**Table of Contents**

[**Overview** 3](#_Toc474920643)

[**Using SkyEngine** 4](#_Toc474920644)

[Starting a Project 4](#_Toc474920645)

[Using the Game Loop 5](#_Toc474920646)

[Getting User Keyboard Input 5](#_Toc474920647)

[Rendering Quads to the Screen 6](#_Toc474920648)

[Understanding and Using Delta Time 6](#_Toc474920649)

[Getting User Mouse Input 7](#_Toc474920650)

[Using Game States 8](#_Toc474920651)

[Playing Audio 8](#_Toc474920652)

[Rendering with Textures 10](#_Toc474920653)

[Rendering Text 12](#_Toc474920654)

[Using VSync 13](#_Toc474920655)

[Using Antialiasing 14](#_Toc474920656)

[Using Buttons 15](#_Toc474920657)

[Using Checkboxes 16](#_Toc474920658)

[Using Radio Buttons 17](#_Toc474920659)

[Creating Items 19](#_Toc474920660)

[Using Static Inventories 19](#_Toc474920661)

[Using Dynamic Inventories 23](#_Toc474920662)

[**Sample Project - Pong** 26](#_Toc474920663)

[**Summary of Methods** 35](#_Toc474920664)

[skyEngine.Core 35](#_Toc474920665)

[AbstractGame 35](#_Toc474920666)

[GameContainer 35](#_Toc474920667)

[Window 37](#_Toc474920668)

[Renderer 38](#_Toc474920669)

[Input 40](#_Toc474920670)

[SoundClip 41](#_Toc474920671)

[SkyFont 41](#_Toc474920672)

[skyEngine.RenderingUtilities 43](#_Toc474920673)

[AntiAliasing 43](#_Toc474920674)

[VSync 44](#_Toc474920675)

[skyEngine.Menu 44](#_Toc474920676)

[Button 44](#_Toc474920677)

[Checkbox 45](#_Toc474920678)

[RadioButton 46](#_Toc474920679)

[skyEngine.Inventory 48](#_Toc474920680)

[AbstractItem 48](#_Toc474920681)

[StaticInventory 48](#_Toc474920682)

[DynamicInventory 50](#_Toc474920683)

[**License** 53](#_Toc474920684)

# **Overview**

SkyEngine is, at its core, a game engine designed for the Java programming language – it is what enables a game to run efficiently in a Java Runtime Environment. SkyEngine is a collection of Java packages intended to help developers who wish to create 2D games. Within each package are classes which act as the unique features of the engine. The classes are sorted into packages by intended function; each package contains classes whose functions are related to or dependent upon others. SkyEngine was designed to be semi-modular, so it is possible to use some packages independent of other packages, however some packages depend on each other.

SkyEngine makes use of the Lightweight Java Game Library (LWJGL) and Slick2D. By making use of these Java libraries, SkyEngine is able to efficiently use modern computing technology to enhance the gameplay experience. These libraries allow SkyEngine to use the OpenGL API. This moves all graphics processing from the CPU of a computer to the GPU which improves the overall appearance and performance of games made with SkyEngine.

SkyEngine was built with the intent of being platform independent, which is why it uses the Java programming language. Games made with SkyEngine can run on any Java-enabled device with a GPU capable of running OpenGL.

Open-source software has become increasingly popular in recent years, which is why SkyEngine will be kept open source. One goal of SkyEngine is to assist Java developers in making games, and keeping it open source enables any Java developer to use it without having to pay a fee. Additionally, open-source software can be modified and extended by anyone who has an interest in it, so SkyEngine will grow as the community around it grows. SkyEngine is not compiled into a binary format, so it is flexible and can be edited by developers to suit their needs as they develop their games. This means that over time, several different “flavors” of SkyEngine could emerge, each with specific functionalities that do not exist in other versions.

# **Using SkyEngine**

## Starting a Project

**Note:** If you are using Eclipse, you can just import an empty SkyEngine project into Eclipse and skip to step 5.

1. Create a new project in the IDE of your choice
2. Add LWJGL 2.9.3 and Slick-Util to your project
3. Configure your build path to include the added libraries
4. Import the required SkyEngine packages

**Note:** If you do not know which packages to import, import all packages and remove what you do not need later.

1. Create a new package to contain your game files
2. Add a new Java class to your package
3. Make your class extend AbstractGame
4. Import AbstractGame from skyEngine.Core in your class
5. Add unimplemented methods from AbstractGame
6. Import GameContainer and Renderer from skyEngine.Core in your class
7. Add a main method to your class
8. Add the following code to your main method, inserting your own desired values:

GameContainer gc = new GameContainer(new ***<CLASS>***());

gc.setDesignWidth(***<DESIGN WIDTH>***);

gc.setDesignHeight(***<DESIGN HEIGHT>***);

gc.setFullscreenOnStart(***<FULL SCREEN>***);

gc.setMaxResolutionOnStart(***<MAX RESOLUTION>***);

gc.startGame();

***<CLASS>*** = The name of the class your main method is in

***<DESIGN WIDTH>*** = The width your game was designed for in pixels (Type: int) (Default: 1920)

***<DESIGN HEIGHT>*** = The height your game was designed for in pixels (Type: int) (Default: 1080)

***<FULL SCREEN>*** = Should the game launch in full screen mode (Type: boolean) (Default: true)

***<MAX RESOLUTION>*** = Should the game launch with a resolution to match the display (Type: boolean) (Default: true)

**Note:** You do not have to invoke the setter method if your desired value is the same as the default value.

## Using the Game Loop

The game loop of SkyEngine calls up to two methods each cycle; these methods are update and render respectively. Knowing this is essential to using SkyEngine properly. The update method is called before the render method and updates all of the objects and variables of your game. Once everything in the game updates, the render method is called. When this method is called, everything within the render method is rendered to the screen. After the render method is called, the loop begins again with the update method. Keep this in mind when making your game.

## Getting User Keyboard Input

User input is handled in the update method of your game. User input is also static, so you do not need to create an instance of an Input object in your game. If you ever need to make a non-static reference to the Input class, call the *getInput()* method using the GameContainer object that is passed into the update method of the game loop. [Example: *gc.getInput()*]

There are three methods that can be used to get keyboard input from the user. All of the following methods should be accessed statically through the Input class. These methods are *isKey()*, *isKeyPressed()*, and *isKeyReleased()*. Each method takes an integer, which represents the key on the keyboard, as a parameter. Get the static key value from org.lwjgl.input.Keyboard to use as the parameter when calling these methods. An example of this would be calling *Input.isKey(Keyboard.KEY\_F11)* which would test if the F11 key is pressed. The three methods are discussed in further detail below.

isKey(Keyboard.***<VALUE>***) – This method tests whether or not a key is currently down. It returns true if the key is down, false otherwise.

isKeyPressed(Keyboard.***<VALUE>***) – This method tests whether or not a key was just pressed. It returns true if the key was up during the previous update and is currently down, false otherwise.

isKeyReleased(Keyboard.***<VALUE>***) – This method tests whether or not a key was just released. It returns true if the key was down during the previous update and is currently up, false otherwise.

***<VALUE>*** = The value of the key to be checked; an integer value from org.lwjgl.input.Keyboard

## Rendering Quads to the Screen

Rendering quads is the most basic way of rendering data to the screen and is called in the render method of the main game loop. The quads in SkyEngine are all rectangles and oriented with no rotation. To fully understand how to render to the screen, it is important to realize that the top left corner has an ordered pair value of (0, 0). It is also important to understand that it is possible to draw to the screen using float values instead of being restricted to integer values.

To render a quad to the screen, call the *drawColorQuad()* method using the Renderer object that is passed into the render method of the game loop. An example of calling this method would look like *r.drawColorQuad(100, 100, 200, 200, 255, 0, 0, 255)*. This example would create a rectangle with dimensions of 200px by 200px at (100, 100), a color value of (255, 0, 0), and an alpha value of 255.

drawColorQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<RED>***, ***<GREEN>***, ***<BLUE>***, ***<ALPHA>***) – This method draws a colored rectangle to the screen.

drawColorQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<RED>***, ***<GREEN>***, ***<BLUE>***, ***<ALPHA>***, ***<ROT>***) – This method draws a colored rectangle to the screen that is rotated about its center.

***<X>*** = The x-coordinate of the top left corner of the rectangle

***<Y>*** = The y-coordinate of the top left corner of the rectangle

***<WIDTH>*** = The width of the rectangle in pixels  
***<HEIGHT>*** = The height of the rectangle in pixels  
***<RED>*** = The amount of red in the color of the rectangle (0 – 255)  
***<GREEN>*** = The amount of green in the color of the rectangle (0 – 255)

***<BLUE>*** = The amount of blue in the color of the rectangle (0 – 255)

***<ALPHA>*** = The amount of alpha (transparency) of the rectangle (0 – 255) where a value of 255 is an opaque rectangle

***<ROT>*** = The number of degrees clockwise to rotate the rectangle

## Understanding and Using Delta Time

Delta time is the change in time (in seconds) between calls of the update method. The goal of delta time is to make sure gameplay is not hindered due to frame drops or excessive framerates. It accomplishes this by multiplying the change of a variable by the time between updates. So, if a variable is supposed to be incremented by 200 every second, the end user does not have to worry about doing division with the framerate. By simply incrementing the variable by the product of 200 and the delta time amount, the variable will be incremented by 200 every second regardless of changes in framerate.

Take for example, a variable (in this case, it will be xPosition) that represents the x-coordinate of the location of a rectangle that is being rendered to the screen. To move the rectangle to the right at a rate of 100 pixels per second, increment it by 100 multiplied by the delta time variable passed into the update method (deltaTime). All uses of delta time should occur within the update method of the main game loop. So, in the update method, increasing the xPosition variable by 100 each second would look like: xPosition += 100 \* deltaTime;

## Getting User Mouse Input

Mouse input is handled very similarly to keyboard input. It is static and is typically handled in the update method, but it can be handled in the render method if needed. If you ever need to make a non-static reference to the Input class, call the *getInput()* method using the GameContainer object that is passed into the update method of the game loop. [Example: *gc.getInput()*]

Mouse input has more methods than keyboard input, and it has three variables that you can reference which will be explained later. The mouse has three methods to get which button is being used and how it is being used, just like keyboard input. These methods are *isButton()*, *isButtonPressed()*, and *isButtonReleased()*. All three of these methods take an integer value as a parameter and return a boolean value. The integer input is an integer representation of which button of the mouse is being used. They can be called statically from the Input class. In addition to having methods related to which buttons are being used, mouse input involves the position of the mouse within the game. There are two methods to get the mouse position [*getMouseX()* and *getMouseY()*] and one method to set the mouse position [*setMousePosition()*].

IsButton(Input.***<VALUE>***) – This method tests whether or not a button is currently down. It returns true if the button is down, false otherwise.

IsButtonPressed(Input.***<VALUE>***) – This method tests whether or not a button was just pressed. It returns true if the button was up during the previous update and is currently down, false otherwise.

IsButtonReleased(Input.***<VALUE>***) – This method tests whether or not a button was just released. It returns true if the button was down during the previous update and is currently up, false otherwise.

getMouseX() – This method gets the x-coordinate of the mouse based on the game’s design width.

getMouseY() – This method gets the y-coordinate of the mouse based on the game’s design height.

setMousePosition(***<X>***, ***<Y>***) – This method sets the mouse’s position to the specified position in the window relative to the viewport.

***<VALUE>*** = An integer value that can be expressed as *leftMouseButton*, *rightMouseButton*, or *middleMouseButton*, each of which is accessed statically through the Input class

***<X>*** = An integer value representing a specific x-coordinate within the window.

***<Y>*** = An integer value representing a specific y-coordinate within the window.

## Using Game States

A game state is exactly as the name implies; it is the state that your game is in. Game states can include the initialization state, the runtime state, the menu states, and more. Typically, game states are used to control the flow of your game and to make sure that every single resource is not stored in memory at once if it is avoidable. There are a wide variety of ways to handle game states including the use of separate classes, stack structures, and using enum types. Only the use of enum types will be discussed in this documentation.

1. Declare a new enum type for your game states
2. Add the different states to your new enum type
3. Add a variable of the newly declared type to your game to hold the current state

**Note:** Make sure this variable is initialized before the game starts

1. Create a switch statement for the variable in the update and render methods

**Note:** Make sure each state has its own case

1. Write the code for each game state within the different cases of the switch statements

**Note:** Make sure there is a way to switch between states in your code

While game states are not required to be in a game, they greatly help code organization and the flow of the program.

## Playing Audio

Most modern games use audio along with video to create a memorable experience. SkyEngine has its own way to decode and play Wave files (.wav extension). This is all done through the SoundClip class. Playing audio requires there to be an instance of a SoundClip object with a valid path to the sound file. Creating a SoundClip object can be done by calling one of the two overloaded constructors.

**Note:** Sound files must be Wave files with 16-bit depth

SoundClip(***<PATH>***) – This constructor takes only the path to the sound file and has a default gain of 0.0 dB.

SoundClip(***<PATH>***, ***<GAIN>***) – This constructor takes both the path to the sound file and the desired gain of the clip.

***<PATH>*** = A string that represents the path to the sound file in the project

***<GAIN>*** = The desired gain of the clip in dB (-80.0 to 6.0206)

Only after a SoundClip object has been successfully created can it be played. It has basic starting and stopping capabilities through the *startClip()* and *stopClip()* methods respectively. There is also a way to pause and resume playing the sound at the position it was paused at through the *pauseClip()* and *unpauseClip()* methods. Additionally, there is a method to toggle the paused status of the audio; it is *togglePause()*. Beyond this, there is even a way to play the sound on an endless loop by calling *loopClip()*.

startClip() – This method plays the sound from the beginning of the file.

stopClip() – This method stops the clip and resets it to the beginning for the next playback.

pauseClip() – This method stops the clip but allows it to continue being played from the frame on which it was stopped.

unpauseClip() – This method plays a paused clip from the frame on which it was stopped.

togglePause() – If the clip is paused, it calls the *unpauseClip()* method; otherwise it calls the *pauseClip()* method.

loopClip() – This begins playing the clip in an endless loop.

One other feature of audio on computers is the capability to change the volume of the output. SkyEngine does this by allowing the user to adjust the gain setting with the *setVolume()* method.

setVolume(***<GAIN>***) – This method changes the gain of the audio clip.

***<GAIN>*** = A float value of the desired gain in dB (-80.0 – 6.0206)

**Note:** When setting the gain, remember that decibels are a logarithmic unit.

## Rendering with Textures

Rendering colored rectangles is not the only way to get graphics in a game; you can also render textures (images) in your game. The first thing you need to do is create a Texture object (imported from org.newdawn.slick.opengl.Texture). This Texture object must be initialized before it can be used. To do this, set the Texture object to be equal to the value that the *loadTexture()* method returns. The *loadTexture()* method can be called by using an instance of a Renderer object [which can be accessed in the render method of the main game loop or by using an instance of a GameContainer object to call the *getRenderer()* method]. An example using the variable “image” of type Texture would look like:

image = gc.getRenderer().loadTexture(“package/filename.png”, “PNG”);

loadTexture(***<PATH>***, ***<TYPE>***) – This method tries to load an image from a file and return the loaded image as a Texture object.

***<PATH>*** = A string representation of the path to the image to be loaded

***<TYPE>*** = A string representing the type of file to be loaded (typically the file extension)

Once the Texture object is properly initialized, it can be rendered to the screen by calling the *drawTextureQuad()* or *drawScaledTextureQuad()* methods using an instance of a Renderer object in the render method of the game loop. The quad will automatically have a width and height matching those of the image of the Texture object that is being rendered unless otherwise specified.

drawTextureQuad(***<X>***, ***<Y>***, ***<TEX>***) – This method draws an image at a specified location. The dimensions of the quad match those of the image used as the texture.

drawTextureQuad(***<X>***, ***<Y>***, ***<TEX>***, ***<ROT>***) – This method draws an image at a specified location. The dimensions of the quad match those of the image used as the texture. The quad is rotated about its center.

drawTextureQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<TEX>***, ***<INTERP>***) – This method draws an image at a specified location. The dimensions of the quad are specified and the texture is stretched to fit the dimensions.

drawTextureQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<TEX>***, ***<INTERP>***, ***<ROT>***) – This method draws an image at a specified location. The dimensions of the quad are specified and the texture is stretched to fit the dimensions. The quad is rotated about its center.

drawScaledTextureQuad(***<X>***, ***<Y>***, ***<SCALEX>***, ***<SCALEY>***, ***<TEX>***, ***<INTERP>***) – This method draws an image at a specified location. The quad has dimensions equal to the product of the dimensions of the image and the scale values along each axis.

drawScaledTextureQuad(***<X>***, ***<Y>***, ***<SCALEX>***, ***<SCALEY>***, ***<TEX>***, ***<INTERP>***, ***<ROT>***) – This method draws an image at a specified location. The quad has dimensions equal to the product of the dimensions of the image and the scale values along each axis.

***<X>*** = The x-coordinate of the top left corner of the texture

***<Y>*** = The y-coordinate of the top left corner of the texture

***<WIDTH>*** = The overridden width of the texture

***<HEIGHT>*** = The overridden height of the texture

***<SCALEX>*** = The factor by which the width of the texture should be scaled

***<SCALEY>*** = The factor by which the height of the texture should be scaled

***<TEX>*** = The texture to be rendered

***<INTERP>*** = A boolean value representing whether or not interpolation should be used when scaling a texture. When scaling textures, using interpolation causes there to be a blur effect to get rid of pixelation that may occur during the scaling process. Conversely, if interpolation is not used, the scaled image may appear to be pixelated. Generally, do not use interpolation if you are looking for a sharper look. Whenever possible, use textures with higher resolutions rather than scaling the textures up to fit a certain size.

***<ROT>*** = The number of degrees clockwise to rotate the rectangle

When rendering with textures you may notice some digital artifacts around the edges of your textures. If this is occurring, use an instance of the Renderer to call the *clampTextures()* method. This method only needs to be called once and should be called either right before or after the textures are initialized. If you find other issues that occur as a result of calling the *clampTextures()* method, you can either delete the call to *clampTextures()* or call *unclampTextures()*.

## Rendering Text

The last major part of using the core functionalities of SkyEngine is rendering text to the screen. This is an extremely important part of most games because it allows you to directly express an idea to the player. Currently, there is support for default Java fonts and TrueType fonts. Before it is possible to render text to the screen, a SkyFont object must be created and initialized. SkyFont objects are initialized with normal constructors, unlike Texture objects.

SkyFont() – The default constructor for SkyFont objects. Initializes the object with a plain Times New Roman font and a size of 32.

SkyFont(***<FONT>***, ***<STYLE>***, ***<SIZE>***) – The constructor to be used when creating SkyFont objects with a system font. The size can be as large as you want, but there are severe issues when the size exceeds a specific number that varies for each font.

SkyFont(***<PATH>***, ***<SIZE>***) – The constructor to be used when creating a SkyFont object from a TrueType font file. The size can be as large as you want, but there are severe issues when the size exceeds a specific number that varies for each font.

***<FONT>*** = A string value containing the name of a system font

***<STYLE>*** = An integer value for the font style (default fonts) taken from java.awt.Font

***<PATH>*** = A string value containing the path to a TrueType font file (.ttf extension)

***<SIZE>*** = The height of the font to be used. There are known issues regarding the rendering of text and the size of the fonts. For each font, there is essentially a threshold for how large the font can be before serious rendering issues occur. This will hopefully be patched in the next update.

After the SkyFont object is initialized if you want to change the color from white, call the *setColor()* method which supports transparency. If you are using a system font and would like to change the style, call the *setStyle()* method.

setColor(***<RED>***, ***<GREEN>***, ***<BLUE>***, ***<ALPHA>***) – Changes the color of the font to the specified color.

setStyle(***<STYLE>***) – Changes the style of a system font to the specified style.

***<RED>*** = The amount of red to be in the color (0 – 255)

***<GREEN>*** = The amount of green to be in the color (0 – 255)

***<BLUE>*** = The amount of blue to be in the color (0 – 255)

***<ALPHA>*** = The amount of alpha (opacity) to be in the color (0 – 255) where a value of 255 makes the text opaque

***<STYLE>*** = An integer value for the font style (default fonts) taken from java.awt.Font

Now that the SkyFont object is properly configured, it is possible to render text to the screen by using your SkyFont object to call the *drawText()* method. This is generally handled in the render method of the main game loop.

drawText(***<X>***, ***<Y>***, ***<TEXT>***) – This method draws a specified string to the screen at a given position.

***<X>*** = The x-coordinate of the upper left corner of the text that is being rendered

***<Y>*** = The y-coordinate of the upper left corner of the text that is being rendered

***<TEXT>*** = The text that is to be rendered to the screen

**Note:** There are many known issues with the SkyFont class so it may be easier to just render text by using different textures until the SkyFont class is reworked in future updates.

## Using VSync

VSync is an important part of modern games. In short, it prevents graphical tearing when rendering at high framerates and tries to synchronize the framerate of the game with that of the monitor. VSync is handled statically in SkyEngine and only has a few methods from the skyEngine.RenderingUtilities.VSync class.

enableVSync() – This method turns VSync on

disableVSync() – This method turns VSync off

toggleVSync() – This method turns VSync on if it is off and turns VSync off if it is on

While SkyEngine tries to cap the framerate to match the refresh rate of the player’s monitor by default, it does not fix issues with graphical tearing. Therefore, it is a wise decision to use VSync whenever possible.

## Using Antialiasing

Antialiasing is another major part of rendering in modern games. In a nutshell, antialiasing smooths edges of shapes that may appear to be jagged due to the nature of pixels. With SkyEngine, there is a way to set the game to launch with antialiasing enabled. In the main method created in the Starting a Project section, simply call the static *setInitialAntiAliasValues()* method from the skyEngine.RenderingUtilities.AntiAliasing class before you call the *startGame()* method.

setInitialAntiAliasValues(***<ALP>***, ***<DEP>***, ***<STE>***, ***<SAM>***) – This method sets the values for antialiasing for when the window is created. Should the values provided in this method fail, the game will be launched without antialiasing to prevent it from crashing.

***<ALP>*** = The number of alpha bits for antialiasing

***<DEP>*** = The number of depth bits for antialiasing  
***<STE>*** = The number of stencil bits for antialiasing

***<SAM>*** = The number of samples for antialiasing

**Note:** If you do not know what the value does, you should probably leave it set to zero. A good system to use if you do not know what any of these values do is having the parameters set to ***<ALP>*** = 8, ***<DEP>*** = 0, ***<STE>*** = 0, and ***<SAM>*** = a power of two less than or equal to 16. This rule holds true for the entire Using Antialiasing section. It is also possible to call the *setInitialAntiAliasValues()* method from a GameContainer object [gc.setInitialAntiAliasValues()] which just calls the *setInitialAntiAliasValues()* method from the skyEngine.RenderingUtilities.AntiAliasing class. It uses the same parameters.

Sometimes it is necessary to change the antialiasing values in the middle of the game via a menu option. This is possible by calling any of the following methods which enable antialiasing if it is disabled.

setAntiAliasAlpha(***<ALP>***) – This method changes the number of alpha bits being used in antialiasing. If the provided value fails, the number of alpha bits will be set to its previous value.

setAntiAliasDepth(***<DEP>***) – This method changes the number of depth bits being used in antialiasing. If the provided value fails, the number of depth bits will be set to its previous value.

setAntiAliasStencil(***<STE>***) – This method changes the number of stencil bits being used in antialiasing. If the provided value fails, the number of stencil bits will be set to its previous value.

setAntiAliasSamples(***<SAM>***) – This method changes the number of samples being used in antialiasing. If the provided value fails, the number of samples will be set to its previous value.

setAntiAliasValues(***<ALP>***, ***<DEP>***, ***<STE>***, ***<SAM>***) – This method changes all of the antialiasing variables. If any of the provided values fail, the variables will be set to their previous values.

***<ALP>*** = The number of alpha bits for antialiasing

***<DEP>*** = The number of depth bits for antialiasing  
***<STE>*** = The number of stencil bits for antialiasing

***<SAM>*** = The number of samples for antialiasing

In other cases, it is necessary to disable antialiasing entirely. As a result, there is also a need to enable antialiasing with the values that are already stored. This can be accomplished by calling the *disableAntiAlias()* and *enableAntiAlias()* methods.

**Note:** If you notice that the textures that you are using are not being antialiased, this is because the antialiasing in SkyEngine does not affect textures that are rendered inside quads. The antialiasing in SkyEngine only affects shapes that are drawn to the screen. If you want images to be antialiased, this will need to be done in a third-party image manipulation software.

## Using Buttons

SkyEngine comes with a few classes that can be used to make menus without much coding. One of these classes is the Button class in the skyEngine.Menu package. This allows you to create buttons which need to be updated and rendered with simple calls in the update and render methods of the main game loop. The following steps will create a button object and implement it properly.

1. Create a new Button object using the skyEngine.Menu.Button class
2. Create a new Texture object just as you would for rendering with textures
3. Initialize the Texture object just as you would for rendering with textures
4. Initialize the Button object by calling the constructor [Button(***<X>***, ***<Y>***, ***<TEX>***)]
5. Call the *update()* method of the button in the update method of the game loop
6. Call the *render()* method of the button in the render method of the game loop

***<X>*** = The x-coordinate of the upper left corner of the button

***<Y>*** = The y-coordinate of the upper left corner of the button

***<TEX>*** = The texture of the button to be rendered

**Note:** Textures used for buttons currently do not have scaling supported, so the button will render the texture at the resolution and size that it reads from the file.

Once the button object is created successfully and has its *update()* and *render()* methods called in the appropriate parts of the game loop, the button can be used to handle events. Both of the methods that handle events return boolean values and rely on mouse input which is handled internally by the Button object.

isPressed() – Returns true if the button was just pressed. If the button is held down, it will return true for the first update and then false until it is released and pressed again.

isDown() – Returns true if the left mouse button is down and the mouse is positioned within the bounds of the Button object.

## Using Checkboxes

Checkboxes are frequently used in menus, so they were added to SkyEngine. Creating a Checkbox object is very similar to Button objects except for the fact that two Texture objects are needed instead of one. There is also an overloaded constructor to provide more flexibility while working with Checkbox objects.

Checkbox(***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***) – This is the constructor for Checkbox objects. It creates a Checkbox object at a given position and sets the two textures to be used in rendering.

Checkbox(***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***, ***<CHK>***) – This is another constructor for Checkbox objects. It creates a Checkbox object at a given position, sets the two textures to be used in rendering, and sets whether the Checkbox object should initially be checked.

***<X>*** = The x-coordinate of the upper left corner of the Checkbox object

***<Y>*** = The y-coordinate of the upper left corner of the Checkbox object  
***<UTEX>*** = The texture for the unchecked Checkbox object

***<CTEX>*** = The texture for the checked Checkbox object

***<CHK>*** = A boolean value for whether the Checkbox should be initialized as checked or not

Just like with Button objects, Checkbox objects must have their *update()* and *render()* methods called in the update and render methods of the main game loop. Once the Checkbox object is properly created, if the player clicks within the bounds of the Checkbox object, it toggles whether it is checked or not. The boolean value of whether or not the Checkbox object is checked can be accessed through the *isChecked()* method. In addition to getting the value of whether the Checkbox object is checked, it is possible to set this value by calling the *setChecked()* method and passing in a parameter of the value the checked status of the Checkbox object should be set to.

## Using Radio Buttons

The last class in the skyEngine.Menu package, RadioButton, is an extension of the Checkbox class. RadioButton objects are simply collections of Checkbox objects that behave in a way such that the player can only select one box from the collection. Just like with Checkbox objects, two Texture objects are required for the constructor. There is an overloaded constructor which allows for flexibility when making RadioButton objects.

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***) – This creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position. This uses the default gap between Checkbox objects (half of the height) and a vertical orientation.

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***, ***<HORIZ>***) – This creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position. The orientation is taken as a parameter and the gap between Checkbox objects is the default (half of the height or width depending on orientation).

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<GAP>***, ***<UTEX>***, ***<CTEX>***) – This creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position. The gap between Checkbox objects is passed in as a parameter and the orientation is vertical.

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<GAP>***, ***<UTEX>***, ***<CTEX>***, ***<HORIZ>***) – This creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position. The gap between Checkbox objects and the orientation are passed in as parameters.

***<QTY>*** = The number of Checkboxes to make the RadioButton object with

***<X>*** = The x-coordinate of the upper left corner of the first Checkbox object

***<Y>*** = The y-coordinate of the upper left corner of the first Checkbox object

***<UTEX>*** = The texture for the unchecked Checkbox object

***<CTEX>*** = The texture for the checked Checkbox object

***<HORIZ>*** = A boolean representing if the RadioButton object should be oriented horizontally

***<GAP>*** = The gap between Checkbox objects in pixels

Similarly to Button objects and Checkbox objects, the *update()* and *render()* methods of the RadioButton object must be put in the update and render methods of the game loop. Due to the nature of RadioButtons, only one Checkbox object may be selected at a time. Sometimes, one of the Checkbox objects must be selected for a reason and other times it should be possible to have none of the Checkbox objects selected. This can be accomplished by the *lock()* and *unlock()* methods. The *lock()* method locks the RadioButton object into a state where one Checkbox object must always be selected. The *unlock()* method unlocks the RadioButton object and allows the possibility of there being no Checkbox objects selected.

The most important information to get from a RadioButton object is the index of the Checkbox object that is selected. The value of the index can be accessed by calling the *getSelectedIndex()* method. When calling this, remember that the values start with 0 and count up from there, so the first Checkbox object has an index of 0, the second has an index of 1, and so on. If none of the Checkbox objects are selected, the *getSelectedIndex()* method returns a value of -1. It is also possible to set which Checkbox object is selected by calling the *setChecked()* method and passing in the index of the Checkbox object to be selected as a parameter.

## Creating Items

Many modern games have items that the player can use or keep in their inventory for a variety of reasons. SkyEngine has a template for items in the skyEngine.Inventory package called AbstractItem. Each item has a few variables that are inherited by subclasses of the AbstractItem class. These variables are quantity, maxQuantity, name, and itemID. The instructions for how to create items to use in games will be on the next page.

1. Create a new Java class for your item
2. Make the class extend the AbstractItem class in skyEngine.Inventory
3. Create a constructor for the new class
4. Within the constructor, add the following code:

quantity = ***<QTY>***;

maxQuantity = ***<MAX>***;

name = ***<NAME>***;

itemID = ***<ID>***;

1. Add any other methods that the item may need

***<QTY>*** = The initial value of the quantity variable

***<MAX>*** = The maximum value for the quantity variable

***<NAME>*** = The name of the object

***<ID>*** = The ID number for the object

To use the item that you created in your game, simply create an object using the class that you just made. It is possible to manipulate the variables of the object by calling standard getter and setter methods for the variables.

## Using Static Inventories

A lot of games have inventories for the player, NPCs, and containers. There are two default ways to handle inventories in SkyEngine, the first being static inventories. The other way of handling inventories is by using dynamic inventories which will be discussed later. Static inventories are inventories which have a fixed length, which can be thought of as a collection of “item slots”. Static inventories are handled with an array, which is important to know when working with them.

The first step in using static inventories is to create a StaticInventory object using the skyEngine.Inventory.StaticInventory class. The only parameter in the constructor of this object is the size of the inventory or the number of slots in the inventory. When coming up with a size, do not factor in the quantity of different items because quantities are handled within AbstractItem objects, not StaticInventory objects.

After the StaticInventory object is properly created, it can have items added to it. Typically, this is the first step of working with StaticInventory objects after they are created. To add an item to the player’s inventory, call the *addItem()* method and pass in an AbstractItem object as the parameter. This will add the AbstractItem object to the first available spot of the inventory or stack it onto an existing object with the same itemID value if possible. If there are no available spots, it will try to stack as many of the item as it can on existing objects with the same itemID value and simply discard any remaining items.

One other important functionality of inventories in video games is the capability to remove items from the inventory. There are a few ways to do this with StaticInventory objects.

removeItem(***<ITEM>***) – This removes all instances of the specified item from the inventory. It is recommended to use the method with an itemID value as the parameter.

removeItem(***<ID>***) – This removes all instances of items with the specified itemID value from the inventory. This is the recommended method for removing items.

removeItem(***<NAME>***) – This removes all instances of items with the specified name from the inventory. It is recommended to use the method with an itemID value as the parameter.

removeItemAtIndex(***<INDEX>***) – This removes the object at the specified index if the given index is valid.

clearInventory() – This method removes all items from the inventory.

***<ITEM>*** = An AbstractItem object

***<ID>*** = An itemID value

***<NAME>*** = An item’s name

***<INDEX>*** = An index of the inventory

There is also the need to see which item is at a certain position of the inventory and set the item at a given position of the inventory.

getItemAtIndex(***<INDEX>***) – Returns the AbstractItem object at the specified index if an AbstractItem object exists there. Otherwise, it returns null.

setItemAtIndex(***<ITEM>***, ***<INDEX>***) – Sets the value of the AbstractItem object at the specified index to the specified AbstractItem object if the index is valid.

***<INDEX>*** = An index of the inventory

***<ITEM>*** = An AbstractItem object

Sometimes it is also necessary to see if a certain item is in the player’s inventory. This can be used to see if the player has an item that is required to progress in the game or for other reasons. In addition to simply seeing if the item exists, it can also be necessary to see where it exists within the inventory.

containsItem(***<ITEM>***) – Returns true if an instance of the specified item exists within the inventory. Otherwise, it returns false. It is recommended to use the method with an itemID value as the parameter.

containsItem(***<ID>***) – Returns true if an instance of an item with the specified itemID value exists. Otherwise, it returns false. This is the recommended method for seeing if an item exists within the inventory.

containsItem(***<NAME>***) – Returns true if an instance of an item with the specified name exists. Otherwise, it returns false. It is recommended to use the method with an itemID value as the parameter.

indexOfItem(***<ITEM>***) – Returns the index of the first occurrence of the specified item if it exists. If it does not exist, a value of -1 is returned. It is recommended to use the method with an itemID value as the parameter.

indexOfItem(***<ID>***) – Returns the index of the first occurrence of an item with the specified itemID value if it exists. If it does not exist, a value of -1 is returned. This is the recommended method for getting the index of an item.

indexOfItem(***<NAME>***) – Returns the index of the first occurrence of an item with the specified name if it exists. If it does not exist, a value of -1 is returned. It is recommended to use the method with an itemID value as the parameter.

***<ITEM>*** = An AbstractItem object

***<ID>*** = An itemID value

***<NAME>*** = An item’s name

Manipulating the location of the contents of inventories is also a common feature in games. The StaticInventory class contains methods which move all empty slots to the end of the array and multiple ways to sort the inventory.

moveEmtpyToEnd() – Moves all empty slots of the inventory to the end of the array

sortByQuantityAscending() – Sorts the items by their quantity from low to high

sortByQuantityDescending() – Sorts the items by their quantity from high to low

sortByNameAscending() -Sorts the items by their name from A to Z

sortByNameDescending() – Sorts the items by their name from Z to A

sortByIDAscending() – Sorts the items by their itemID value from low to high

sortByIDDescending() – Sorts the items by their itemID value from high to low

The last major functionality of StaticInventory objects is the ability to read the size of the inventory and set the size of the inventory to a new size.

getSize() – Returns the size of the inventory

setSize(***<SIZE>***) – Sets the size of the inventory to a specified size. If the new inventory is smaller than the old inventory, the items at the end of the inventory are removed to fit the new size of the inventory. If the new inventory is larger than the old inventory, empty slots will be created at the end of the new inventory.

***<SIZE>*** = The number of slots in the new inventory

**Note:** Sometimes, you may find that working with objects in the inventory is difficult when using AbstractItem objects as parameters. It is highly recommended to use itemID values as parameters whenever possible as this way reduces possible errors.

## Using Dynamic Inventories

Dynamic inventories are very similar to static inventories, but the main difference is the fact that the size of dynamic inventories is not fixed. This means that the inventory grows and shrinks as items are added to and removed from it. One feature of dynamic inventories is that there are never spots filled with null values. The skyEngine.Inventory.DynamicInventory class handles this by default and makes sure that no null pointers can exist within a dynamic inventory.

Like with StaticInventory objects, the first step to using DynamicInventory objects is to create an instance of a DynamicInventory object using the default constructor [*DynamicInventory()*]. After creating an instance of a DynamicInventory object, items can be added to the inventory and the inventory can be manipulated.

Adding items to the inventory is typically the first step in using DynamicInventory objects. This can be done by calling the *addItem()* method and using and AbstractItem object as the parameter. DynamicInventory objects handle items in an ArrayList, so any items that are added tend to be added to the end of the ArrayList. The exception to this case is when an AbstractItem with the same itemID value as an item already in the inventory is added. In this situation, the DynamicInventory object tries to stack as many of the item that is being added onto preexisting items with the same itemID value that have not reached their maximum quantities. Only after it stacks the objects as well as possible, it will put the item that is being added at the end of the inventory with the modified quantity.

**Note:** When using DynamicInventory objects to handle inventory management, it is often wise to set the maximum quantity value for all AbstractItems to Integer.MAX\_VALUE so that stacking can occur properly.

Just like StaticInventory objects, DynamicInventory objects also have the ability to remove items from themselves when necessary.

removeItem(***<ITEM>***) – This removes all instances of the specified item from the inventory. It is recommended to use the method with an itemID value as the parameter.

removeItem(***<ID>***) – This removes all instances of items with the specified itemID value from the inventory. This is the recommended method for removing items.

removeItem(***<NAME>***) – This removes all instances of items with the specified name from the inventory. It is recommended to use the method with an itemID value as the parameter.

removeItemAtIndex(***<INDEX>***) – This removes the object at the specified index if the given index is valid.

clearInventory() – This method removes all items from the inventory.

***<ITEM>*** = An AbstractItem object

***<ID>*** = An itemID value

***<NAME>*** = An item’s name

***<INDEX>*** = An index of the inventory

It is possible to see the item at an index and replace it with a different item. The process is exactly like it is with StaticInventory objects.

getItemAtIndex(***<INDEX>***) – Returns the AbstractItem object at the specified index if an AbstractItem object exists there. Otherwise, it returns null.

setItemAtIndex(***<ITEM>***, ***<INDEX>***) – Sets the value of the AbstractItem object at the specified index to the specified AbstractItem object if the index is valid.

***<INDEX>*** = An index of the inventory

***<ITEM>*** = An AbstractItem object

There are also methods to check to see if a specific item is in the player’s inventory. Beyond this, it is possible to get the index of a certain item in the player’s inventory. Once again, these methods behave exactly like the methods for StaticInventory objects.

containsItem(***<ITEM>***) – Returns true if an instance of the specified item exists within the inventory. Otherwise, it returns false. It is recommended to use the method with an itemID value as the parameter.

containsItem(***<ID>***) – Returns true if an instance of an item with the specified itemID value exists. Otherwise, it returns false. This is the recommended method for seeing if an item exists within the inventory.

containsItem(***<NAME>***) – Returns true if an instance of an item with the specified name exists. Otherwise, it returns false. It is recommended to use the method with an itemID value as the parameter.

indexOfItem(***<ITEM>***) – Returns the index of the first occurrence of the specified item if it exists. If it does not exist, a value of -1 is returned. It is recommended to use the method with an itemID value as the parameter.

indexOfItem(***<ID>***) – Returns the index of the first occurrence of an item with the specified itemID value if it exists. If it does not exist, a value of -1 is returned. This is the recommended method for getting the index of an item.

indexOfItem(***<NAME>***) – Returns the index of the first occurrence of an item with the specified name if it exists. If it does not exist, a value of -1 is returned. It is recommended to use the method with an itemID value as the parameter.

***<ITEM>*** = An AbstractItem object

***<ID>*** = An itemID value

***<NAME>*** = An item’s name

It is also possible to manipulate the location of the contents of DynamicInventory objects. The main difference between manipulating the location of the contents of DynamicInventory objects and StaticInventory objects is the lack of the moveEmptyToEnd() method in the DynamicInventory class. This change is due to the fact that DynamicInventory objects cannot have empty spots.

sortByQuantityAscending() – Sorts the items by their quantity from low to high

sortByQuantityDescending() – Sorts the items by their quantity from high to low

sortByNameAscending() -Sorts the items by their name from A to Z

sortByNameDescending() – Sorts the items by their name from Z to A

sortByIDAscending() – Sorts the items by their itemID value from low to high

sortByIDDescending() – Sorts the items by their itemID value from high to low

It is also possible to get the size of a Dynamic inventory with the *getSize()* method. Beyond this, there are standard getters and setters for DynamicInventory objects.

# **Sample Project - Pong**

package gamePackage;

import java.util.Random;

import org.lwjgl.input.Keyboard;

import org.newdawn.slick.opengl.Texture;

import skyEngine.Core.AbstractGame;

import skyEngine.Core.GameContainer;

import skyEngine.Core.Input;

import skyEngine.Core.Renderer;

import skyEngine.Core.SoundClip;

import skyEngine.RenderingUtilities.AntiAliasing;

import skyEngine.RenderingUtilities.VSync;

public class MainGame extends AbstractGame

{

private static Paddle *playerPaddle*;

private static Paddle *cpuPaddle*;

private static Ball *ball*;

private static SoundClip *bounceSound*;

private GameState gs = GameState.*Initialize*;

private static Texture *menuOverlay*;

private static Texture *pausedOverlay*;

private static Texture *winOverlay*;

private static Texture *loseOverlay*;

private int dWidth;

private int dHeight;

public static void main(String[] args)

{

GameContainer gc = new GameContainer(new MainGame());

gc.setDesignWidth(1920);

gc.setDesignHeight(1080);

gc.setFullscreenOnStart(true);

gc.setMaxResolutionOnStart(true);

AntiAliasing.*setInitialAntiAliasValues*(8, 0, 0, 4);

gc.startGame();

}

@Override

public void update(GameContainer gc, float deltaTime)

{

switch (gs)

{

case *Initialize*:

dWidth = GameContainer.*getDesignWidth*();

dHeight = GameContainer.*getDesignHeight*();

*playerPaddle* = new Paddle(gc, 255, 255, 255);

*playerPaddle*.setX(0);

*playerPaddle*.setY((float) ((dHeight - *playerPaddle*.getHeight()) / 2));

*cpuPaddle* = new Paddle(gc, 255, 0, 0);

*cpuPaddle*.setX(dWidth - *cpuPaddle*.getWidth());

*cpuPaddle*.setY((float) ((dHeight - *cpuPaddle*.getHeight()) / 2));

*ball* = new Ball(gc, 255, 255, 255);

*ball*.setX((float) ((dWidth - *ball*.getWidth()) / 2));

*ball*.setY((float) ((dHeight - *ball*.getHeight()) / 2));

*ball*.initializeDirection();

*bounceSound* = new SoundClip("/gameResources/Pong Bounce.wav");

*menuOverlay* = gc.getRenderer().loadTexture("gameResources/Pong Menu Overlay.png", "PNG");

*pausedOverlay* = gc.getRenderer().loadTexture("gameResources/Pong Paused Overlay.png", "PNG");

*loseOverlay* = gc.getRenderer().loadTexture("gameResources/Pong Lose Overlay.png", "PNG");

*winOverlay* = gc.getRenderer().loadTexture("gameResources/Pong Win Overlay.png", "PNG");

VSync.*enableVSync*();

gs = GameState.*Menu*;

break;

case *Menu*:

if (Input.*isKeyPressed*(Keyboard.*KEY\_ESCAPE*))

{

gs = GameState.*CleanUp*;

}

if (Input.*isKeyReleased*(Keyboard.*KEY\_SPACE*))

{

gs = GameState.*RunGame*;

}

break;

case *RunGame*:

// Input

if (Input.*isKeyPressed*(Keyboard.*KEY\_ESCAPE*))

{

gs = GameState.*Paused*;

}

if (Input.*isKey*(Keyboard.*KEY\_W*) && (*playerPaddle*.getY() > 0))

{

*playerPaddle*.moveUp(deltaTime);

}

if (Input.*isKey*(Keyboard.*KEY\_S*) && (*playerPaddle*.getY() + *playerPaddle*.getHeight() < dHeight))

{

*playerPaddle*.moveDown(deltaTime);

}

// Collision

// Walls

if (*ball*.getY() <= 0)

{

*ball*.setY(0);

*ball*.bounceWall();

*bounceSound*.startClip();

}

if (*ball*.getY() >= dHeight - *ball*.getHeight())

{

*ball*.setY(dHeight - *ball*.getHeight());

*ball*.bounceWall();

*bounceSound*.startClip();

}

// Paddles

if (*ball*.getX() <= *playerPaddle*.getWidth() && *ball*.getX() + *ball*.getWidth() >= *playerPaddle*.getX()

&& *ball*.getY() <= *playerPaddle*.getY() + *playerPaddle*.getHeight()

&& *ball*.getY() + *ball*.getHeight() >= *playerPaddle*.getY() && *ball*.getIsRight() == false)

{

*ball*.bouncePaddle();

*bounceSound*.startClip();

}

if (*ball*.getX() + *ball*.getWidth() >= dWidth - *cpuPaddle*.getWidth()

&& *ball*.getX() <= dWidth - *cpuPaddle*.getWidth()

&& *ball*.getY() <= *cpuPaddle*.getY() + *cpuPaddle*.getHeight()

&& *ball*.getY() + *ball*.getHeight() >= *cpuPaddle*.getY() && *ball*.getIsRight() == true)

{

*ball*.bouncePaddle();

*bounceSound*.startClip();

}

// Movement

if ((*ball*.getY() + *ball*.getHeight() / 2 < *cpuPaddle*.getY() + *cpuPaddle*.getHeight() / 2)

&& (*cpuPaddle*.getY() >= 0))

{

*cpuPaddle*.moveUp(deltaTime);

} else if ((*ball*.getY() + *ball*.getHeight() / 2 > *cpuPaddle*.getY() + *cpuPaddle*.getHeight() / 2)

&& (*cpuPaddle*.getY() + *cpuPaddle*.getHeight() <= dHeight))

{

*cpuPaddle*.moveDown(deltaTime);

}

*ball*.move(deltaTime);

// Test for win or lose

if (*ball*.getX() <= -250)

{

gs = GameState.*Lose*;

} else if (*ball*.getX() >= dWidth + 250)

{

gs = GameState.*Win*;

}

break;

case *Paused*:

if (Input.*isKeyPressed*(Keyboard.*KEY\_ESCAPE*))

{

gs = GameState.*RunGame*;

}

break;

case *Win*:

if (Input.*isKeyPressed*(Keyboard.*KEY\_ESCAPE*))

{

gs = GameState.*CleanUp*;

}

if (Input.*isKeyReleased*(Keyboard.*KEY\_SPACE*))

{

*playerPaddle*.setX(0);

*playerPaddle*.setY((float) ((dHeight - *playerPaddle*.getHeight()) / 2));

*cpuPaddle*.setX(dWidth - *cpuPaddle*.getWidth());

*cpuPaddle*.setY((float) ((dHeight - *cpuPaddle*.getHeight()) / 2));

*ball*.setX((float) ((dWidth - *ball*.getWidth()) / 2));

*ball*.setY((float) ((dHeight - *ball*.getHeight()) / 2));

*ball*.initializeDirection();

gs = GameState.*Menu*;

}

break;

case *Lose*:

if (Input.*isKeyPressed*(Keyboard.*KEY\_ESCAPE*))

{

gs = GameState.*CleanUp*;

}

if (Input.*isKeyReleased*(Keyboard.*KEY\_SPACE*))

{

*playerPaddle*.setX(0);

*playerPaddle*.setY((float) ((dHeight - *playerPaddle*.getHeight()) / 2));

*cpuPaddle*.setX(dWidth - *cpuPaddle*.getWidth());

*cpuPaddle*.setY((float) ((dHeight - *cpuPaddle*.getHeight()) / 2));

*ball*.setX((float) ((dWidth - *ball*.getWidth()) / 2));

*ball*.setY((float) ((dHeight - *ball*.getHeight()) / 2));

*ball*.initializeDirection();

gs = GameState.*Menu*;

}

break;

case *CleanUp*:

System.*exit*(0);

break;

default:

System.*out*.println("There has been a fatal error");

}

}

@Override

public void render(GameContainer gc, Renderer r)

{

switch (gs)

{

case *Initialize*:

// Do nothing

break;

case *Menu*:

*playerPaddle*.render();

*cpuPaddle*.render();

r.drawTextureQuad(0, 0, *menuOverlay*);

break;

case *RunGame*:

*playerPaddle*.render();

*cpuPaddle*.render();

*ball*.render();

break;

case *Paused*:

*playerPaddle*.render();

*cpuPaddle*.render();

*ball*.render();

r.drawTextureQuad(0, 0, *pausedOverlay*);

break;

case *Win*:

*playerPaddle*.render();

*cpuPaddle*.render();

r.drawTextureQuad(0, 0, *winOverlay*);

break;

case *Lose*:

*playerPaddle*.render();

*cpuPaddle*.render();

r.drawTextureQuad(0, 0, *loseOverlay*);

break;

case *CleanUp*:

r.drawColorQuad(0, 0, dWidth, dHeight, 0, 0, 0, 255);

break;

default:

System.*out*.println("There has been a fatal error");

}

}

}

class Paddle

{

private float x;

private float y;

private final float width = 64;

private final float height = 256;

private float red;

private float green;

private float blue;

private GameContainer gc;

private final float speed = 250;

public Paddle(GameContainer gc, float r, float g, float b)

{

red = r;

green = g;

blue = b;

this.gc = gc;

}

public void render()

{

gc.getRenderer().drawColorQuad(x, y, width, height, red, green, blue, 255);

}

public void moveUp(float dt)

{

y -= speed \* dt;

}

public void moveDown(float dt)

{

y += speed \* dt;

}

public float getX()

{

return x;

}

public float getY()

{

return y;

}

public void setX(float newX)

{

x = newX;

}

public void setY(float newY)

{

y = newY;

}

public float getWidth()

{

return width;

}

public float getHeight()

{

return height;

}

}

class Ball

{

private float x;

private float y;

private final float width = 32;

private final float height = 32;

private float red;

private float green;

private float blue;

private GameContainer gc;

private boolean isRight;

private boolean isDown;

private float speed = 250;

public Ball(GameContainer gc, float r, float g, float b)

{

red = r;

green = g;

blue = b;

this.gc = gc;

}

public void render()

{

gc.getRenderer().drawColorQuad(x, y, width, height, red, green, blue, 255);

}

public void initializeDirection()

{

Random rand = new Random();

isRight = rand.nextBoolean();

isDown = rand.nextBoolean();

}

public void move(float dt)

{

if (isRight)

{

x += speed \* dt;

} else

{

x -= speed \* dt;

}

if (isDown)

{

y += speed \* dt;

} else

{

y -= speed \* dt;

}

}

public void bounceWall()

{

isDown = !isDown;

}

public void bouncePaddle()

{

isRight = !isRight;

increaseSpeed();

}

private void increaseSpeed()

{

speed += 5;

}

public float getX()

{

return x;

}

public float getY()

{

return y;

}

public void setX(float newX)

{

x = newX;

}

public void setY(float newY)

{

y = newY;

}

public float getWidth()

{

return width;

}

public float getHeight()

{

return height;

}

public boolean getIsRight()

{

return isRight;

}

public boolean getIsDown()

{

return isDown;

}

}

enum GameState

{

*Initialize*, *Menu*, *RunGame*, *Paused*, *Win*, *Lose*, *CleanUp*

}

# **Summary of Methods**

## skyEngine.Core

### AbstractGame

public abstract void update(***<GC>***, ***<DELTA>***) – This method should call all of the update functions for the objects in your game. It should also tell the engine which objects to update.

public abstract void render(***<GC>***, ***<R>***) – This method should call all of the render functions for the objects in your game. It should also tell the engine which objects to render.

***<GC>*** = An instance of a GameContainer object [handled by game loop]

***<DELTA>*** = The time between updates [handled by game loop]

***<R>*** = An instance of a Renderer object [handled by game loop]

### GameContainer

public GameContainer(***<GAME>***) – This is the constructor for a GameContainer object. It takes your AbstractGame and runs it on SkyEngine.

public void startGame() – Starts the game if it is not already running

public void stopGame() – Stops the game if it is not already stopped

public void run() – This is the main game loop where updates and rendering are called up.

public void setInitialAntiAliasing(***<ALP>***, ***<DEP>***, ***<STE>***, ***<SAM>***) – Sets the Anti-Aliasing values to launch the game with so that they do not need to be altered later.

private void cleanUp() - Cleans up all components of the engine

public static int getDesignWidth() – Gets the design width

public static int getDesignHeight() – Gets the design height

public boolean isFullscreenOnStart() – Gets if the game is launching in fullscreen mode

public boolean isMaxResolutionOnStart() – Gets if the game launches with the maximum resolution

public Window getWindow() – Gets an instance of a Window object

public Renderer getRenderer() – Gets an instance of a Renderer object

public Input getInput() – Gets an instance of an Input object

public int getCurrentFrameRate() – Gets the current framerate

public void setDesignWidth(***<DW>***) – Sets the design width to a new value

public void setDesignHeight(***<DH>***) – Sets the design height to a new value

public void setFullscreenOnStart(***<FS>***) – Sets if the game should launch in fullscreen mode

public void setMaxResolutionOnStart(***<MR>***) – Sets if the game should launch using the maximum resolution

public double getFrameRateCap() – Gets the cap on the framerate

public void setFrameRateCap(***<FCAP>***) – Sets the cap on the framerate

***<GAME>*** = An AbstractGame object

***<ALP>*** = The number of alpha bits for antialiasing

***<DEP>*** = The number of depth bits for antialiasing  
***<STE>*** = The number of stencil bits for antialiasing

***<SAM>*** = The number of samples for antialiasing

***<DW>*** = The design width in pixels

***<DH>*** = The design height in pixels

***<FS>*** = Should the game launch in fullscreen

***<MR>*** = Should the game launch with maximum resolution

***<FCAP>*** = Framerate cap in frames per second

### Window

public Window(***<GC>***) – The constructor for a Window object

private static void createWindow(***<FS>***, ***<MR>***) – Creates the physical window for the game

public static void updateDisplayMode(***<WIDTH>***, ***<HEIGHT>***, ***<FS>***) – This method is used to adjust DisplayMode settings [resolution and fullscreen mode].

private static void adjustRenderArea() – Adjusts the area in which the game will be rendered

private static void adjustViewport() – Sets the viewport to fit the window size of the user

public void update() – Updates the window every frame; prevents game from crashing

public void cleanUp() – Cleans up the OpenGL environment so that the player's computer doesn't freeze

public static DisplayMode[] getIdealDisplayModes() – Gets an array of DisplayMode objects that match the monitor's current bits per pixel and frequency settings

public static DisplayMode[] getIdealDisplayModesByAspectRatio() – Gets an array of DisplayMode objects that match the monitor's current bits per pixel and frequency settings that is sorted by aspect ratio from low to high

public static DisplayMode[] getIdealDisplayModesByWidth() – Gets an array of DisplayMode objects that match the monitor's current bits per pixel and frequency settings that is sorted by width in pixels

public static DisplayMode[] getIdealDisplayModesByHeight() – Gets an array of DisplayMode objects that match the monitor's current bits per pixel and frequency settings that is sorted by height in pixels

private static DisplayMode[] sortByAspectRatio(***<DMUNSORT>***) – Sorts an array of DisplayMode objects by aspect ratio [width / height]

private static DisplayMode[] sortByWidth(***<DMUNSORT>***) – Sorts an array of DisplayMode objects by width

private static DisplayMode[] sortByHeight(***<DMUNSORT>***) – Sorts an array of DisplayMode objects by height

public static void setViewParameters() – Calls a collection of methods to set the render area, viewport, and OpenGL matrix mode to the proper settings

public static boolean isFullscreen() – returns whether or not the game is currently running in fullscreen mode

***<GC>*** = An instance of a GameContainer object

***<FS>*** = Should the game use fullscreen mode

***<MR>*** = Should the game launch with maximum resolution

***<WIDTH>*** = Width of the window

***<HEIGHT>*** = Height of the window

***<DMUNSORT>*** = An unsorted array of DisplayMode objects

### Renderer

public void clearScreen() – Clears the screen of color and depth data

public void drawColorQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<RED>***, ***<GREEN>***, ***<BLUE>***, ***<ALPHA>***) – Draws a colored quad to the screen

public void drawColorQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<RED>***, ***<GREEN>***, ***<BLUE>***, ***<ALPHA>***, ***<DEG>***) – Draws a rotated colored quad to the screen

public Texture loadTexture(***<PATH>***, ***<FORM>***) – Loads a texture into the game environment

public void drawTextureQuad(***<X>***, ***<Y>***, ***<TEX>***) – This method draws a quad with a texture onto the screen. The quad has the dimensions of the image used as the texture.

public void drawTextureQuad(***<X>***, ***<Y>***, ***<TEX>***, ***<DEG>***) – This method draws a quad with a texture onto the screen. The quad has the dimensions of the image used as the texture and is rotated about its center.

public void drawTextureQuad(***<X>***, ***<Y>***, ***<WIDTH>***, ***<HEIGHT>***, ***<TEX>***, ***<INTERP>***) – This method draws a quad with a texture onto the screen. The quad has specified dimensions.

public void drawTextureQuad(***<X>***, ***<Y>***, ***<WIDTH>***,

***<HEIGHT>***, ***<TEX>***, ***<INTERP>***, ***<DEG>***) - This method draws a quad with a texture onto the screen. The quad has specified dimensions and is rotated about its center.

public void drawScaledTextureQuad(***<X>***, ***<Y>***,

***<SCALEX>***, ***<SCALEY>***, ***<TEX>***, ***<INTERP>***) - This method draws a quad with a texture to the screen. The quad is scaled by a specified amount [a scale of 1.0f uses the dimensions of the texture as the dimensions of the quad].

public void drawScaledTextureQuad(***<X>***, ***<Y>***,

***<SCALEX>***, ***<SCALEY>***, ***<TEX>***, ***<INTERP>***, ***<DEG>***) - This method draws a quad with a texture to the screen. The quad is scaled by a specified amount [a scale of 1.0f uses the dimensions of the texture as the dimensions of the quad] and rotated about its center.

***<X>*** = The x-coordinate of the top left corner of the quad

***<Y>*** = The y-coordinate of the top left corner of the quad

***<WIDTH>*** = The width of the quad in pixels  
***<HEIGHT>*** = The height of the quad in pixels  
***<RED>*** = The amount of red in the color of the rectangle (0 – 255)  
***<GREEN>*** = The amount of green in the color of the rectangle (0 – 255)

***<BLUE>*** = The amount of blue in the color of the rectangle (0 – 255)

***<ALPHA>*** = The amount of alpha (transparency) of the rectangle (0 – 255) where a value of 255 is an opaque rectangle

***<ROT>*** = The number of degrees clockwise to rotate the rectangle

***<PATH>*** = A string representation of the path to the image to be loaded

***<FORM>*** = A string representing the type of file to be loaded (typically the file extension)

***<TEX>*** = The texture to be rendered

***<INTERP>*** = A boolean value representing whether or not interpolation should be used when scaling a texture.

***<SCALEX>*** = The factor by which the width of the texture should be scaled

***<SCALEY>*** = The factor by which the height of the texture should be scaled

### Input

public void update() – Updates the boolean arrays and checks for keyboard and mouse updates

private void keyboardUpdate() – Checks for key events and determines if the key was pressed or released

private void mouseUpdate() – Checks for button events and determines if the button was pressed or released

public static boolean isKey(***<KEY>***) – Determines if a key is down

public static boolean isKeyPressed(***<KEY>***) – Determines if a key was just pressed

public static boolean isKeyReleased(***<KEY>***) – Determines if a key was just released

public static boolean isButton(***<BTN>***) – Determines if a button is down

public static boolean isButtonPressed(***<BTN>***) – Determines if a button was just pressed

public static boolean isButtonReleased(***<BTN>***) – Determines if a button was just released

public static int getMouseX() – Gets the x-coordinate of the mouse’s position

public static int getMouseY() – Gets the y-coordinate of the mouse’s position

public static void setMousePosition(***<X>***, ***<Y>***) – Sets the mouse’s position to the specified coordinates

***<KEY>*** = A key from org.lwjgl.input.Keyboard

***<BTN>*** = A button value from skyEngine.Core.Input

***<X>*** = A coordinate along the x-axis   
***<Y>*** = A coordinate along the y-axis

### SoundClip

public SoundClip(***<PATH>***) – The constructor for the SoundClip class; decodes a 16-bit WAV file

public SoundClip(***<PATH>***, ***<GAIN>***) – The constructor for the SoundClip class; decodes a 16-bit WAV file and sets the gain of the clip

public void startClip() – Plays the audio from the beginning

public void stopClip() – Stops the audio and resets its frame position to the start of the file

public void pauseClip() – Stops the audio and saves the frame position for unpausing

public void unpauseClip() – Plays the audio from the saved frame position

public void togglePause() – Toggles whether or not the audio is paused or not

public void closeClip() – Cleans up SoundClip objects to prevent memory leaks

public void loopClip() – Plays the audio on loop indefinitely

public void setVolume(***<GAIN>***) – Sets the gain of the audio

public float getVolume() – Gets the current gain of the audio

public boolean isClipPaused() – Returns whether or not the audio is paused

***<PATH>*** = The path to the WAV file

***<GAIN>*** = The gain of the audio in dB

### SkyFont

public SkyFont() - The default constructor for SkyFont; it creates a font that uses Times New Roman, plain, 32px

public SkyFont(***<FONT>***, ***<STYLE>***, ***<SIZE>***) - The SkyFont constructor which uses system fonts

public SkyFont(***<PATH>***, ***<SIZE>***) – The SkyFont constructor which uses .ttf files

public void setColor(***<RED>***, ***<GREEN>***, ***<BLUE>***, ***<ALPHA>***) – Sets the color of the font

public void setStyle(***<STYLE>***) – Changes the style of system fonts

public void drawText(***<X>***, ***<Y>***, ***<TEXT>***) – Draws a string on the screen

public float getTextWidth(***<TEXT>***) – Gets the width of a string in pixels

public TrueTypeFont getUseFont() – Gets the font that is being used

public int getFontStyle() – Gets the style of the font

public float getFontSize() – Gets the size of the font

public int getRed() – Gets the amount of red in the font

public int getGreen() – Gets the amount of green in the font

public int getBlue() – Gets the amount of blue in the font

public int getAlpha() – Gets the amount of alpha in the font

public Color getColor() – Gets the color of the font

***<FONT>*** = The name of a system font  
***<STYLE>*** = An integer value for the font style (default fonts) taken from java.awt.Font  
***<SIZE>*** = The height of the font in pixels  
***<PATH>*** = The path to a .ttf file  
***<RED>*** = The amount of red in the font’s color (0 – 255)  
***<GREEN>*** = The amount of green in the font’s color (0 – 255)   
***<BLUE>*** = The amount of blue in the font’s color (0 – 255)  
***<ALPHA>*** = The amount of alpha in the font’s color (0 – 255)  
***<X>*** = A position along the x-axis  
***<Y>*** = A position along the y-axis  
***<TEXT>*** = A string of text

## skyEngine.RenderingUtilities

### AntiAliasing

public static void disableAntiAlias() – Disables anti-aliasing without changing the anti-aliasing variables

public static void enableAntiAlias() – Enables anti-aliasing without changing the anti-aliasing variables

public static void setAntiAliasAlpha(***<ALP>***) – Sets the alpha bits of the PixelFormat for anti-aliasing. Using this method enables anti-aliasing if it is disabled. If the desired value fails, it will be set to the previous value.

public static void setAntiAliasDepth(***<DEP>***) – Sets the depth bits of the PixelFormat for anti-aliasing. Using this method enables anti-aliasing if it is disabled. If the desired value fails, it will be set to the previous value.

public static void setAntiAliasStencil(***<STE>***) – Sets the stencil bits of the PixelFormat for anti-aliasing. Using this method enables anti-aliasing if it is disabled. If the desired value fails, it will be set to the previous value.

public static void setAntiAliasSamples(***<SAM>***) – Sets the number of samples of the PixelFormat for anti-aliasing. Using this method enables anti-aliasing if it is disabled. If the desired value fails, it will be set to the previous value.

public static void setAntiAliasValues(***<ALP>***, ***<DEP>***, ***<STE>***, ***<SAM>***) – Sets all values of the PixelFormat for anti-aliasing. Using this method enables anti-aliasing if it is disabled. If the desired value fails, it will be set to the previous value.

public static void setInitialAntiAliasValues(***<ALP>***, ***<DEP>***, ***<STE>***, ***<SAM>***) – Sets the anti-aliasing values to be used when the game is launched

public static int getAntiAliasAlpha() – Gets the number of alpha bits being used

public static int getAntiAliasDepth() – Gets the number of depth bits being used

public static int getAntiAliasStencil() – Gets the number of stencil bits being used

public static int getAntiAliasSamples() – Gets the number of samples being used

public static PixelFormat getPixelFormat() – Gets the PixelFormat that is being used

***<ALP>*** = The number of alpha bits for antialiasing

***<DEP>*** = The number of depth bits for antialiasing  
***<STE>*** = The number of stencil bits for antialiasing

***<SAM>*** = The number of samples for antialiasing

### VSync

public static void enableVSync() – Enables vertical synchronization

public static void disableVSync() – Disables vertical synchronization

public static void toggleVSync() – Enables vertical synchronization if it is disabled; disables vertical synchronization if it is enabled

public static boolean isVSyncEnabled() – Gets whether or not vertical synchronization is enabled

## skyEngine.Menu

### Button

public Button(***<X>***, ***<Y>***, ***<TEX>***) – Creates an instance of a Button object

public void update() – Updates the Button object

public void render(***<R>***) – Renders the Button object

public boolean isPressed() – Gets if the Button object was just pressed

public boolean isDown() – Gets if the Button is currently down

public float getX() – Gets the x-coordinate of the Button object

public float getY() – Gets the y-coordinate of the Button object

public int getWidth() – Gets the width of the button

public int getHeight() – Gets the height of the button

public Texture getImage() – Gets the texture of the Button object

***<X>*** = The x-coordinate of the upper left corner of the Button object

***<Y>*** = The y-coordinate of the upper left corner of the Button object

***<TEX>*** = The texture of the Button object

***<R>*** = An instance of a renderer object

### Checkbox

public Checkbox(***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***) – Creates an instance of a Checkbox object

public Checkbox(***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***, ***<CHECK>***) – Creates an instance of a Checkbox object

public void update() – Updates the Checkbox object

public void render(***<R>***) – Renders the Checkbox object

public boolean isChecked() – Gets whether the Checkbox object is checked or not

public void setChecked(***<CHECK>***) – Sets whether the Checkbox object is checked or not

public float getX() – Gets the x-coordinate of the Checkbox object

public void setX(***<X>***) – Sets the x-coordinate of the Checkbox object

public float getY() – Gets the y-coordinate of the Checkbox object

public void setY(***<Y>***) – Sets the y-coordinate of the Checkbox object

public int getWidth() – Gets the width of the Checkbox object

public int getHeight() – Gets the height of the Checkbox object

***<X>*** = The x-coordinate of the upper left corner of the Checkbox object

***<Y>*** = The y-coordinate of the upper left corner of the Checkbox object

***<UTEX>*** = The texture for the unchecked Checkbox object

***<CTEX>*** = The texture for the checked Checkbox object

***<CHECK>*** = A boolean value for if the Checkbox object should be checked or not

***<R>*** = An instance of a Renderer object

### RadioButton

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***) – Creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position; uses the default gap between Checkbox objects (half of the height) and a vertical orientation

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<UTEX>***, ***<CTEX>***, ***<HORIZ>***) – Creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position; orientation is taken as a parameter and the gap between Checkbox objects is the default (half of the height or width depending on orientation)

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<GAP>***, ***<UTEX>***, ***<CTEX>***) – Creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position; the gap between Checkbox objects is passed in as a parameter and the orientation is vertical

RadioButton(***<QTY>***, ***<X>***, ***<Y>***, ***<GAP>***, ***<UTEX>***, ***<CTEX>***, ***<HORIZ>***) – Creates an instance of a RadioButton object with the specified number of Checkbox objects at a specified position; the gap between Checkbox objects and the orientation are passed in as parameters

private void initializeRadioButtons() – Creates the ArrayList of Checkbox objects

public void update() – Updates the RadioButton object

public void render(***<R>***) – Renders the RadioButton object

public void setChecked(***<INDEX>***) – Sets which Checkbox object should be checked

public int getSelectedIndex() – Gets the index of the checked Checkbox in the RadioButton object

public void lock() – Locks the RadioButton object so that one Checkbox must always be checked

public void unlock() – Unlocks the RadioButton object so that it is possible for none of the Checkbox objects to be checked

public int getQuantity() – Gets the number of Checkboxes used to make the RadioButton object

public float getX() – Gets the position of the RadioButton object along the x-axis

public float getY() – Gets the position of the RadioButton object along the y-axis

public float getGap() – Gets the gap between Checkbox objects

public boolean isLocked() – Gets whether or not the RadioButton object is locked

public boolean isOneChecked() – Gets whether one of the Checkboxes is checked or not

***<QTY>*** = The number of Checkboxes to make the RadioButton object with

***<X>*** = The x-coordinate of the upper left corner of the first Checkbox object

***<Y>*** = The y-coordinate of the upper left corner of the first Checkbox object

***<UTEX>*** = The texture for the unchecked Checkbox object

***<CTEX>*** = The texture for the checked Checkbox object

***<HORIZ>*** = A boolean representing if the RadioButton object should be oriented horizontally

***<GAP>*** = The gap between Checkbox objects in pixels

***<R>*** = An instance of a Renderer object

***<INDEX>*** = The index of a Checkbox object in the ArrayList

## skyEngine.Inventory

### AbstractItem

getQuantity() – Gets the quantity of the item

setQuantity(***<QTY>***) – Sets the quantity of the item

addToQuantity(***<AMT>***) – Adds an amount to the quantity

subtractFromQuantity(***<AMT>***) – Subtracts an amount from the quantity

getMaxQuantity() – Gets the maximum quantity of the item

setMaxQuantity(***<MAX>***) – Sets the maximum quantity of the item

getName() – Gets the name of the item

setName(***<NAME>***) – Sets the name of the object

getItemID() – Gets the itemID value of the item

setItemID(***<ID>***) – Sets the itemID value of the item

***<QTY>*** = The quantity of the item

***<AMT>*** = The amount to be added to or subtracted from the item

***<MAX>*** = The maximum quantity of the item

***<NAME>*** = The name of the item

***<ID>*** = The itemID value of the item

### StaticInventory

public StaticInventory(***<SIZE>***) – Creates a StaticInventory object of a specified size

public void addItem(***<ITEM>***) – Adds an item to the first available spot if possible

public void removeItem(***<ITEM>***) – Removes all instances of an item from the inventory; it is recommended to use the method with an itemID value as the parameter

public void removeItem(***<ID>***) – Removes all instances of items that have a specific itemID value; the recommend method for removing objects from the inventory

public void removeItem(***<NAME>***) – Removes all instances of items that have a specific name; it is recommended to use the method with an itemID value as the parameter

public void removeItemAtIndex(***<INDEX>***) – Removes the item from a certain index if an item exists there

public void clearInventory() – Removes all items from the inventory

public AbstractItem getItemAtIndex(***<INDEX>***) – Returns the item found at a specific index if the index is valid

public void setItemAtIndex(***<ITEM>***, ***<INDEX>***) – Sets the item at a specific index to a specified AbstractItem if the index is valid

public boolean containsItem(***<ITEM>***) – Checks to see whether an item exists within the inventory or not; it is recommended to use an itemID value to test for a certain item

public boolean containsItem(***<ID>***) – Checks to see whether an item with a specific itemID value exists within the inventory or not; the recommended way to check for an item

public boolean containsItem(***<NAME>***) – Checks to see whether an item with a specific name exists within the inventory or not; it is recommended to use an itemID value to test for a certain item

public int indexOfItem(***<ITEM>***) – Gets the index of the first occurrence of a specific item in the inventory if it exists; it is recommended to use an itemID value to get the index of an item

public int indexOfItem(***<ID>***) – Gets the index of the first occurrence of an item with a specific itemID value in the inventory if it exists; the recommended way to get the index of an item

public int indexOfItem(***<NAME>***) – Gets the index of the first occurrence of an item with a specific name in the inventory if it exists; it is recommended to use an itemID value to get the index of an item

public int getSize() – Gets the size of the inventory

public void setSize(***<SIZE>***) – Sets the size of the inventory to a new value

public void moveEmptyToEnd() – Moves all empty spots in the inventory to the end

public void sortByQuantityAscending() – Sorts the inventory by quantity from low to high

public void sortByQuantityDescending() – Sorts the inventory by quantity from high to low

public void sortByNameAscending() – Sorts the inventory by name from A to Z

public void sortByNameDescending() – Sorts the inventory by name from Z to A

public void sortByIDAscending() – Sorts the inventory by itemID from low to high

public void sortByIDDescending() – Sorts the inventory by itemID from high to low

public AbstractItem[] getInventory() – Gets the entire inventory

public void setInventory(***<INVENT>***) – Sets the inventory to an array of AbstractItem objects

***<SIZE>*** = The size of the inventory

***<ITEM>*** = An AbstractItem object

***<ID>*** = An itemID value

***<NAME>*** = The name of an item

***<INDEX>*** = An index of the inventory

***<INVENT>*** = An array of AbstractItem objects

### DynamicInventory

public void addItem(***<ITEM>***) – Adds an item to the end of the inventory

public void removeItem(***<ITEM>***) – Removes all instances of an item from the inventory; it is recommended to use the method with an itemID value as the parameter

public void removeItem(***<ID>***) – Removes all instances of items that have a specific itemID value; the recommend method for removing objects from the inventory

public void removeItem(***<NAME>***) – Removes all instances of items that have a specific name; it is recommended to use the method with an itemID value as the parameter

public void removeItemAtIndex(***<INDEX>***) – Removes the item from a certain index if an item exists there

public void clearInventory() – Removes all items from the inventory

public AbstractItem getItemAtIndex(***<INDEX>***) – Returns the item found at a specific index if the index is valid

public void setItemAtIndex(***<ITEM>***, ***<INDEX>***) – Sets the item at a specific index to a specified AbstractItem if the index is valid

public void insertItemAtIndex(***<ITEM>***, ***<INDEX>***) – Inserts an AbstractItem at a specific index if the index is valid

public boolean containsItem(***<ITEM>***) – Checks to see whether an item exists within the inventory or not; it is recommended to use an itemID value to test for a certain item

public boolean containsItem(***<ID>***) – Checks to see whether an item with a specific itemID value exists within the inventory or not; the recommended way to check for an item

public boolean containsItem(***<NAME>***) – Checks to see whether an item with a specific name exists within the inventory or not; it is recommended to use an itemID value to test for a certain item

public int indexOfItem(***<ITEM>***) – Gets the index of the first occurrence of a specific item in the inventory if it exists; it is recommended to use an itemID value to get the index of an item

public int indexOfItem(***<ID>***) – Gets the index of the first occurrence of an item with a specific itemID value in the inventory if it exists; the recommended way to get the index of an item

public int indexOfItem(***<NAME>***) – Gets the index of the first occurrence of an item with a specific name in the inventory if it exists; it is recommended to use an itemID value to get the index of an item

public int getSize() – Gets the size of the inventory

public void sortByQuantityAscending() – Sorts the inventory by quantity from low to high

public void sortByQuantityDescending() – Sorts the inventory by quantity from high to low

public void sortByNameAscending() – Sorts the inventory by name from A to Z

public void sortByNameDescending() – Sorts the inventory by name from Z to A

public void sortByIDAscending() – Sorts the inventory by itemID from low to high

public void sortByIDDescending() – Sorts the inventory by itemID from high to low

public ArrayList<AbstractItem> getInventory() – Gets the inventory

public void setInventory(ArrayList<AbstractItem> inventory) – Sets the inventory to an ArrayList of AbstractItem objects

***<SIZE>*** = The size of the inventory

***<ITEM>*** = An AbstractItem object

***<ID>*** = An itemID value

***<NAME>*** = The name of an item

***<INDEX>*** = An index of the inventory

***<INVENT>*** = An array of AbstractItem objects

# **License**

Copyright (c) 2016-present, Adam Petro

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARANTIES OF MERCHANTABLITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABLITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.