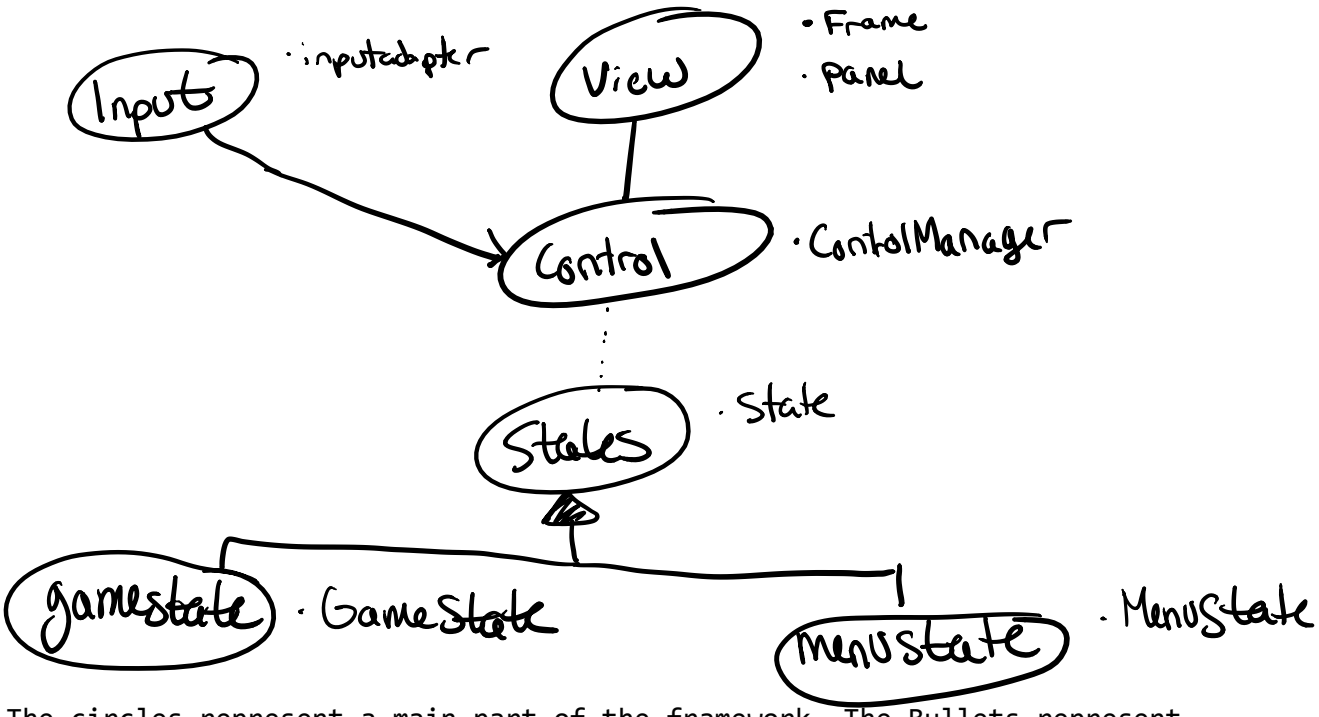
Development in Menus

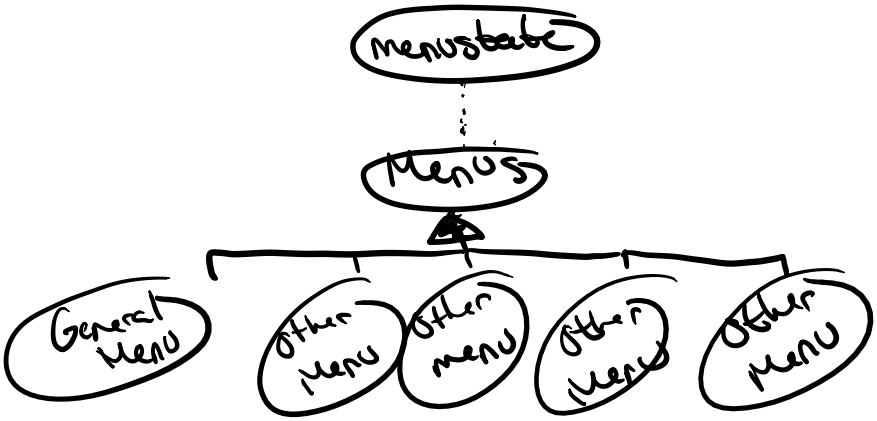
Basheer B.

I will explain how the menu framework is setup now. Also, there will be some things you will need to learn on your own that is not described in this document.

Before we get to the actual coding part, it is crucial that you first understand how the menu framework works. Here is a simplified diagram of the framework for the whole game.



The circles represent a main part of the framework. The Bullets represent what classes is in that part. The “view” is for displaying all the information to the window. The “control” is responsible for the direction of the whole game. The “states” represent a certain part of the game. They are self-described. The beauty of this framework is the ease of sending information down this hierarchy. We will talk about this more later. As you can see, the states (menustate and gamestate) is directly instantiated within the ControlManager. We will look at the code for that later as well. Now we will extend that diagram, but we will only zoom in on the menustate part since that is what you will be working on.



So the MenuState will have a list of “Menus”, (Ex: GeneralMenu, PlayerMenu, MapMenu, etc.). The MenuState is responsible for changing the menus. The specific menus will do its own duties. They will consist of a list of options and will react when an option is selected (Ex: Play, Exit, etc.). We will look more into this as we get into the code. Really, to simplify the whole understanding of this framework, it is best that you are oblivious to all the framework in the game except the menu framework. The only exception to this is if there is an option to change the states (from menustate to the actual gamestate), then you will need to know a little about the control framework. Now let’s get into the actual code!

```
public interface State {

	// init()
	// Method that creates the initial objects and other tasks
	public void init();

	// draw()
	// Method that draws the state's objects to the panel
	public void draw(Graphics g);

	// update()
	// Method that updates all of the state's objects
	public void update();

	// keyPressed()
	// Method that receives the input from the control manager
	// Then is sent to the state's objects
	public void keyPressed(KeyEvent k);

	// keyPressed()
	// Method that receives the input from the control manager
	// Then is sent to the state's objects
	public void keyReleased(KeyEvent k);

	//TODO: Implement mouse input methods if needed
}
```

This is the State interface. What this class does is describe the methods that a state MUST have. If you don’t know what an interface is, please look it up. As you can see, a state must have a draw, update, and keyinput methods. This guarantees that if I want to call a method in a “state”, then I know for sure that it has one of these methods. In the Control Manager, it can just call update() method on the current state, then that state has an update() method that then calls the update methods of all the characters and models. That is the beauty of this framework. A simple update method in the Control can fluently call the updates of all models. This applies to keyinput as well. In the Control Manager, I just sent the KeyEvent k (which is the integer value of a key input), and I can send it down the framework to a specific character. The character is in charge of handling that information and reacting to it. Applies to the draw method as well. Let’s look in the actual MenuState now.

```java
public class MenuState implements State {
	//
	//
	// List of all menu screens
	private ArrayList<Menu> menus;
	//
	//
	// Index of the current menu
	protected int currentMenu = 0;
	//
	//
	// Index of each menu in the list
	// USE THESE CONSTANTS TO YOUR ADVANTAGE
	public static final int GENERALMENU = 0;
	public static final int OPTIONMENU = 1;
	public static final int PLAYERMENU = 2;
	public static final int MAPMENU = 3;
	public static final int FINALMENU = 4;

	// Constructor: Calls the init() method which handles all the construction
	public MenuState() {
		init();
	}

	// init(): Initializes the menu list, adds menus, set current menu
	public void init() {
		menus = new ArrayList<Menu>();
		menus.add(GENERALMENU, new GeneralMenu());
		currentMenu = GENERALMENU;
	}

	// setMenu()
	// PRECONDITION: currentMenu in menu list must be non-null
	// POSTCONDITION: The preffered menu is initialized
	public void setMenu(int menuIndex) {
		// Change the currentMenu to the preferred menu index (menuIndex)
		currentMenu = menuIndex;
		// Initialize the new menu
		menus.get(currentMenu).init();
	}

	// draw()
	// PRECONDITION: Must have a graphics object and a non-null menu in the
	// index
	// POSTCONDITION: Sends the graphics to the current menus draw method
	// All drawing is done there, this method just sends it from above
	public void draw(Graphics g) {
		// Sends the graphics to the current menu's draw method
		menus.get(currentMenu).draw(g);
	}
```

```
	// update()
	// PRECONDITION: The current menu must have a working update method. The
	// index must not be null in the menu list
	// POSTCONDITION: Calls the current menu's update method. This method
	// is called from the control manager
	public void update() {
		// Calls the current menu's update method
		menus.get(currentMenu).update();
	}

	// keyPressed()
	// PRECONDITION: Provide a KeyEvent, currentMenu must have a working
	// keyPressed method. The index must not be null in the menu list
	// POSTCONDITION: Calls the current menu's keyPressed method. This method
	// is called from the control manager
	public void keyPressed(KeyEvent k) {
		// Sends the KeyEvent to the currentMenus keyPressed method
		menus.get(currentMenu).keyPressed(k);
	}

	// keyReleased()
	// PRECONDITION: Provide a KeyEvent, currentMenu must have a working
	// keyPressed method. The index must not be null in the menu list
	// POSTCONDITION: Calls the current menu's keyReleased method. This method
	// is called from the control manager
	public void keyReleased(KeyEvent k) {
		// Sends the KeyEvent to the currentMenus keyReleased method
		menus.get(currentMenu).keyReleased(k);
	}
}
```

So here is the class for the Menu State. As you can see, this class **implements** State which means that this is described as a State class, hence, all the methods that a State should have will be forced to be implemented here. In the instance variables, it contains a list of Menus and the index of all the Menus just for ease of changing between them. Take a look at the comments, they will explain each action specifically. What the MenuState is in charge of is sending information to a specific menu, and changing to and from another specific menu. If you have an option to change to another menu, you can call the setMenu() method to the appropriate index. Now I created one menu just as an example called General Menu. But first, let's look at another interface called Menu.

```
public interface Menu {

	// Initiate the menu
	public void init();

	// Update and information within the menu
	public void update();

	// Draw whatever is on the menu such as text, images, players, etc
	// This is where the design comes into play
	public void draw(Graphics g);

	// Select any options and transporting any data or other changes...
```

```
	public void select(int index);

	// Method to receive keyPressed inputs
	public void keyPressed(KeyEvent k);

	// Method to receive keyReleased inputs
	public void keyReleased(KeyEvent k);

	// TODO: Add mouse input methods if needed
}
```

This works the same way as the State interface. These are methods that a Menu must have. As you can see, there are some of the methods that are identical to the methods in the State class such as the keyInput methods, the draw method, and the update method. This is for what I described before. The framework is designed to send information down the hierarchy, so in order for that to take place, all the models (or menus in this case, could be characters in the gamestate) should have these important methods. Let's take a look at the GeneralMenu which implements the Menu interface.

```
public class GeneralMenu implements Menu {

	protected int currentChoice = 0;
	protected ArrayList<String> options;

	private static final int STARTINDEX = 0;
	private static final int OPTIONSINDEX = 1;
	private static final int EXITINDEX = 2;
	private static final int GAMESTATE = 3;

	private BufferedImage background;
	private Color titleColor;
	private Font titleFont;

	private int textSpace;

	public GeneralMenu() {
		super();
		options = new ArrayList<String>();
		options.add(STARTINDEX, "START");
		options.add(OPTIONSINDEX, "OPTIONS");
		options.add(EXITINDEX, "EXIT");
		// options.add(GAMESTATE, "GAMESTATE");

		textSpace = 50;
	}

	public void init() {

	}

	public void update() {

	}

	public void draw(Graphics g) {
```

```
		// iterate through "options" and draw the string (with current option
		// highlighted
		// use the color and font variables
		// draw background images
		for (int i = 0; i < options.size(); i++) {
			if (i == currentChoice) {
				g.setColor(Color.RED);
				// font change or any other emphasis
			} else {
				g.setColor(Color.WHITE);
			}

			g.drawString(options.get(i), Panel.WIDTH / 2, Panel.HEIGHT
					* (3 / 2) + (i * textSpace));
		}
	}

	public void select(int currentChoice) {
		// OVERRIDE THIS, BUT HERE IS A SIMPLE TEMPLATE
		if (currentChoice == 0) { // START
			Panel.control.setState(Panel.control.GAMESTATE);
		} else if (currentChoice == 1) { // OPTIONS
			// do something
		} else if (currentChoice == 2) { // EXIT
			System.exit(0);
		} else if (currentChoice == 3) { // CHANGEGAMESTATE

		}
	}

	public void keyPressed(KeyEvent k) {
		// OVERRIDE THIS, BUT HERE IS A SIMPLE TEMPLATE
		switch (k.getKeyCode()) {
		case KeyEvent.VK_ENTER:
			select(currentChoice);
			break;
		case KeyEvent.VK_UP:
			// TODO:Didn't know how to implement this with modulos
			currentChoice = (currentChoice <= 0) ? options.size() - 1
					: currentChoice - 1;
			break;
		case KeyEvent.VK_DOWN:
			currentChoice = (currentChoice + 1) % options.size();
			break;
		// No default unless stating an exception
		}
	}

	public void keyReleased(KeyEvent k) {

	}

}
```

If you take a look at the interface variables, you will see a list of options:

```
protected int currentChoice = 0;
protected ArrayList<String> options;

private static final int STARTINDEX = 0;
private static final int OPTIONSINDEX = 1;
private static final int EXITINDEX = 2;
private static final int GAMESTATE = 3;
```

You have a currentChoice representing which option is highlighted. You have an ArrayList of strings of options. You also have a list of indexes for each specific option. This is kind of the same as keeping the indexes of menus inside the MenuState class. So if you wanted to add a new option called “Select Player”, you will create that index in here, this is how you will actually add the option inside the list:

```
options = new ArrayList<String>();
options.add(STARTINDEX, "START");
options.add(OPTIONSINDEX, "OPTIONS");
options.add(EXITINDEX, "EXIT");
```

This is inside the init() method. As you can see, when adding an option, you first start off with the index of the option, then in the second argument, you describe the title of the option. This is why it is a list of Strings. Let’s take a look on how the drawing works:

```
public void draw(Graphics g) {
	// iterate through "options" and draw the string (with current
	option
	// highlighted
	// use the color and font variables
	// draw background images
	for (int i = 0; i < options.size(); i++) {
		if (i == currentChoice) {
			g.setColor(Color.RED);
			// font change or any other emphasis
		} else {
			g.setColor(Color.WHITE);
		}

		g.drawString(options.get(i), Panel.WIDTH / 2, Panel.HEIGHT
				* (3 / 2) + (i * textSpace));
	}
}
```

So first, we will need to iterate through all the options from 0 to the list.size(). If the option is a currentChoice, we want some kind of emphasis to determine that, so for now, I just changed the color to red. If you wanted a cooler design, you could change the size or anything else. In the future, I am hoping to add images instead of displaying a text, but that is just a feature, we need to first get the framework down. If it is not a currentChoice, we will just make it plain and be white in color. Then we will draw the string on the screen and do a little math to calculate the position of where it should be place. I used the “i” variable to determine the offset

of position. As you can see, the x placement will be the same for all options since it is always Panel.*WIDTH* / 2, but if you take a look at the y placement, it is affected by the (i * textSpace)) which textSpace is defined in the instance variables. Now where the actual actions come into place. Let's look at the keyInput first to understand how to choose the currentChoice as well as wrapping which I will explain.

```
public void keyPressed(KeyEvent k) {
		// OVERRIDE THIS, BUT HERE IS A SIMPLE TEMPLATE
		switch (k.getKeyCode()) {
		case KeyEvent.VK_ENTER:
			select(currentChoice);
			break;
		case KeyEvent.VK_UP:
			// TODO:Didn't know how to implement this with modulos
			currentChoice = (currentChoice <= 0) ? options.size() - 1
					: currentChoice - 1;
			break;
		case KeyEvent.VK_DOWN:
			currentChoice = (currentChoice + 1) % options.size();
			break;
		// No default unless stating an exception
		}
	}

	public void keyReleased(KeyEvent k) {

	}

}
```

We will use a switch statement which you might want to look up for more detail. A switch statement pretty much takes an input as a non-constant variable (in this case: k.getKeyCode()) Then it will choose if it equals one of the **cases**. So we will take in the input received from the argument. If the input equals KeyEvent.*VK_ENTER* or in other words, we press enter (the method is, after all, keyPressed()), Then we will do the actions until it reaches the **break statement**. So if Enter is pressed, we will call the select() method with the currentChoice variable. We will look at the select() method later. If we press up, we will move the current choice to the top seletion. I did a bit of wrap-around in case the currentChoice is the top-most selection, and I want UP to go all the way down to the lower-most selection. Same applies for pressing Down. Let's take a look at when a user pressed Enter, the select() method.

```
	public void select(int currentChoice) {
		// OVERRIDE THIS, BUT HERE IS A SIMPLE TEMPLATE
		if (currentChoice == 0) { // START
			Panel.control.setState(Panel.control.GAMESTATE);
		} else if (currentChoice == 1) { // OPTIONS
			// do something
		} else if (currentChoice == 2) { // EXIT
			System.exit(0);
		} else if (currentChoice == 3) { // CHANGEGAMESTATE

		}
```

```
    }
```
So this will take in the currentChoice and if it equals the index of a specific purpose, it will to the action. For example, If we press index 2 (which is Exit), I called System.exit(0) which terminates the program. And if we choose Start, It changes the state to the gamestate. Really, I just noticed a clearer way to implement this. Instead of setting it equal to a random number, it would be better to set it equal to the indexes we made above.

So that is all about the menus, what you can do now is work on all the separate Menus such as this GeneralMenu (which is the startup), the PlayerMenu (to choose a player), etc. Right now, the GeneralMenu as the options Start, Options, and Exit. Start should instead go to the player selection menu instead of starting right away. All people working on menus should be collaborating together otherwise it will become messy. Talk to me if you need any help on where to start. If you haven't, I highly suggest you look at the other document about GitHub branching. The first section of that document is just some concepts about GitHub, but the second section is how to setup your own branches. Good Luck!