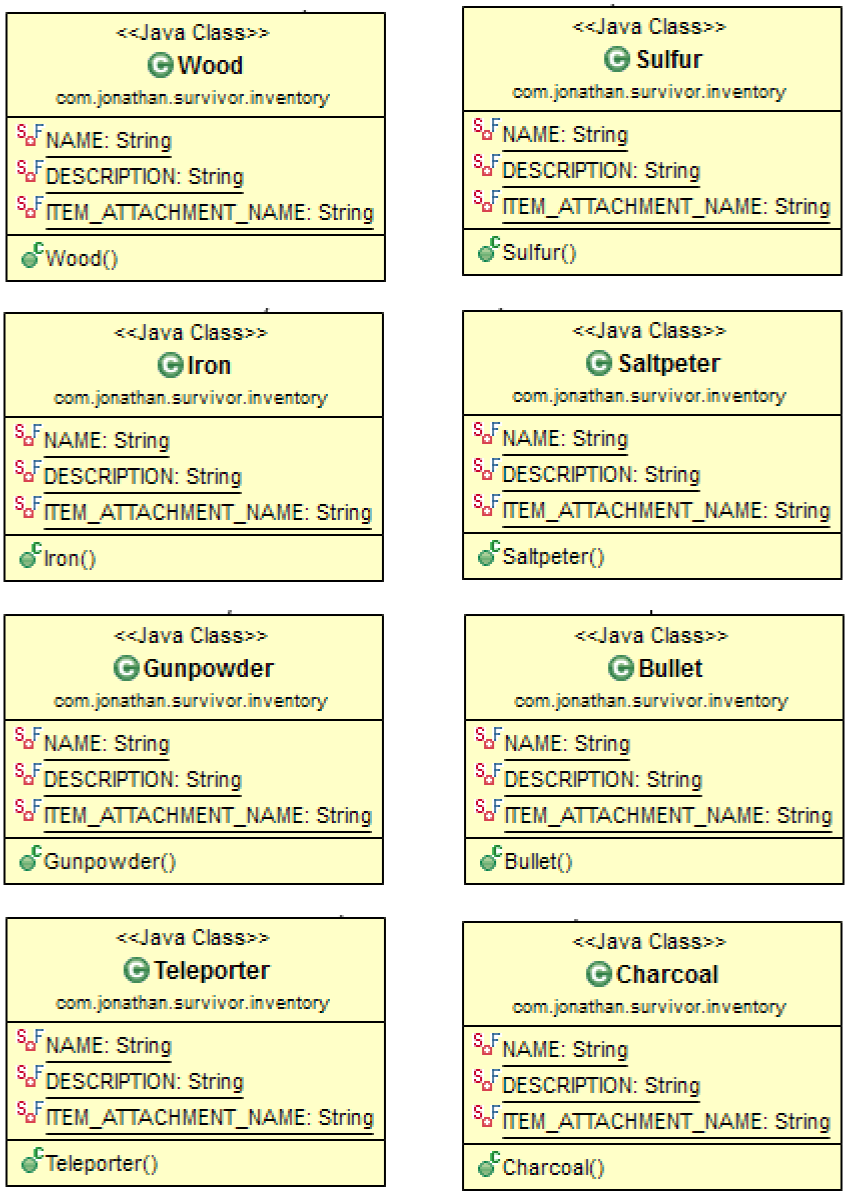


**Figure 77 : Wood, Sulfur, Iron, Saltpeter, Gunpowder, Bullet, Teleporter and Charcoal class diagrams**

Above are the *Craftable* subclasses which act as data containers for every craftable item in the world. They simply contain no-arg constructors which populates the class's data fields. Given that the classes only contain constants used to define each item's data, a single sample is given to describe the data fields and methods of the items.

```
public class Teleporter
extends Craftable
```

## Field Detail

### NAME

```
private static final java.lang.String NAME
```

The name of the teleporter item.

### DESCRIPTION

```
private static final java.lang.String DESCRIPTION
```

The description of the teleporter item.

### ITEM_ATTACHMENT_NAME

```
private static final java.lang.String ITEM_ATTACHMENT_NAME
```

Stores the name of the image placed on the ItemObject skeleton in Spine which displays the Teleporter.

## Constructor Detail

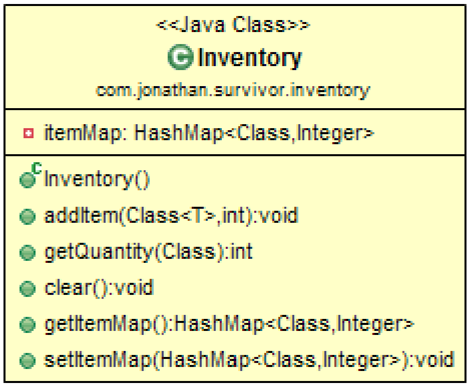
### Teleporter

```
public Teleporter()
```

Creates a Teleporter item that can be displayed in an inventory slot.

(See next page for *Inventory* class)

### II.3.1.56 *Inventory*



<<Java Class>>
Inventory
com.jonathan.survivor.inventory
itemMap: HashMap<Class,Integer>
Inventory()
addItem(Class<T>,int):void
getQuantity(Class):int
clear():void
getItemMap():HashMap<Class,Integer>
setItemMap(HashMap<Class,Integer>):void

**Figure 78 : Inventory class diagram**

The *Inventory* acts as a container for all of the items which the player possesses. It holds all of the player's items through a HashMap called *itemMap*. Its key is the item's class, and the integer is the quantity of this item stored inside the inventory. The default constructor of the class simply creates an *Inventory* with a null *itemMap*. The *Profile* used by the player takes the saved *itemMap*, and passes it to the *Inventory* using the *itemMap's* setter method. The only notable method in the class is the *addItem(Class, int)* method, which accepts a class which is usually an *Item* subclass. It also accepts the quantity to put inside the inventory for that specific item.

```
public class Inventory
extends java.lang.Object
```

## Field Detail

### itemMap

```
private java.util.HashMap<java.lang.Class,java.lang.Integer> itemMap
```

Holds a map between an Item class and the amount of items of that class inside the inventory.

## Constructor Detail

### Inventory

```
public Inventory()
```

Creates an empty inventory.

## Method Detail

### addItem

```
public <T extends Item> void addItem(java.lang.Class<T> itemClass,
                                     int quantity)
```

Adds the Item of the given class inside the Inventory in the given quantity.

**Type Parameters:**

T - the generic type

**Parameters:**

`itemClass` - the item to add to the inventory

`quantity` - the quantity of that item to add to the inventory

### getQuantity

```
public int getQuantity(java.lang.Class itemClass)
```

Returns the quantity of items of the given class inside the inventory.

**Parameters:**

`itemClass` - the item whose quantity is returned

**Returns:**

the quantity of the given item held in the inventory.

### clear

```
public void clear()
```

Clears all of the items stored in the inventory.

### getItemMap

```
public java.util.HashMap<java.lang.Class,java.lang.Integer> getItemMap
()
```

Returns the ItemMap which maps the Item classes to the amount of the item stored inside the inventory.

### setItemMap

```
public void setItemMap(java.util.HashMap<java.lang.Class,java.lang.Int
eger> itemMap)
```

Sets the ItemMap which maps the Item classes to the amount of the item stored inside the inventory.

II.3.1.57 *Loadout*

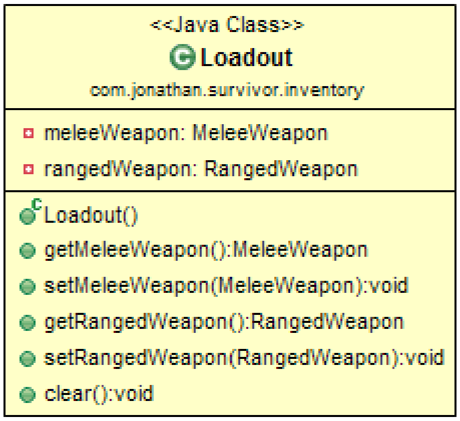

<<Java Class>>
Loadout
com.jonathan.survivor.inventory
meleeWeapon: MeleeWeapon
rangedWeapon: RangedWeapon
Loadout()
getMeleeWeapon():MeleeWeapon
setMeleeWeapon(MeleeWeapon):void
getRangedWeapon():RangedWeapon
setRangedWeapon(RangedWeapon):void
clear():void

**Figure 79 : Loadout class diagram**

The *Loadout* is a simple class. It simply acts as a container for the player's currently equipped weapons. The *Loadout* has a *meleeWeapon* and a *rangedWeapon* member variable, which respectively hold the melee weapon held by the user, along with the range weapon equipped by the player. The default constructor instantiates an empty loadout without any weapons.

```
public class Loadout
extends java.lang.Object
```

## Field Detail

### meleeWeapon

```
private MeleeWeapon meleeWeapon
```

Stores the MeleeWeapon held in the loadout.

### rangedWeapon

```
private RangedWeapon rangedWeapon
```

Holds the RangedWeapon equipped by the player.

## Constructor Detail

### Loadout

```
public Loadout()
```

Creates an empty loadout with nothing inside it.

## Method Detail

### getMeleeWeapon

```
public MeleeWeapon getMeleeWeapon()
```

Gets the MeleeWeapon held in the loadout.

### setMeleeWeapon

```
public void setMeleeWeapon(MeleeWeapon meleeWeapon)
```

Sets the MeleeWeapon held in the loadout.

### getRangedWeapon

```
public RangedWeapon getRangedWeapon()
```

Returns the RangedWeapon equipped by the player.

### setRangedWeapon

```
public void setRangedWeapon(RangedWeapon rangedWeapon)
```

Sets the RangedWeapon equipped by the player.

### clear

```
public void clear()
```

Clears all of the weapons held by the player.

II.3.1.58 *Pool Classes*

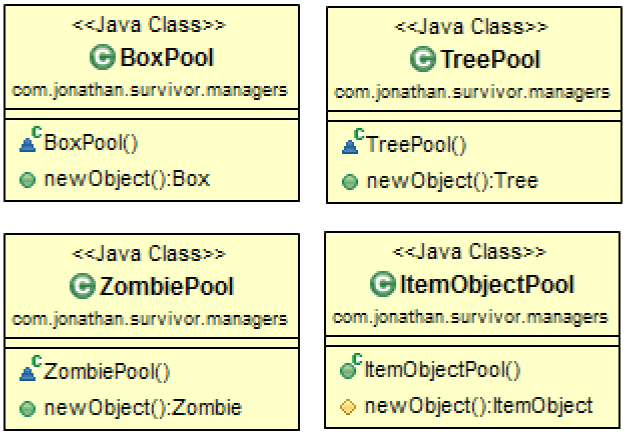


**Figure 80 : BoxPool, TreePool, ZombiePool and ItemObjectPool class diagrams**

The pool classes all extend *LibGDX's* pre-defined *Pool* class. What these pools do is hold a dynamic array of objects. Every time an object from that pool wants to be retrieved, the *Pool.obtain():T* method is called. This method can do one of two things. For one, if the pool has no free objects, it calls the *newObject():T* method, which instantiates a new object and returns it. If, however, the pool already has a free object inside its internal array, it returns it. In order to return an object inside the pool, the *Pool.free(T)* must be called, where *T* is the object type which the pool stores in its internal array. To use extend the *Pool* class, all that is needed is to override the *newObject():T* method to instantiate a new object of the type *T* and return it.

Note that pools were used as they heavily aid the game's performance. Instead of constantly instantiating and garbage collecting objects, objects are only created once, and re-used whenever they are needed. Note that the *Tree*, *Box*, *Zombie*, and *ItemObject* classes were pooled as they are constantly being reused and recycled as the player traverses across the world.

Given that the contents of each class is nearly identical, a single sample is given to describe the *BoxPool's* data fields and methods.

```
class BoxPool
extends com.badlogic.gdx.utils.Pool
```

## Constructor Detail

### BoxPool

```
BoxPool()
```

## Method Detail

### newObject

```
public Box newObject()
```

Called when no free objects are available in the pool, and a new one must be created.

**Specified by:**

`newObject` in class `com.badlogic.gdx.utils.Pool`

**Returns:**

a new Box instance, placed at (0,0)

(See next page for *TiledImage* class)

II.3.1.59 *TiledImage*

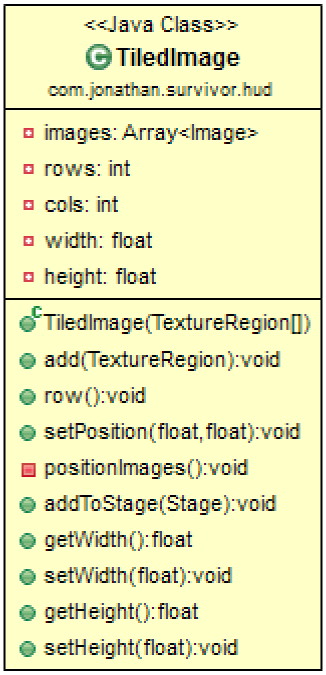


**Figure 81: TiledImage class diagram**

The *TiledImage* class allows multiple images to be grouped together in a grid. This is used for the main menu screens, where the backgrounds were too large to fit into one image. Thus, the backgrounds were split into two images, and grouped together with a *TiledImage* instance.

```
public class TiledImage
extends java.lang.Object
```

## Field Detail

### images

```
private com.badlogic.gdx.utils.Array<com.badlogic.gdx.scenes.scene2d.u
i.Image> images
```

Stores the images which form the TileImage.

### rows

```
private int rows
```

Stores the number of rows of images in the TimeImage.

### cols

```
private int cols
```

Holds the number of columns in which the images are laid out in the TileImage.

### width

```
private float width
```

Stores the total width of the TiledImage.

### height

```
private float height
```

Stores the total height of the TiledImage.

## Constructor Detail

### TiledImage

```
public TiledImage(com.badlogic.gdx.graphics.g2d.TextureRegion... regions)
```

Creates a TiledImage which will have its images laid out in rows and columns.

**Parameters:**

`regions` - The TextureRegions which will compose the TiledImage.

## Method Detail

### add

```
public void add(com.badlogic.gdx.graphics.g2d.TextureRegion region)
```

Adds an image to the TiledImage. Note that the image is added in the next column, after the image which was last added to the TiledImage.

**Parameters:**

`region` - the image to add to the TiledImage

### row

```
public void row()
```

Skip a row.

### setPosition

```
public void setPosition(float x, float y)
```

Sets the bottom-left position of the TiledImage at the given (x,y) coordinates.

**Parameters:**

x - the left x-position of the TiledImage

y - the bottom y-position of the TiledImage

### positionImages

```
private void positionImages()
```

Re-positions the images so that they are in a grid-like fashion, with the same amount of rows and columns specified in the member variables.

### addToStage

```
public void addToStage(com.badlogic.gdx.scenes.scene2d.Stage stage)
```

Adds the TiledImage to the stage, so that it can be rendered.

**Parameters:**

`stage` - the stage which will render the TiledImage

### getWidth

```
public float getWidth()
```

Retrieves the total width of the TiledImage.

### setWidth

```
public void setWidth(float width)
```

Sets the total width of the TiledImage.

### getHeight

```
public float getHeight()
```

Retrieves the total height of the TiledImage.

### setHeight

```
public void setHeight(float height)
```

Sets the total height of the TiledImage.

II.3.1.60 *KoAnimation* and *VersusAnimation* classes

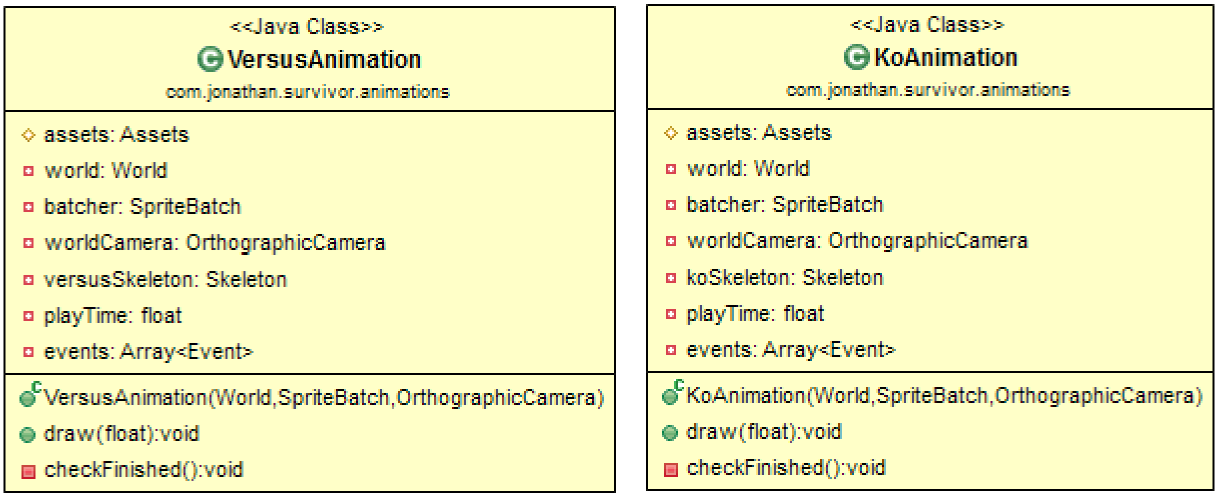


**Figure 82: VersusAnimation and KoAnimation class diagrams**

The *VersusAnimation* is played whenever the user enters combat with a zombie. It pauses the game, and allows the user to see that he has entered combat with a zombie. Conversely, the *KoAnimation* plays whenever the user wins or loses combat. It signals to the player that combat is over. Note that, since the two classes are almost identical, the data field, constructor and method details are only specified for the *KoAnimation* class.

```
public class KoAnimation
extends java.lang.Object
```

## Field Detail

### assets

```
protected Assets assets
```

Stores the Assets singleton of the game used to fetch assets to draw the Spine animations.

### world

```
private World world
```

Stores the world whose enterCombat() method we call when the versus animation is finished.

### batcher

```
private com.badlogic.gdx.graphics.g2d.SpriteBatch batcher
```

Stores the SpriteBatcher used to draw the animations.

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the OrthographicCamera used to view the world.

### koSkeleton

```
private com.esotericsoftware.spine.Skeleton koSkeleton
```

Stores the Spine Skeleton used to show the KO animation when a character dies in COMBAT mode.

### playTime

```
private float playTime
```

Holds the amount of time in seconds that the animation has been playing.

### events

```
private com.badlogic.gdx.utils.Array<com.esotericsoftware.spine.Event>
events
```

## Constructor Detail

### KoAnimation

```
public KoAnimation(World world,
                   com.badlogic.gdx.graphics.g2d.SpriteBatch batcher,
                   com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Accepts the SpriteBatch instance where Spine skeletons are drawn, and the camera used to view the world, which allows the animations to be centered on the screen. Also accepts the World, which this class will signal when the animation is done.

**Parameters:**

`world` - the world, whose methods are called when the KOAnimation is complete

`batcher` - the batcher used to draw the animation

`worldCamera` - the world camera where the animation is rendered

### Method Detail

#### draw

```
public void draw(float deltaTime)
```

Draws the animation to the center of the screen.

**Parameters:**

`deltaTime` - the execution time of the previous render call

#### checkFinished

```
private void checkFinished()
```

Checks if the versus animation has finished playing. If so, the world's correct methods are delegated to switch the player to combat mode.

(See next page for *CrosshairRenderer* class)

II.3.1.61 *CrosshairRenderer*

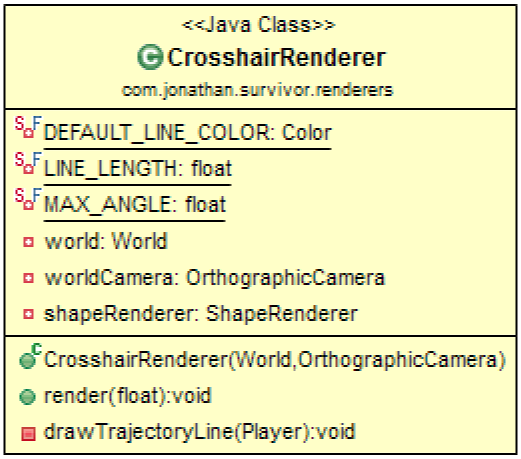

<<Java Class>>
CrosshairRenderer
com.jonathan.survivor.renderers
DEFAULT_LINE_COLOR: Color
LINE_LENGTH: float
MAX_ANGLE: float
world: World
worldCamera: OrthographicCamera
shapeRenderer: ShapeRenderer
CrosshairRenderer(World,OrthographicCamera)
render(float):void
drawTrajectoryLine(Player):void

**Figure 83: CrosshairRenderer class diagram**

The *CrosshairRenderer* is responsible for the ranged weapon's crosshair to the screen. In fact, whenever the player is charging his rifle, this class calls its *drawTrajectoryLine(Player)* method in order to draw two lines which form the rifle's crosshair.

```
public class CrosshairRenderer
extends java.lang.Object
```

**Field Detail**

**DEFAULT_LINE_COLOR**

```
private static final com.badlogic.gdx.graphics.Color
DEFAULT_LINE_COLOR
```

Stores the default color of the lines used to draw the crosshairs.

**LINE_LENGTH**

```
private static final float LINE_LENGTH
```

Holds the default length of a crosshair line/trajectory line.

**MAX_ANGLE**

```
private static final float MAX_ANGLE
```

Stores the max angle of the trajectory line when the player's ranged weapon has just begun charging.

### world

```
private World world
```

Stores the world whose information we use to render crosshairs.

### worldCamera

```
private com.badlogic.gdx.graphics.OrthographicCamera worldCamera
```

Stores the camera where the terrain is drawn. In this case, the world camera.

### shapeRenderer

```
private com.badlogic.gdx.graphics.glutils.ShapeRenderer shapeRenderer
```

Stores the ShapeRenderer instance used to draw the level geometry.

## Constructor Detail

### CrosshairRenderer

```
public CrosshairRenderer(World world,
             com.badlogic.gdx.graphics.OrthographicCamera worldCamera)
```

Accepts the world, from which information is gathered about the crosshairs to draw, and the camera where the crosshair lines will be drawn.

**Parameters:**

`world` - The World instance from which the player and his ranged weapon's information is extracted to draw the necessary crosshairs.

`worldCamera` - The camera where the crosshairs are drawn

## Method Detail

### render

```
public void render(float deltaTime)
```

Renders the given terrainLevel's geometry using OpenGL ES lines.

**Parameters:**

`deltaTime` - The amount of time passed in the last render call

### drawTrajectoryLine

```
private void drawTrajectoryLine(Player player)
```

Draws the player's trajectory line on the tip of his gun.

**Parameters:**

`player` - The crosshair is drawn on the RangedWeapon held by the given player

## II.3.2 Class Hierarchies

Important Notes:

1. To show the relationship between a class and its inner classes or its enumerations, a filled black circle was placed at the tip of the lines instead of a plus sign. This was due to limitations regarding the program used to generate the UML diagrams.

2. The details of enumerations are not explicitly explained in this section. Given that the enumeration constants were often self-explanatory, their explanations were omitted for the sake of brevity.

3. The method and data field explanations for inner classes and interfaces are given in this section. This allows the details of inner classes to be closer to their hierarchies, more effectively demonstrating the relationships between the inner classes and their main classes, and the interfaces and the classes which implement them.

4. The inner classes which represented mere button listeners were not included in this document for brevity's sake. There would be too many to list, and their implementation details would be trivial.

II.3.2.1 *Human* Hierarchy

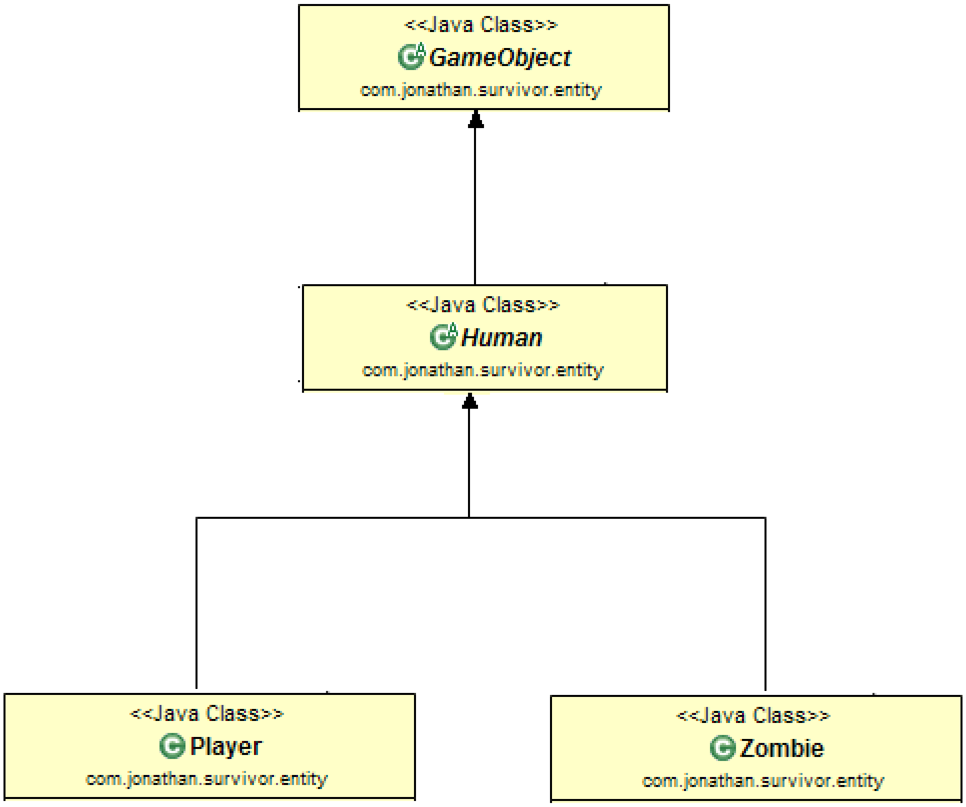

<<Java Class>>
GameObject
com.jonathan.survivor.entity
<<Java Class>>
Human
com.jonathan.survivor.entity
<<Java Class>>
Player
com.jonathan.survivor.entity
<<Java Class>>
Zombie
com.jonathan.survivor.entity

**Figure 84 : Human class hierarchy**

### II.3.2.2 *Human* Enumerations

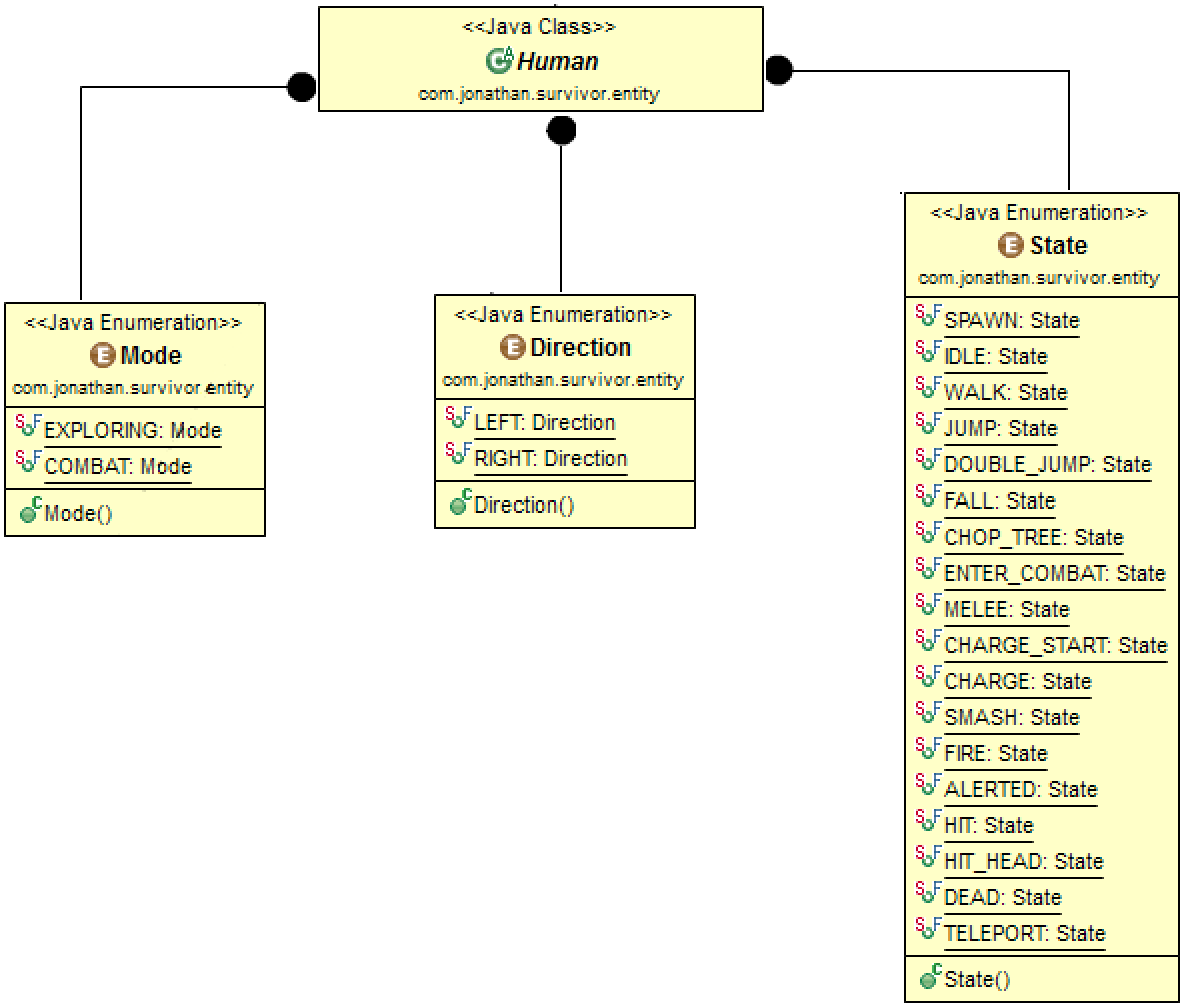


**Figure 85 : Enumerations related to the Human class**

II.3.2.3 *InteractiveObject* Hierarchy & Enumerations

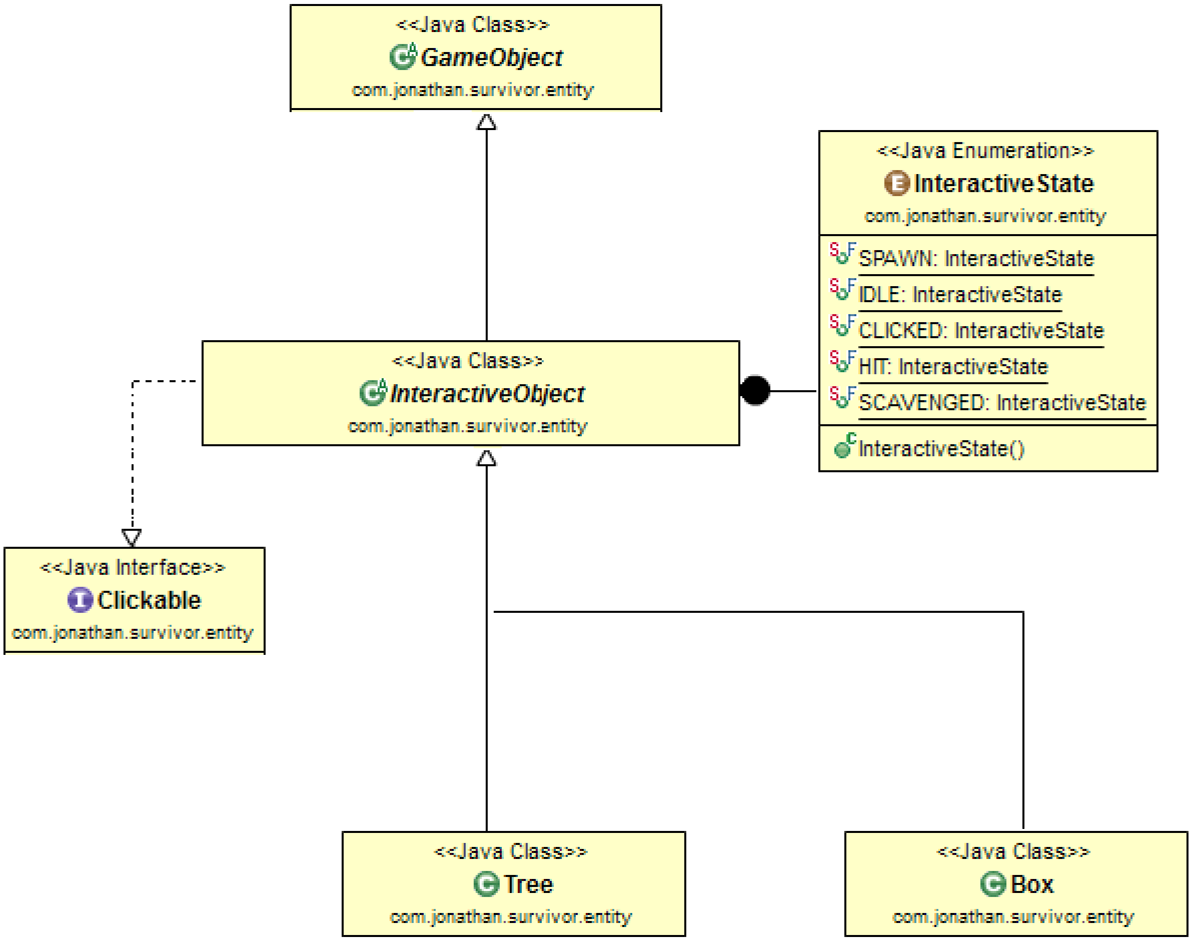


**Figure 86 : Human class hierarchy and related enumerations**

Note: the *Clickable* marker interface denotes a *GameObject*, which, when clicked, will call the *World.gameObjectClicked()* method.

### II.3.2.4 *ItemObject* Hierarchy & Enumerations

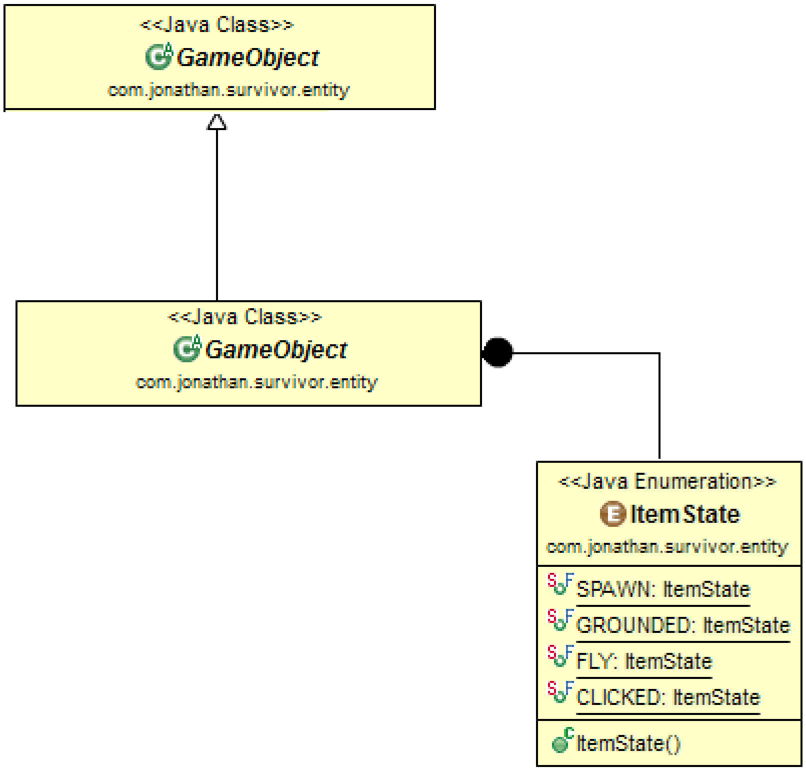


**Figure 87 : ItemObject class hierarchy and related enumerations**

### II.3.2.5 *Collider* Hierarchy

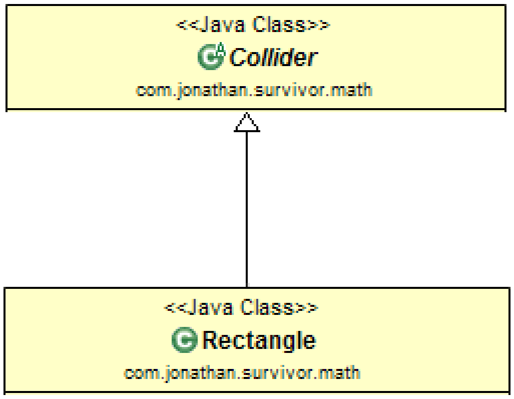


**Figure 88 : Collider class hierarchy**

II.3.2.6 *World* Hierarchy, Inner Classes, & Enumerations

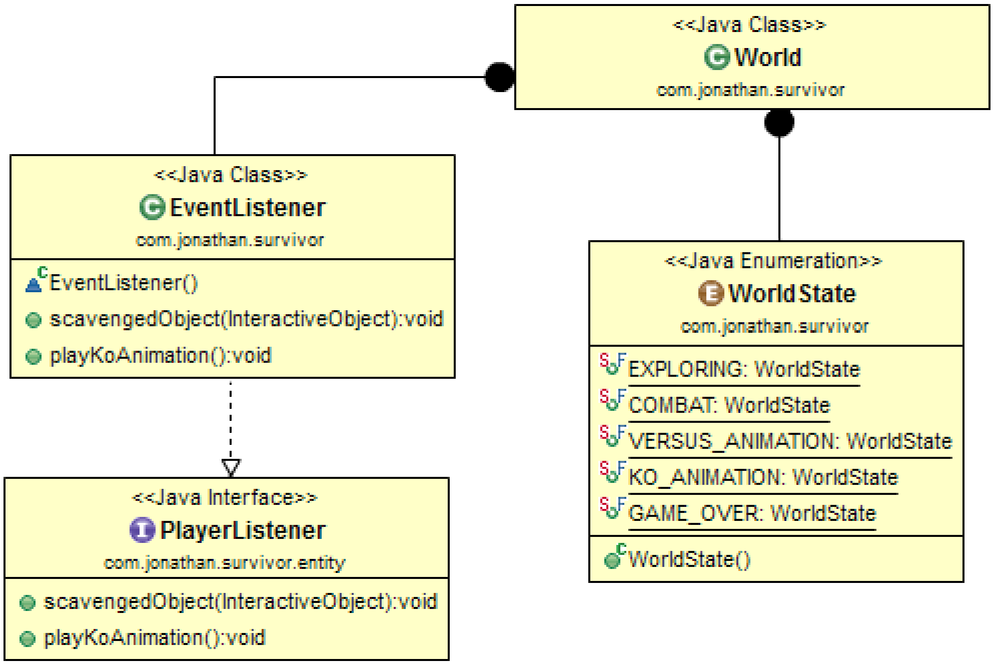


**Figure 89 : World hierarchy, Inner Classes, & related enumerations**

Note: The method *PlayerListener.scavengedObject(InteractiveObject)* method is called whenever the player has scavenged an object. The world receives this event, and reacts by dropping items next to the *InteractiveObject* passed an argument to the method. Conversely, the *playKoAnimation()* simply tells the *World* to start playing the *KoAnimation* when combat mode comes to an end.

II.3.2.7 *Screen* Hierarchy

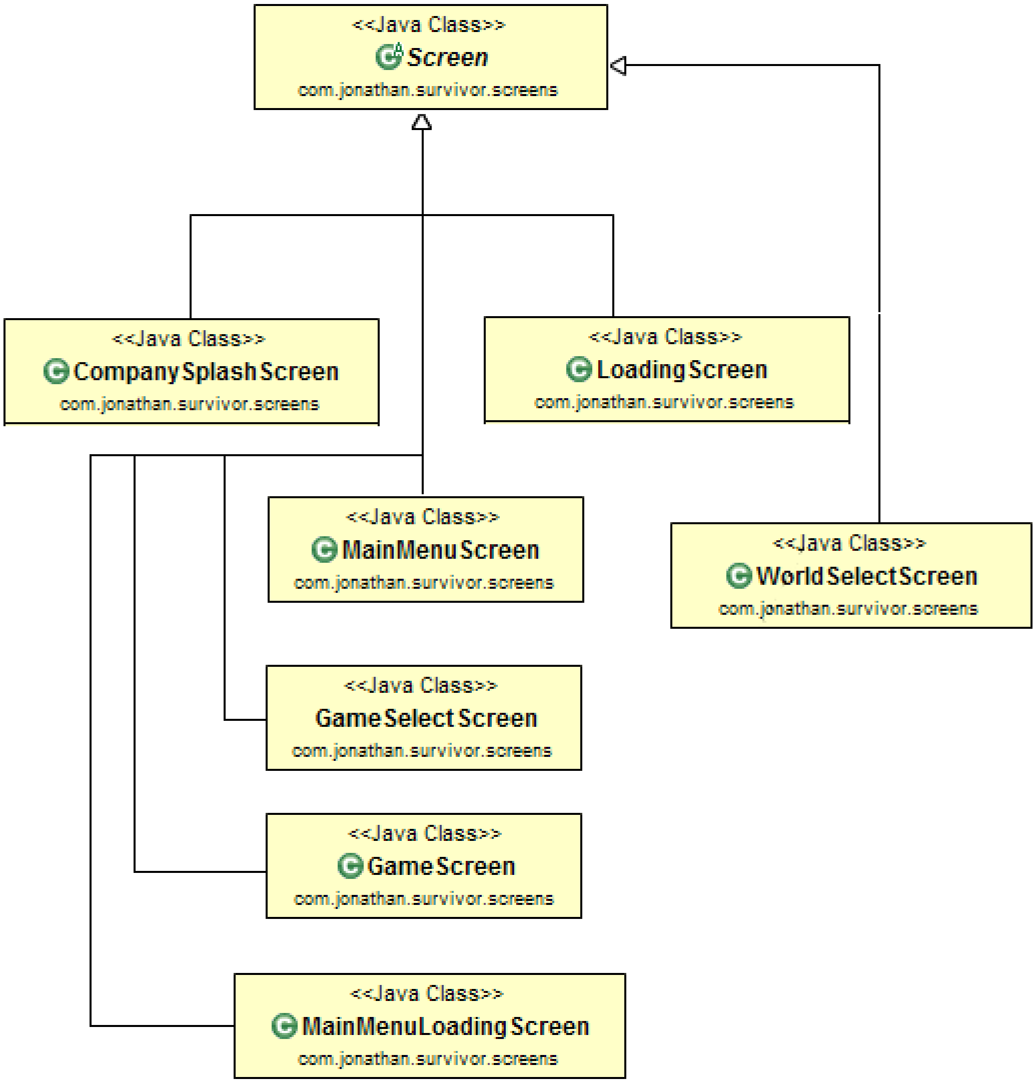

<<Java Class>>
Screen
com.jonathan.survivor.screens
<<Java Class>>
CompanySplashScreen
com.jonathan.survivor.screens
<<Java Class>>
LoadingScreen
com.jonathan.survivor.screens
<<Java Class>>
MainMenuScreen
com.jonathan.survivor.screens
<<Java Class>>
WorldSelectScreen
com.jonathan.survivor.screens
<<Java Class>>
GameSelectScreen
com.jonathan.survivor.screens
<<Java Class>>
GameScreen
com.jonathan.survivor.screens
<<Java Class>>
MainMenuLoadingScreen
com.jonathan.survivor.screens

**Figure 90 : Screen class hierarchy**

### II.3.2.8 *GameScreen* Enumerations & Inner Classes

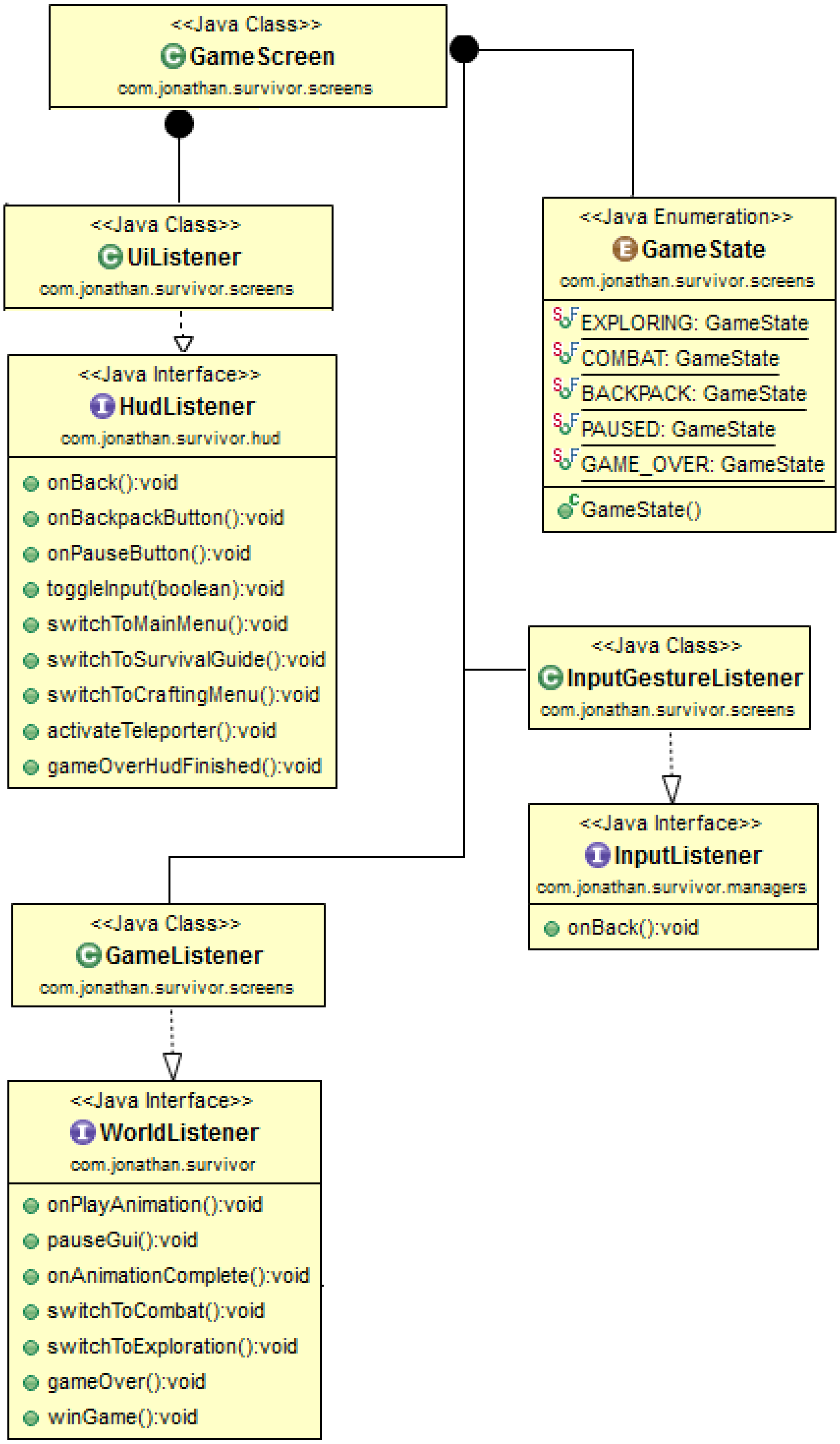


**Figure 91 : GameScreen related enumerations & inner classes**

Note: The details of the *WorldListener* and *HudListener* classes are given in this section. As such, the explanations are closer to the class diagrams themselves, as explained in the notes at the beginning of this section.

```
public interface WorldListener
```

Listens to events fired by the World and delegates them to the *GameScreen*.

### Method Detail

#### onPlayAnimation

```
void onPlayAnimation()
```

Called when an animation plays which overlays the screen. In this case, the GameScreen will pause the game until the animation is done.

#### pauseGui

```
void pauseGui()
```

Pauses the currently-active Heads-up-display so that no buttons can be pressed.

#### onAnimationComplete

```
void onAnimationComplete()
```

Called when an animation finishes playing. This is for overlay animations which fill the screen. When complete, the GameScreen knows to resume the game.

#### switchToCombat

```
void switchToCombat()
```

Delegated when the player switches to combat mode. Tells the GameScreen to switch to the combat HUD.

#### switchToExploration

```
void switchToExploration()
```

Delegated after the KO animation plays in COMBAT mode. Tells the GameScreen to switch the HUD back to the Exploration HUD.

#### gameOver

```
void gameOver()
```

Tells the GameScreen to switch to its GameOverHud.

### winGame

```
void winGame()
```

Called when the player's TELEPORT animation is done playing after crafting a teleporter. Transitions the player back to the main menu.

The *HudListener* is registered to each HUD instance. Whenever a button is pressed that should notify the GameScreen, one of the listener's methods are called, and the GameScreen can then handle the event by itself.

```
public interface HudListener
```

## Method Detail

### onBack

```
void onBack()
```

Called when the Back Button is pressed on any Hud instance

### onBackpackButton

```
void onBackpackButton()
```

Called when the Backpack Button is pressed. This is the button on the top-left of the screen in exploration mode.

### onPauseButton

```
void onPauseButton()
```

Called when the Pause Button is pressed whilst in-game.

### toggleInput

```
void toggleInput(boolean on)
```

Called when the user presses/releases a button on a HUD. Toggles input handling on/off. Allows/disallows input from changing the player's state.

**Parameters:**

on - if true, input is toggled on

### toggleGestures

```
void toggleGestures(boolean on)
```

Called when gesture processing should be switched on or off. If 'off', GestureManager stops listening for gestures and the user can no longer control the player with gestures.

**Parameters:**

on - if true, gestures are toggled on

### saveGame

```
void saveGame()
```

Delegates when the user presses the 'Save' button in the pause menu, and wants to save the game in his current profile.

### switchToMainMenu

```
void switchToMainMenu()
```

Called when the main menu button was pressed to transition to the main menu.

### switchToSurvivalGuide

```
void switchToSurvivalGuide()
```

Called when the survival guide button is pressed in the backpack, in order to transition to the survival guide hud.

### switchToCraftingMenu

```
void switchToCraftingMenu()
```

Called when the crafting button is pressed in the backpack. Transitions to the crafting HUD.

### activateTeleporter

```
void activateTeleporter()
```

Delegated by the CraftingHud when the player crafts a TimeMachine. Tells the GameScreen to make the player win the game.

### gameOverHudFinished

```
void gameOverHudFinished()
```

Called when the GameOverHud is finished displaying. Informs the GameScreen that the player has died.

II.3.2.9 *Hud* Hierarchy (a)

(Note: Hierarchy split into two due to lack of space)

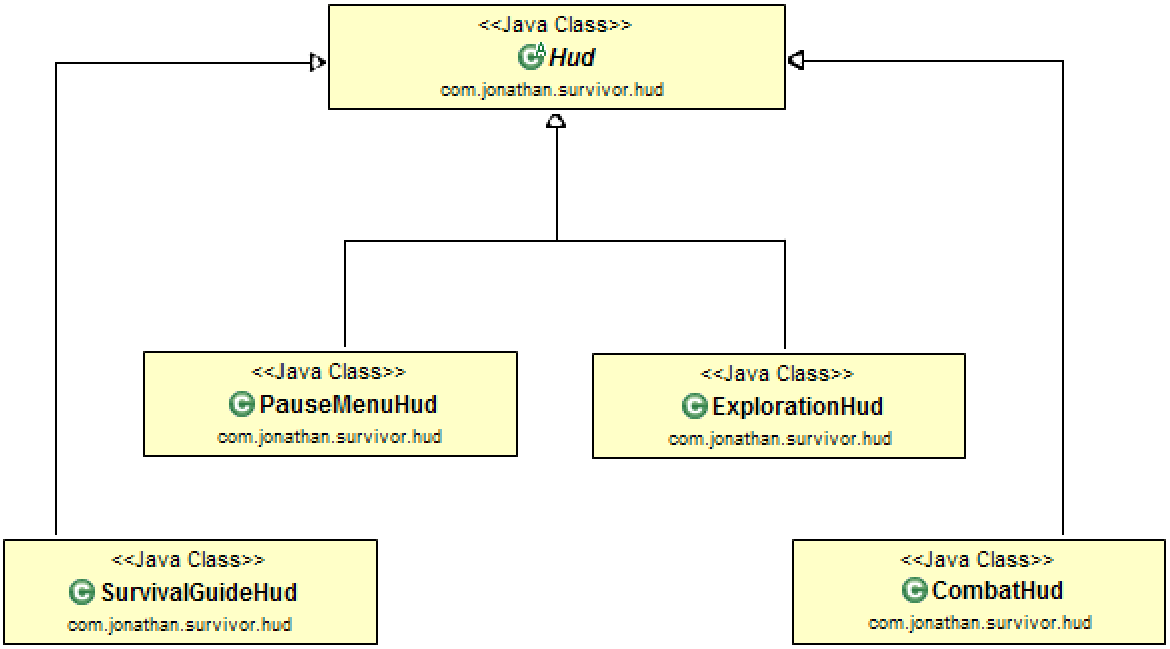

<<Java Class>>
Hud
com.jonathan.survivor.hud
<<Java Class>>
PauseMenuHud
com.jonathan.survivor.hud
<<Java Class>>
ExplorationHud
com.jonathan.survivor.hud
<<Java Class>>
SurvivalGuideHud
com.jonathan.survivor.hud
<<Java Class>>
CombatHud
com.jonathan.survivor.hud

**Figure 92 : Hud class hierarchy (a)**

### II.3.2.10 *Hud* Hierarchy (b)

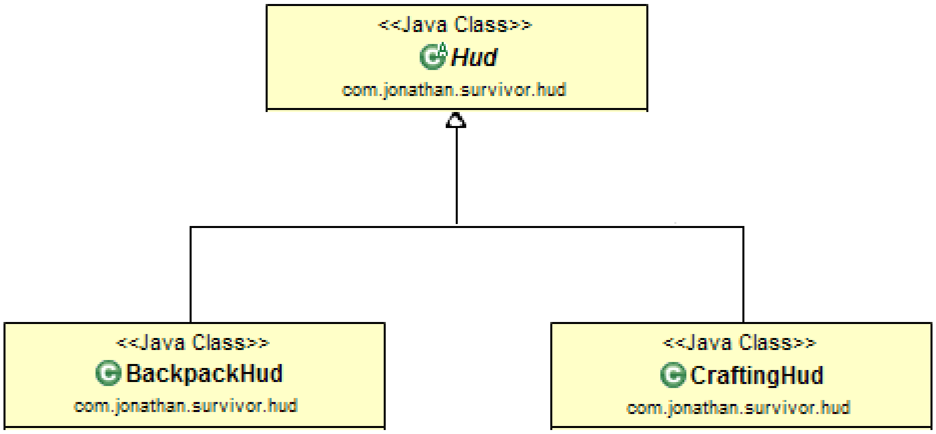

<<Java Class>>
Hud
com.jonathan.survivor.hud
<<Java Class>>
BackpackHud
com.jonathan.survivor.hud
<<Java Class>>
CraftingHud
com.jonathan.survivor.hud

**Figure 93 : Hud class hierarchy (b)**

### II.3.2.11 *CraftingTable* inner class

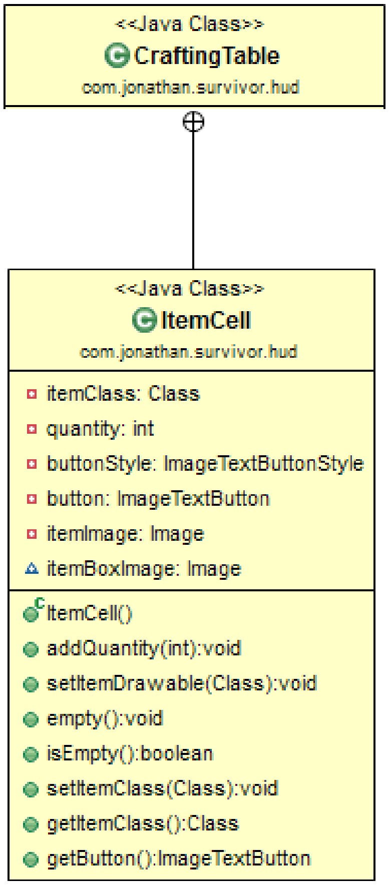


**Figure 94 : CraftingTable inner class**

```
private class CraftingTable.ItemCell
extends java.lang.Object
```

A cell representing an item as a button in the crafting table.

## Field Detail

### itemClass

```
private java.lang.Class itemClass
```

Stores the Item subclass represented by this cell.

### quantity

```
private int quantity
```

Holds the amount of items of the same type contained in the cell.

### buttonStyle

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageTextButton.ImageTextBu
ttonStyle buttonStyle
```

Holds the style which dictates the look of the button.

### button

```
private com.badlogic.gdx.scenes.scene2d.ui.ImageTextButton button
```

Stores the button which displays the given item and its quantity in the crafting table.

### itemImage

```
private com.badlogic.gdx.scenes.scene2d.ui.Image itemImage
```

Holds the image displaying the item on the cell.

### itemBoxImage

```
com.badlogic.gdx.scenes.scene2d.ui.Image itemBoxImage
```

Stores the image displaying the grey box background to each button.

## Constructor Detail

### CraftingTable.ItemCell

```
public CraftingTable.ItemCell()
```

Creates a default ItemCell with no item inside.

## Method Detail

### addQuantity

```
public void addQuantity(int amount)
```

Adds the given amount of items to this cell. Can be negative

### setItemDrawable

```
public void setItemDrawable(java.lang.Class itemClass)
```

Updates the button to display the image for the given class.

**Parameters:**

`itemClass` - the new item type to display

### empty

```
public void empty()
```

Resets the ItemCell to an empty cell.

### isEmpty

```
public boolean isEmpty()
```

Returns true if this cell is not filled with an item.

### setItemClass

```
public void setItemClass(java.lang.Class itemClass)
```

Sets the item class held by the cell.

### getItemClass

```
public java.lang.Class getItemClass()
```

Returns the item class held by the cell.

### getButton

```
public com.badlogic.gdx.scenes.scene2d.ui.ImageTextButton getButton()
```

Returns the button which displays the item held by this cell.

### II.3.2.12 *Pool* Hierarchy

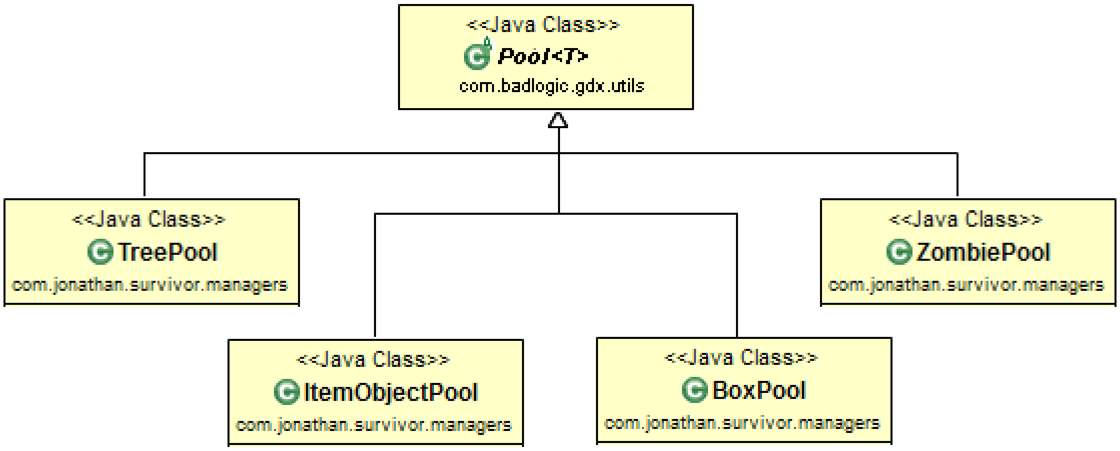


**Figure 95 : Pool class hierarchy**

### II.3.2.13 *TerrainLayer* Enumerations

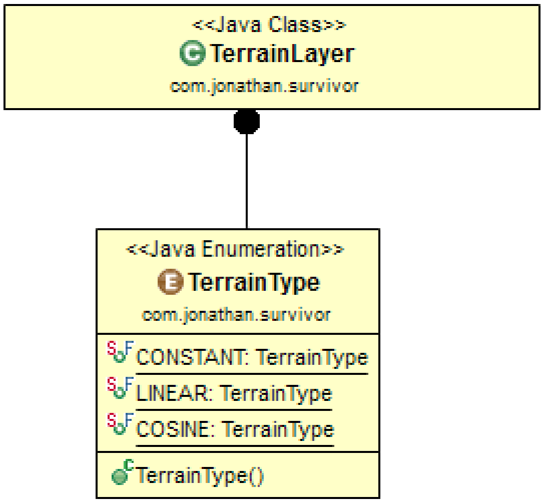


**Figure 96 : TerrainLayer related enumerations**

### II.3.2.14 *Level* Hierarchy

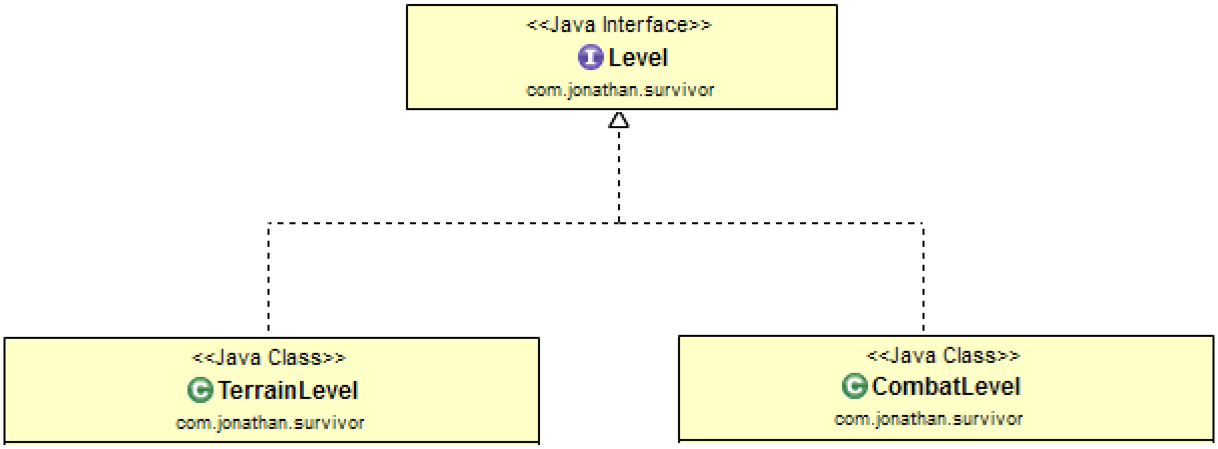


**Figure 97 : Level class hierarchy**

### II.3.2.15 *Weapon* Hierarchy

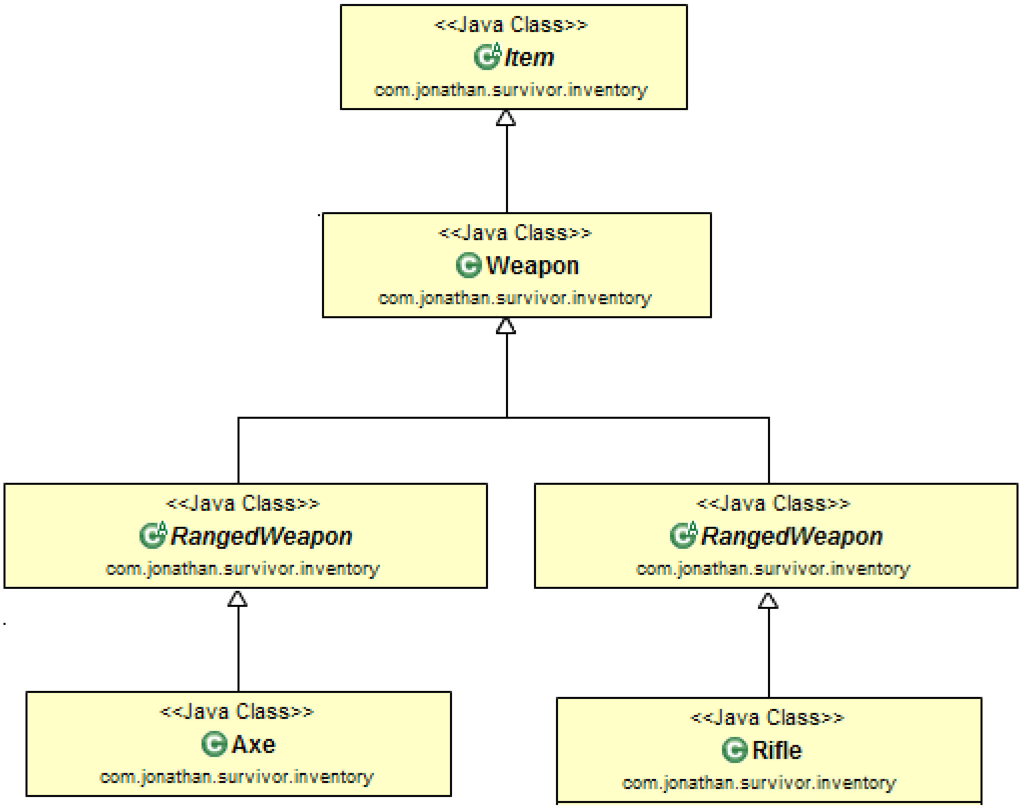

<<Java Class>>
Item
com.jonathan.survivor.inventory
<<Java Class>>
Weapon
com.jonathan.survivor.inventory
<<Java Class>>
RangedWeapon
com.jonathan.survivor.inventory
<<Java Class>>
RangedWeapon
com.jonathan.survivor.inventory
<<Java Class>>
Axe
com.jonathan.survivor.inventory
<<Java Class>>
Rifle
com.jonathan.survivor.inventory

**Figure 98 : Weapon class hierarchy**

### II.3.2.16 *Craftable* Hierarchy (a)

(Note: Hierarchy split into two due to lack of space)

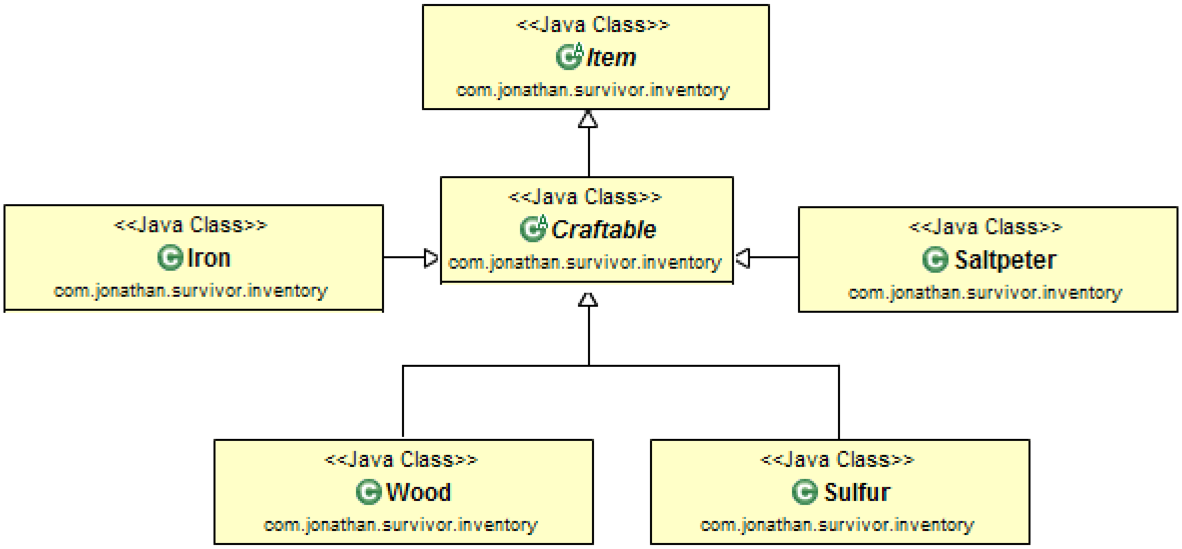

<<Java Class>>
Item
com.jonathan.survivor.inventory
<<Java Class>>
Iron
com.jonathan.survivor.inventory
<<Java Class>>
Craftable
com.jonathan.survivor.inventory
<<Java Class>>
Saltpeter
com.jonathan.survivor.inventory
<<Java Class>>
Wood
com.jonathan.survivor.inventory
<<Java Class>>
Sulfur
com.jonathan.survivor.inventory

**Figure 99 : Craftable class hierarchy (a)**

### II.3.2.17 *Craftable* Hierarchy (b)

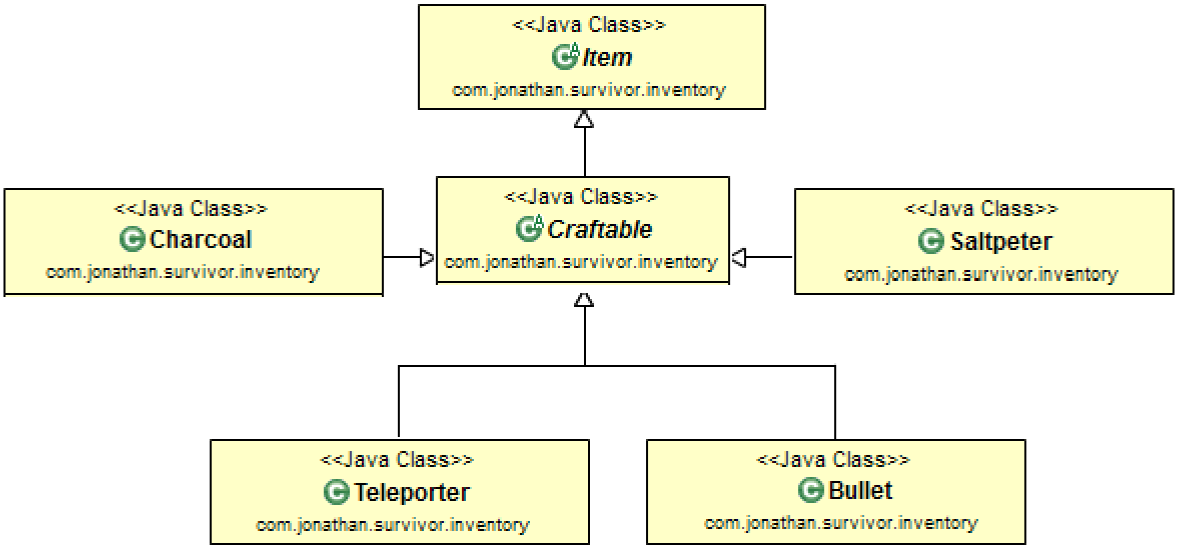

<<Java Class>>
Item
com.jonathan.survivor.inventory
<<Java Class>>
Charcoal
com.jonathan.survivor.inventory
<<Java Class>>
Craftable
com.jonathan.survivor.inventory
<<Java Class>>
Saltpeter
com.jonathan.survivor.inventory
<<Java Class>>
Teleporter
com.jonathan.survivor.inventory
<<Java Class>>
Bullet
com.jonathan.survivor.inventory

**Figure 100 : Craftable class hierarchy (b)**

### II.3.2.18 *CraftingManager* Inner Classes

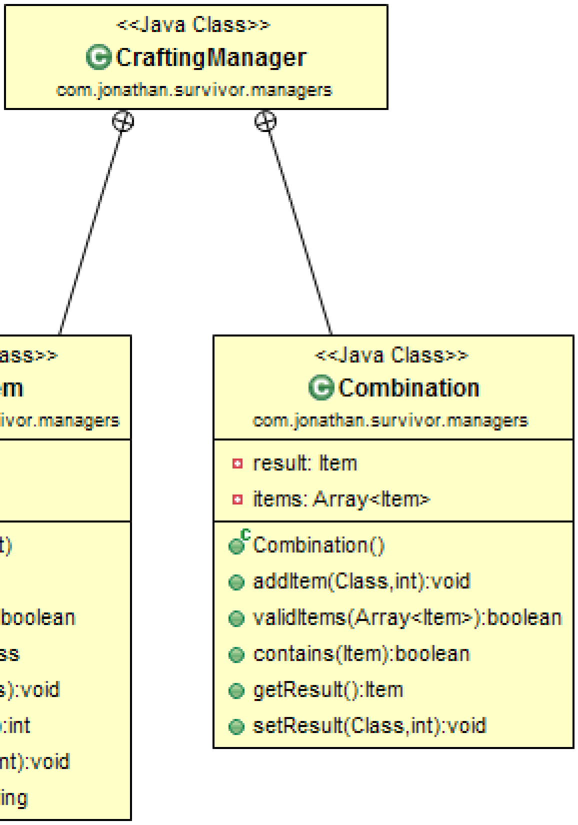


**Figure 101 : CraftingManager inner classes (b)**

```
private class CraftingManager.Combination
extends java.lang.Object
```

Describes a combination of items which, together, craft an item.

## Field Detail

### result

```
private CraftingManager.Item result
```

Stores the item which results from the combination.

### items

```
private com.badlogic.gdx.utils.Array<CraftingManager.Item> items
```

The array of items which must be combined to form the above the resulting item.

## Constructor Detail

### CraftingManager.Combination

```
public CraftingManager.Combination()
```

Creates a new, empty combination.

## Method Detail

### addItem

```
public void addItem(java.lang.Class item, int quantity)
```

Adds a specific item to the combination.

**Parameters:**

`item` - the item to add to the combination

`quantity` - the quantity of that item required to satisfy the combination

### validItems

```
public boolean validItems(com.badlogic.gdx.utils.Array<CraftingManager.Item> givenItems)
```

Tests if the given list of items corresponds to the items needed in this combination. Returns true if the given list of items form this combination.

**Parameters:**

`givenItems` - a list of items tested to match the combination

**Returns:**

true, if the given list of items match the combination.

### contains

```
public boolean contains(CraftingManager.Item item)
```

Returns true if this combination contains this item, which is needed to form this combination.

**Parameters:**

`item` - the item to be tested to be contained in the combination

**Returns:**

true, if the item is contained in the combination

### getResult

```
public CraftingManager.Item getResult()
```

Sets the resulting item from the crafting combination

### setResult

```
public void setResult(java.lang.Class item, int quantity)
```

Gets the resulting item from the crafting combination.

**Parameters:**

`item` - the item which will be crafted by the combination

`quantity` - the quantity of the item that will be crafted.

```
public class CraftingManager.Item
extends java.lang.Object
```

Pairs an item with a specific quantity for use inside a Combination.

## Field Detail

### quantity

```
private int quantity
```

Stores the quantity of the item needed in a combination

### item

```
private java.lang.Class item
```

Holds the item needed in a combination

## Constructor Detail

### CraftingManager.Item

```
public CraftingManager.Item(java.lang.Class item, int quantity)
```

Creates a pair between an item and a specific quantity.

**Parameters:**

`item` - the item type

`quantity` - the quantity of the item

## Method Detail

### add

```
public void add(int quantity)
```

Adds the given quantity to this item instance.

**Parameters:**

`quantity` - the quantity to add

### equals

```
public boolean equals(CraftingManager.Item other)
```

Returns true if the given item is equal to this item.

**Parameters:**

`other` - the other item to test equivalence with

**Returns:**

true, if the given item is the same in type and quantity as this item.

### getItem

```
public java.lang.Class getItem()
```

Gets the item class held by this instance.

### setItem

```
public void setItem(java.lang.Class item)
```

Sets the item class held by this instance.

### getQuantity

```
public int getQuantity()
```

Gets the quantity of the item.

### setQuantity

```
public void setQuantity(int quantity)
```

Sets the quantity of the item.

### toString

```
public java.lang.String toString()
```

**Overrides:**

`toString` in class `java.lang.Object`

## II.3.2.19 *ItemManager* Inner Classes

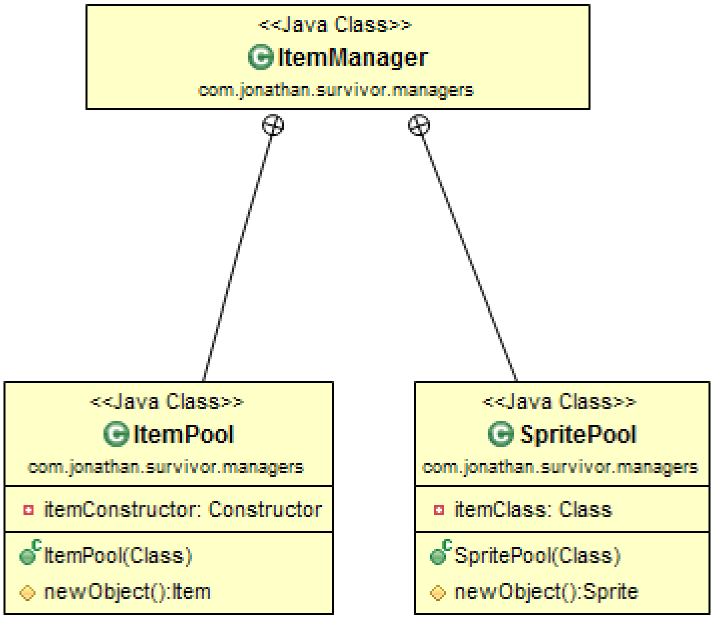


**Figure 97 : ItemManager inner classes**

```
class ItemManager.ItemPool
extends com.badlogic.gdx.utils.Pool<Item>
```

Stores a pool of an *Item* subclass. The *Item* subclass stored in this pool is determined by the argument passed into the constructor of this class.

### Field Detail

#### itemConstructor

```
private com.badlogic.gdx.utils.reflect.Constructor itemConstructor
```

Stores the constructor used to create instances of the item in the pool.

### Constructor Detail

#### ItemManager.ItemPool

```
public ItemManager.ItemPool(java.lang.Class itemClass)
```

Creates a pool of items for the given class.

**Parameters:**

`itemClass` - the type of item to pool

### Method Detail

#### newObject

```
protected Item newObject()
```

Returns a new instance of the Item when none are free in the pool.

**Specified by:**

`newObject` in class `com.badlogic.gdx.utils.Pool<Item>`

**Returns:**

an item instance of the class denoted by the pool

```
class ItemManager.SpritePool
extends
com.badlogic.gdx.utils.Pool<com.badlogic.gdx.graphics.g2d.Sprite>
```

Stores a pool of sprites for each item. The sprite displays the item passed as an argument to the constructor.

## Field Detail

### itemClass

```
private java.lang.Class itemClass
```

Stores an item subclass. This is the item for which sprites will be produced. These sprites represent the inventory sprites displayed for this item.

## Constructor Detail

### ItemManager.SpritePool

```
public ItemManager.SpritePool(java.lang.Class itemClass)
```

Creates a pool of items for the given class.

**Parameters:**

`itemClass` - the item class for which to generate sprites for

## Method Detail

### newObject

```
protected com.badlogic.gdx.graphics.g2d.Sprite newObject()
```

Returns a new inventory sprite for the item when none are already free in the pool.

**Specified by:**

`newObject` in class `com.badlogic.gdx.utils.Pool<com.badlogic.gdx.graphics.g2d.Sprite>`

**Returns:**

a new Sprite instance for the item class represented by the pool

II.3.2,20 *GestureManager* Hierarchy

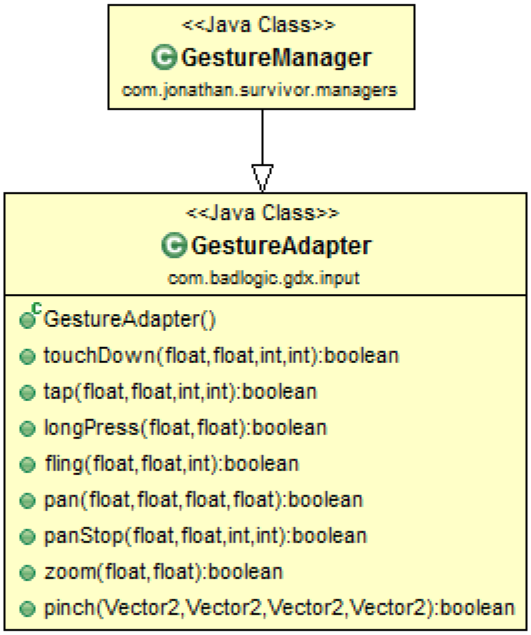

<<Java Class>>
GestureManager
com.jonathan.survivor.managers
<<Java Class>>
GestureAdapter
com.badlogic.gdx.input
GestureAdapter()
touchDown(float,float,int,int):boolean
tap(float,float,int,int):boolean
longPress(float,float):boolean
fling(float,float,int):boolean
pan(float,float,float,float):boolean
panStop(float,float,int,int):boolean
zoom(float,float):boolean
pinch(Vector2,Vector2,Vector2,Vector2):boolean

**Figure 97 : GestureManager class hierarchy**

Note: Given that the *GestureAdapter* is part of the *LibGDX* library, its methods and data fields are not given any explanations. For such an explanation, refer to *LibGDX's* documentation.

### II.3.2.21 *InputManager* Hierarchy

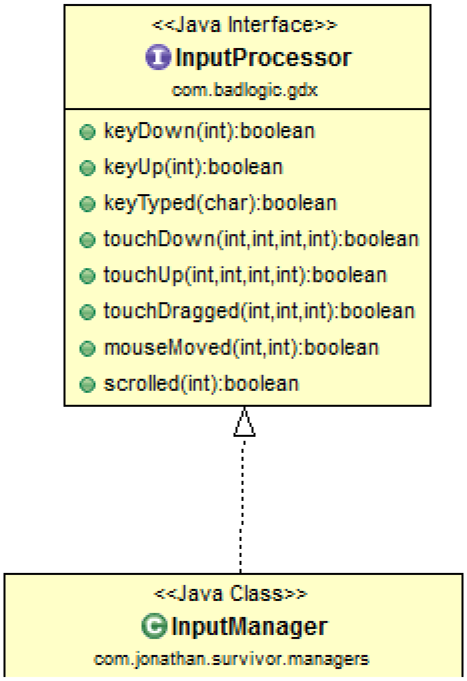

<<Java Interface>>
InputProcessor
com.badlogic.gdx
keyDown(int):boolean
keyUp(int):boolean
keyTyped(char):boolean
touchDown(int,int,int,int):boolean
touchUp(int,int,int,int):boolean
touchDragged(int,int,int):boolean
mouseMoved(int,int):boolean
scrolled(int):boolean
<<Java Class>>
InputManager
com.jonathan.survivor.managers

**Figure 97 : InputManager class hierarchy**

Note: Given that the *InputProcessor* is part of the *LibGDX* library, its methods and data fields are not given any explanations. For such an explanation, refer to *LibGDX's* documentation.

## III. Methods of Evaluation

To evaluate our project, we let other people play our game. For instance, we allowed our instructor play the game, and, in turn, we extracted valuable feedback which allowed us to implement extra features, or change already-existing ones in order to enhance the quality of our application.

Below this is a list of the individuals who tested the game, along with their feedback (i.e., the results of our evaluation) :

| **Tester** | **Feedback (Evaluation Results)** |
|---|---|
| Amin Ranj Bar | • Add music and sound to the finished product<br>• Add a game select screen, which allows the player to either choose, load, or create a new world |
| Ho Man Chan | • Make the combat feel faster |
| Adrian Gammon | • Make the hit boxes on the items larger, so that users have less difficulty in tapping on items to pick them up |
| Olivier Carrière | • Make the aiming mechanic for the rifle faster, in order to avoid boring the user as he charges his ranged weapon |
| Jason Ege | • Make the backpack button bigger |
| Julian Lucuix-André | • Add a *Save* button in the pause menu |
| Jacky Ma | • Don't make the game save when you quit (implement a save button) |
| Franky Tam | • Make the player walk faster<br>• Make the zombie walk slower |

| | |
|---|---|
| Frederic Lam | • Make the pause button bigger |
| Bradley Arsenault | • Add a list of recipes inside the survival guide<br>• Don't make the player lose all his items when he dies from a zombie |
| Trevor Fernandez | • Make the zombie run faster |
| Wai Lun Lau | • Put a yellow exclamation point on top of the zombie's head when he sees you |
| Kevin Tran | • Put the entries inside the survival guide inside a scroll pane in order to be able to add more entries |
| Olivier Zephir | • Make the zombie's charge attack faster |
| Maxime Nguyen | • Make the earthquake do less damage |

**Table 1: Testing and feedback results**

As a result, all of the feedback provided for the project was taken into great consideration. For this reason, all of the suggestions given above were implemented into the final product. Although some of the feedback required additional features to be implemented, we nevertheless decided to include them for the sake of delivering a polished finished product.

The following sections will include further methods we used to evaluate our project, along with the results of said evaluations. They were omitted from this section to avoid repetition.

# IV. Results: System Quality

## IV.1 Developer Perception

The strength of the system we are most proud is the optimizations that were made with the code, and the overall design of our algorithms. Among these algorithms are procedural generation. In fact, programmatically, the forest in *Free the Bob* is built using a “terrain level”. This level consists of a fixed-size matrix, where every element consists of a “terrain layer”. In turn, each layer is represented as either a constant, linear, or cosine function which the player can walk on. From a programmatic perspective, the player is always at the center of the terrain level. When the player moves from one layer to another, the level adjusts its matrix and moves the layers so that the player always resides in the center-most layer. Once the layers are re-positioned inside the matrix, their row and columns relative to the world also change. Using the layer’s cell coordinates, along with a random number generator, the layer’s terrain is built. As explained in *Section II.1 Algorithms*, this algorithm is very efficient.In fact, the operation of defining an elementary function for a *TerrainLayer* takes *O(1)* time. For this reason, our procedural terrain generation algorithm runs very efficiently, and we are thus very proud of it.

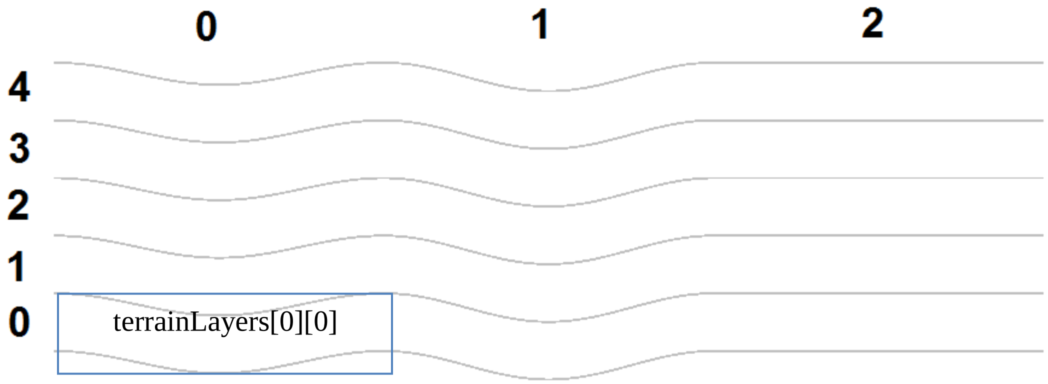


**Figure 102: Visual representation of the *TerrainLayer* matrix**

Conversely, to place objects on the layer, a different random number generator seed is computed. To calculate said seed, the terrain layer’s row, column are added. To this sum is added the world seed. The resulting number is called the object seed. Since every layer has unique cell coordinate in the world, each layer has a unique seed. This object-placement algorithm takes

*O(n)* time, as explained in *Section II.1 Algorithms*. For this reason, the procedural generation is able to run efficiently. Thus, we are proud of our coding on this front.

Moreover, the game contains the *World* class, which holds a reference to the terrain level, along with the *GameObjects* contained in the game. In addition, it controls all game logic. Thus, the class can be seen as a container or a master class for most of the game's entities and gameplay mechanics. This *World* class allowed us to use the *Model-View-Controller (MVC)* pattern. In fact, the world represents a model. In turn, the renderers represent the view, and the *Input/GestureManager* represents the controller. Thus, given that we successfully implemented a design pattern which we did not cover in class, we are satisfied with the quality of our system.

Furthermore, our input handling algorithms were optimized to avoid causing framerate spikes whenever the user taps on the screen. As everything else in the project, delivering a user-friendly experience was at the forefront of all our coding decisions. For instance, to process input, every finger tap is first delegated to the *InputManager* class. Then, the *InputManager*, which holds an instance of the camera used to render the world, can convert the given tap coordinates into world coordinates. The position of the tap is then delegated to the *World,* which can subsequently decide how to process the touch. This operation takes *O(n)* time, as explained in *Section II.1 Algorithms*. Thus, judging from the efficiency of our algorithms, we could state that the quality of our project was very strong.

A technique called camera culling was implemented to optimize object rendering. This is another algorithmic strength of the project that we like. If an object happens to be outside the camera, it is not rendered by *OpenGL*. If it is viewable by the camera, it is drawn.

We also like the coding methods we used to manage the in-game GUIs. In fact, to render a heads-up display for the game, several subclasses are derived from the *Hud* class. This class holds a *Stage*, a *LibGDX* object used to draw 2d widgets to the screen. Each *Hud* subclass holds buttons and widgets that will be drawn to the screen using this stage. Using LibGDX's scene2d library allowed us to create responsive and user-friendly GUIs.

Furthermore, we are happy with the way the GUIs were designed, and thus like the layout of our program. In fact, all of the game's menus were optimized to make the user experience feel approachable on a mobile phone. Thus, we ensured that every button was large, and that every button was easily visible and distinguishable from all the others. We even went as far as to implement the *GameSelectScreen* as per the teacher's feedback, which allowed us to greatly improve user comprehension of the graphical user interface and of the game's functionality.

However, one of the shortcomings of our project at the beginning was the *WorldSelectScreen*. In fact, players were confused as they looked at the screen. They had a hard time understanding the concept of profiles. However, upon showing the program to the instructor, he gave us the feedback to add a *GameSelectScreen*, wherein the user would have an option to either load, continue, or create a new profile. Thus, by implementing this new screen, we removed the weakness of our layout and turned it into a strength. Therefore the reasons stated above, we are proud of our coding and the layout of our system.

## IV.2 Objective Measure

In order to measure the quality of our sytem, we utilized a frame rate counter. In fact, we logged the frame rate of our application on several different devices. Using the data collected from these tests, we were able to measure quality of our system, and make necessary adjustments. In point of fact, we changed certain weaknesses in our code in order to optimize the game's performance.

Furthermore, as detailed in *Section III Methods of Evaluation*, we let others play the game and collected their feedback. This allowed us to ensure that we met our objectives for the project. In turn, we created a table which collected all of the suggestions given from the people who tested it. Subsequently, we made necessary adjustments and improvements to our code, optimizing the user experience to make the game feel as fun and responsive as possible.

In addition, in order to test the quality of our project, we tested the game on low-end Android devices. In fact, we chose to test it on lower-end phones in order to ensure that it would run smoothly on every other Android device. If we were to test it on high-end Android devices, the game would run smoothly, and we would not be able to determine whether our program is optimized for every phone on the market.

Finally, in order to test our project's quality, we tested the game on different-sized phones and tablets. In turn, we were able to deduce whether or not the game would be able to run on every size phone.

## IV.3 Developer Evaluation Sheets

As stated in the previous section, we used a frame rate counter to test our project's quality, and recorded the average frame rate for the game on several Android devices. We did this near the end of the project, where we had to ensure that the quality of our project was as strong as possible.

The first time we tested the game and logged its frame rate, we received the following results:

| **Device** | **Android Version** | **Average Frame Rate (frames/second)** |
|---|---|---|
| HTC Desire Z | 2.3.4 | 45.2 |
| Kindle Fire | 2.3.6 | 47.1 |
| Samsung Galaxy SII | 4.2.1 | 58.7 |

**Table 2: Trial one: Frame rate tests on different Android devices**

As seen in the table above, the game runs very smoothly on the Samsung Galaxy SII, but less well on the HTC Desire Z and the Kindle Fire. In fact, the optimal frame rate for a game is 60 frames per second. Furthermore, there seemed to be a somewhat linear relationship between the Android version and the average frame rate.

Considering the results we gathered, we needed to find ways in which to optimize the game for lower-end Android devices. Therefore, upon conducting online research, we found that, in order to optimize image rendering, we needed to store images inside *TextureAtlases.* This makes it so that several images are grouped into a larger image, reducing the number of draw calls due to GPU texture bindings. After placing our images into atlases, the following performances were observed.

| **Device** | **Android Version** | **Average Frame Rate (frames/second)** |
|---|---|---|
| HTC Desire Z | 2.3.4 | 52.2 |
| Kindle Fire | 2.3.6 | 53.1 |
| Samsung Galaxy SII | 4.2.1 | 58.8 |

**Table 3: Trial two: Frame rate tests on different Android devices**

As seen with the evaluation sheet above, placing images in atlases highly optimized the performance of our program. In fact, the HTC Desire Z gained a significant 7.0 increase in frame rate. With this improvement, our program almost reached optimum performance. In truth, a 52.2 frame rate on a low-end device such as the aforementioned HTC phone is highly satisfactory.

However, we wanted to increase the program's performance even further. Therefore, we decided to implement different asset loading techniques. For instance, even when the player was in the game, the main menu images were still loaded in memory. This was very poor memory management, given the low amount of RAM on most low-end phones. Therefore, we added the optimization which consisted of disposing of the main menu assets whenever the user left the main menu. Subsequent to this optimization, the following performances were observed.

| **Device** | **Android Version** | **Average Frame Rate (frames/second)** |
|---|---|---|
| HTC Desire Z | 2.3.4 | 54.2 |
| Kindle Fire | 2.3.6 | 55.1 |
| Samsung Galaxy SII | 4.2.1 | 58.8 |

**Table 4: Trial three: Frame rate tests on different Android devices**

Although the higher-end Galaxy SII did not gain much of a performance boost, the HTC Desire Z and the Kindle Fire both gained a 1.0 frame rate increase. Given that these frame rates were solid, and that the program was nearing its deadline, we decided to stay with these optimizations, and move on to bug-fixing.

# V. Project Management

## V.1 Timeline

| **Task** | **Planned date** | **Actual date** | **Assigned Person** | **Notes** |
|---|---|---|---|---|
| Brainstorm Project Ideas, Create Game's Story | Week 1 | Week 1 | Jonathan | Completed within expected time |
| Create Timeline, Features List, Design UML | Week 2 | Week 2 | Team | Started preparing for presentation |
| Create PowerPoint Presentation, Create Design & GUI Portion of Proposal Document | Week 3 | Week 3 | Jonathan | Extra drawings had to be made for presentation |
| Randomly-Generated Terrain | Week 4 | Week 5 | Team | Delayed by one week due to exams. |
| Profiles & Saving Data to the Hard Drive | Week 5 | Week 5 | Jonathan | Profile saving also modified on week 14 |
| Character Movement (Jumping, Falling &Walking), Exploration GUI | Week 6 | Week 6 | Jonathan | Modified gravity from 9.8m/s$^2$ to custom value to ensure game felt fun |
| Scavenging (Trees & Boxes) | Week 7 | Week 6 | Team | Completed earlier than expected since we both worked on it |
| Item Dropping, Inventory & Crafting GUI, Splash Screen | Week 8 | Week 8 | Mugisha | Research had to be made on chemical formulas |
| Loading Screen, World Select Menu, Main Menu | Week 9 | Week 9 | Jonathan | Also added *GameSelectScreen* on week 14 |

| | | | | |
|---|---|---|---|---|
| Weapons (Melee & Ranged), Polished Animations | Week 10 | Week 10 | Jonathan | Received good feedback on animations |
| Zombies, Zombie AI (Walking), Tutorial | Week 11 | Week 11 | Mugisha | Decided to include tutorial inside survival guide instead of in cutscene |
| Zombie AI (Combat), Player Combat (Ranged Weapons) | Week 12 | Week 12 | Jonathan | Zombie combat modified to be faster on week 14 |
| Player Combat (Melee Weapons), Combat GUI | Week 13 | Week 13 | Mugisha | Modified appearance of Combat GUI on week 13 |
| Survival Guide GUI, Crafting System, Teleporter | Week 14 | Week 14 | Jonathan | Also performed play-testing sessions |
| Preparation of Presentation, Optional Features: Eating Mechanic, Player Menu GUI & Eating System | Week 15 | Week 15 | Team | Optional features replaced by audio playback and *GameSelectScreen* |
| Writing Design Document | Week 16 | Week 16 | Team | UML had to be drastically changed |

**Table 5: Timeline**

# VI. Conclusion

In closing, we were very successful in completing our assumed tasks. In fact, we developed a game which is able to test the player's knowledge in chemistry. Furthermore, we were able to build a procedurally generated world using elementary mathematical functions, testing our own knowledge in the field of mathematics.

Additionally, we were successful in completing our tasks, since every mandatory feature that was a part of the project's proposal document was implemented. Not only that, but all of our features are fully functional, delivering on the promises which we initially set out to accomplish. What is more, we scheduled our time highly efficiently, allowing us to let other play-test the game and receive feedback from fellow gamers. From these suggestions, we implemented additional features which looked to improve the user's experience when playing the game. Among these features are the inclusion of audio, a game selection screen, and enlarged item hit boxes. These additions are also fully-functional, allowing the final product to meet a high degree of quality. The project thus utilizes a myriad of different techniques we have learned throughout our CEGEP courses, such as the use of the random number generators, the creation of compounds using chemical items, and the graphing of elementary functions.

| **Feature** | **Completion Status** |
|---|---|
| Randomly Generated World | Completed |
| Character Movement | Completed |
| Survival Guide | Completed |
| Profiles | Completed |
| Zombies | Completed |
| Combat | Completed |
| Scavenging | Completed |

| | |
|---|---|
| Items | Completed |
| Transparently-Tinted GameObjects | Completed |
| Weapons | Completed |
| Android Compatibility | Completed |
| LibGDX Framework | Completed |
| Multiple Screen Size Support | Completed |
| Crisp Graphics | Completed |
| Polished Animations | Completed |
| Tutorial | Completed |
| PC Combatibility | Completed |
| Optional Features: Eating | Uncompleted |
| *Added: GameSelectScreen* | Completed |
| *Added: Sound & Music* | Completed |

**Table 6: Feature completion statuses**

In terms of the procedurally generated world, we were very successful in completing our tasks. In fact, there was a large amount of work required to make it function properly. For instance, the concept of *TerrainLayers* had to be implemented, and the corresponding *TerrainLevel* class had to be created in order to hold a reference to a matrix of layers. What proved to be most difficult was optimizing the procedural generation with object pooling. In fact, creating the world in an efficient way was very difficult. However, given that our target platforms was Android devices, we decided to optimize the process as much as possible, in order to avoid framerate dips on lower-end devices. However, this was a learning experience for us, as it allowed us to test our abilities to optimize code. In the end, given that the game runs at a serviceable framerate, even on lower-end devices, we are happy with the amount of work we put into the optimization of our code. Therefore, this was a learning experience, as it taught us the concept of object pooling, along with the mere fact that optimizing your code can prove to be highly beneficial in the long term.

On a separate note, the implementation of the character movement was slightly less difficult. In fact, using our concept of *TerrainLayers*, the major advantage was that each layer represents an elementary mathematical function. Thus, in order to allow the user to walk across the world, we simply apply an x-velocity to the player *GameObject*. Then, given the x-position of the player, the corresponding y-position can be found simply by inputting the x-position into the layer's function. The same concept is utilized for zombie movement.

This was also a learning experience for us, as it taught us that implementing every feature in an intelligent manner saves time in the long run. For instance, thanks to our natural implementation of the *TerrainLayer* and its corresponding elementary functions, we were able to easily implement character movement, without the need for coding workarounds. Furthermore, this feature was not large in terms of quantity, since the algorithm is simple. The same can be said with regards to the jumping mechanic. All that was needed was to add a y-velocity to the player, apply gravity, make him switch layers, and place him at the right y-position once he landed on his new layer. Once again, this taught us that intelligent programming saves time in the long term.

Next, the Survival Guide feature was rather quick to implement. In fact, the volume of tasks needed to complete such a feature was not very large. In point of truth, given that we had implemented a *HUD* superclass, which allowed us to easily create user interfaces when the user was in-game, we were able to use the *scene2d* framework's GUI widgets in order to place a list of entries in a scrollpane. Further, when the user presses on one of these entries, the list is swapped with a label which displays the contents of the entry. Thus, it was not difficult to implement, as we utilized the pre-existing *scene2d* widgets. Further, this was a learning experience for us, as it forced us to create the *HUD* superclass in order to easily create different

GUIs for the player to interact with. As before, this smart implementation taught us that efficient programming can effectively save time when enlarging the scope of the program.

The implementation of profiles was more difficult. In fact, it was a learning experience for us, as it forced us to use the JSON file format, which we had never seen before. However, given that said file format felt intuitive, it was less difficult than expected to implement. In fact, what we did was we created a *Profile* class, which acted as a data container for each of the player's profiles. When needed, the profiles were saved to the hard drive, and loaded by the *ProfileManager* whenever the game restarted. Therefore, we learned that creating *Manager* classes allowed greater flexibility in our coding, and effectively avoided copy-pasting code that could be implemented once inside a class. Furthermore, the biggest difficulty we encountered was parsing *HashMaps* into the JSON file format. In fact, *LibGDX* did not provide a pre-built way to parse and read *HashMaps*, and we thus had to resort to converting the map into a string of text, and parsing it manually into the JSON file. Thus, the quantity of work required was rather large, as we needed to resort to workarounds in our code for everything to function properly. However, this was a learning experience, because it taught us how to find workarounds when a library presented certain limitations.

Next, in terms of the zombies, the quantity of work required was large. First, the art for the zombie, along with his animations had to be created. Then, we had to import the zombie into the game, and control his walking patterns using artificial intelligence. Moreover, we then had to implement a vision system, which allowed the zombie to see the player when he came too close. To solve these problems, we created an external helper class named *ZombieManager*, which allowed us to separate the zombie game logic from the game logic of every other class. Thus, this allowed our *World* class to become slightly less jumbled, effectively avoiding *spaghetti-code*. For this reason, our implementation of the zombies was a learning experience, as it taught us that separating code into separate classes could avoid over-complicating our program. However, given that our implementation of the zombie movements was rather simple, the only difficult part was animating the zombie, and making him see the player when he came too close. However, this was solved by simply comparing the zombie's distance with that of the player, determining whether or not the player was close enough for the zombie to follow him.

Next, our implementation of combat was less time-consuming than expected. For instance, to create a separate playing field for the combat, we simply implemented the *CombatLevel*, which allowed the player and the zombie to be placed on a horizontal playing field. Furthermore, this class defined the line which denoted the bounds of the level. Next, to enter combat, the player's bounding box was simply checked against that of a zombie. If both the player and the zombie intersected, the *KoAnimation* would play, and the *World* would switch to the *CombatLevel.* It is important to note that the *CombatLevel* extended the *Level* superclass. This was crucial to our implementation. In fact, by creating a *Level* superclass, it allowed us to easily switch between the forest and the combat, without creating any larger code paths. Thus, this taught us that the concept of polymorphism and inheritence is highly useful when attempting to minimize the amount of code paths. Furthermore, thanks to concept of *HUDs,* as explained above, we were able to use *scene2d* widgets to display the jump, melee and fire buttons. For this reason, the implementation of the combat was not difficult. Furthermore, as explained in *Section II.1 Algorithms*, implementing the zombie's artificial intelligence during combat was simple. In fact, all that needed to be done was to switch the state of the zombie and the player accordingly, dealing damage when the zombie or the player hits each other.

Next, the scavenging feature was more difficult to implement. In fact, what we did was include *HashMap* for every object which could drop items. Each *HashMap* had a key corresponding to an *Item* subclass, and a value corresponding to a float between 0 and 1.0. If the value of a certain item was 1.0, the item had a one-hundred percent chance of spawning once the object is scavenged. If the value was 0.5, the item has a 50% chance of spawning. Note that the spawning is done inside the *World* class, whose *spawnItems()* method is called whenever an object is scavenged. In turn, the aformentioned *HashMap* for that object is taken, and are randomly generated accordingly. The *ItemObjects* are shot out using the *ItemObject.spawn()* method, which gives each object a certain velocity when spawned in order to create a confetti effect. Thus, the implementation of the scavenging was not difficult, and there was not a large amount of necessary work needed to implement it. However, coming up with the actual algorithm was far more difficult. It turned out to be a learning process, however, as we were forced to learn how to use *HashMaps.*

Furthermore, the item feature was slightly more difficult. In fact, throughout the program, in order to optimize the creation and destruction of objects, we used the *Class* instance of each item in order to denote an item type. For instance, instead of creating a new *Wood* instance whenever a new *Wood* item needs to be spawned, the item was simply refered to by its class (i.e., *Wood.class).* However, if the information for a particular item needed to be retrieved, the *ItemManager* created and pooled *Wood* instances, and passed them out to interested parties. On a separate note, while the actual *Item* class denotes a data container for each item, the *ItemObject* class represents the physical representation of an item dropped in the world. This allowed us to separate the model from the view when discussing the *MVC (Model-View-Controller)* pattern. Thus, the implementation of items was time-consuming and difficult. In fact, we had to create data containers for each different item, along with sprites for each one. In turn, we also had to implement pools for each *Item* type, along with their corresponding sprites. Thus, although the algorithm used was not complex, the overall design was time-consuming due to the shear amount of items that the player can collect. However, this was a learning experience, as it taught us how to manage an inventory containing a large amount of items, which could ultimately prove useful in many different types of applications.

Next, the transparently-tinted *GameObjects* were far easier to implement. In fact, when the each *GameObject* was drawn, its *terrainCell* was taken, and compared to the *TerrainLevel*. If the *GameObject* was on the same row as the player, then it retained its normal colour. Conversely, if the *GameObject* was on a different row, it was coloured dark-gray. This allows the user to know which objects he can interact with. Thus, given that we used to the *Spine* runtime library, we were able to switch the colour of the *GameObjects* by changing the colour of their skeletons. For instance, in pseudo-code, we set the colour as follows:

```
skeleton.setColor(Color.DARK_GRAY);
```

Thus, given that the code needed to colour the *GameObjects* was already implemented, it was not time-consuming to implement the feature. In fact, it was rather easy. Also, given that we had given each *GameObject* a *terrainCell*, in order to know to which *TerrainLayer* it belongs, it was easy to determine when a *GameObject* should have been coloured transparent or not. Thus, this was a learning experience for us. In truth, we learned that planning ahead in terms of data fields proves to be beneficial in the long term.

Moreover, the *Weapons* feature was slightly more difficult to create. In fact, given that we already created *Item* subclasses for the weapons, we needed to find a way to give them functionality. Therefore, we created the *Weapon, RangedWeapon* and *MeleeWeapon* subclasses. In turn, whenever an item extends *Weapon*, it can be equipped by the player through his *Loadout* class. Further, all weapons have damage floating points, which allows the user to deal damage to zombies. Furthermore, the *RangedWeapon* also has a *chargeTime* float, which determines how much time it will take to charge the weapon. On a different note, in order to visually display the weapons on the player, we placed images on the player.  Whenever he was using a certain weapon, that image was activated, and thus displayed. This was a difficult feature to implement, as it required to set up extra bones and images on the player in the animation engine. For this reason, the quantity of work required was large, as it required us to create animations for each weapon. However, in the long term, it was worth it, as it allowed the player to engage in combat with a zombie. It was also a learning experience, as it taught us how to create weapons and loadouts for the player, a feature we will surely implement in future applications.

On a separate note, *Android Compatibility* was not difficult to implement. In fact, simply by creating a *LibGDX* project, we automatically had the option of deploying the application to Andoird devices. Thus, it was a learning experience, as it taught us that creating Android applications was less difficult than it seemed. Moreover, the quantity of work required was also small, as the creation of a project was enough to create Android compatibility. The same can be said about the *LibGDX Framework* feature. In fact, all we had to do was download the *LibGDX* application, and create a project using this framework. Granted, we did have to learn an abundance of new classes. In this way, it was a learning process, as we were forced to often look at the framework's documentation. Thus, it required a large quantity of work, even though it wasn't very long to do. Granted, traversing the documentation is a skill that will help a great deal us in the future.

Overall, we were very happy with the quality of all our features. In fact, all of them were tested to be working. Furthermore, during the testing phase, most testers found the features to be engaging and thoughtful. In reality, as explained in *Section IV.1 Developer Perception,* we are very satisfied with the quality of our project.

Thus, in closing, with the knowledge collected from prior courses, we have delivered a game that both teaches and entertains, creating a finished product that can be published to gamers' smartphones, and that can potentially intrigue the likes of many Android users.