Welcome to the Course

Thank you for joining this course!

Before we begin, these are the basic requirements for this course:

Prior knowledge

In this course, we will be creating a 2D turn-based RPG using the Godot 4 game engine. This game will allow our players to fight an enemy, where the player can choose their attacks, and the enemy will use a simple AI to fight back.

Course Requirements

In order to easily follow along with this course, you should already have:

- A basic understanding of the Godot editor.
- A basic understanding of the GDScript language.

However, you don't need to be overly proficient in either one. We will be going through each process of creating the game, step by step, especially when it comes to the scripting side of things. Of course, the more you know before starting, the easier to follow along, but only basic knowledge is required.

Project Features

Throughout this project, we will cover various interesting features available in the Godot Engine, including:

- **User Interfaces**, which will allow us to place elements like text and health bars on the screen.
- **Signals**, which will be used to trigger and listen to events.
- **Custom Resources**, which will allow us to create custom asset types and store them on the disk for use in the game.

These features will allow us to create our combat actions and assign them to the player and enemy AI.

Course Breakdown

The content in this course can be broken down into multiple sections:

1. We will begin by creating our player and enemy characters. These will have a health bar, and be able to heal and take damage.
2. Secondly, we will work on the combat actions. We will store these in custom resources and will have the ability to heal and deal damage, whilst being custom to each character.
3. We will then need to set up a UI for the player to allow us to use these custom resources in the game.
4. Finally, we will be focused on creating the enemy AI. This will allow them to choose what action to take, based on their current health percentage.

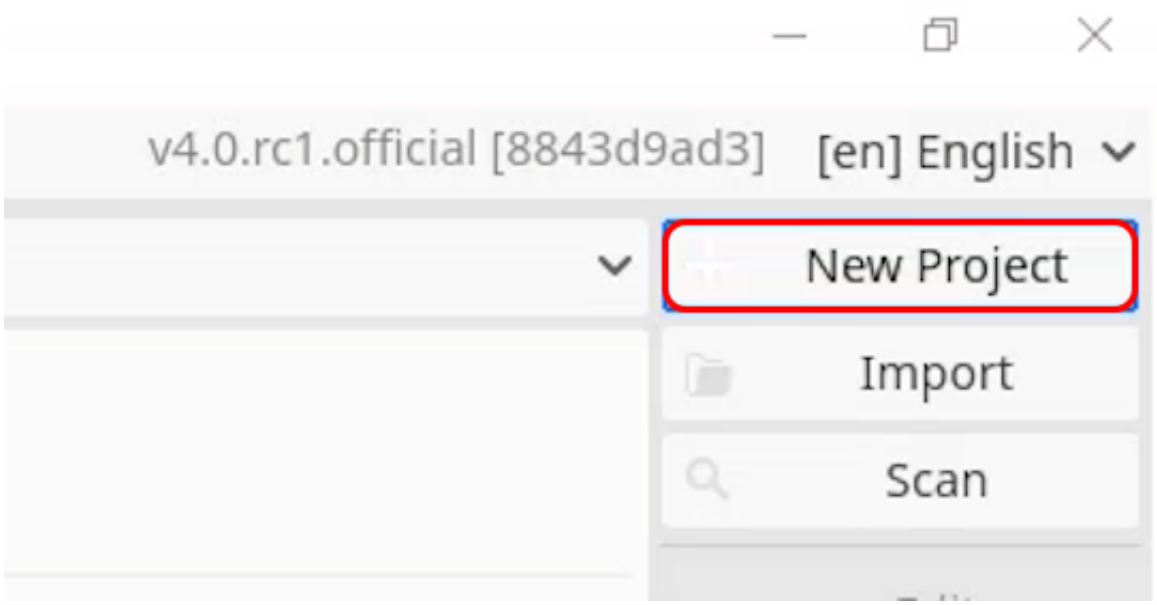
About Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the included course files.

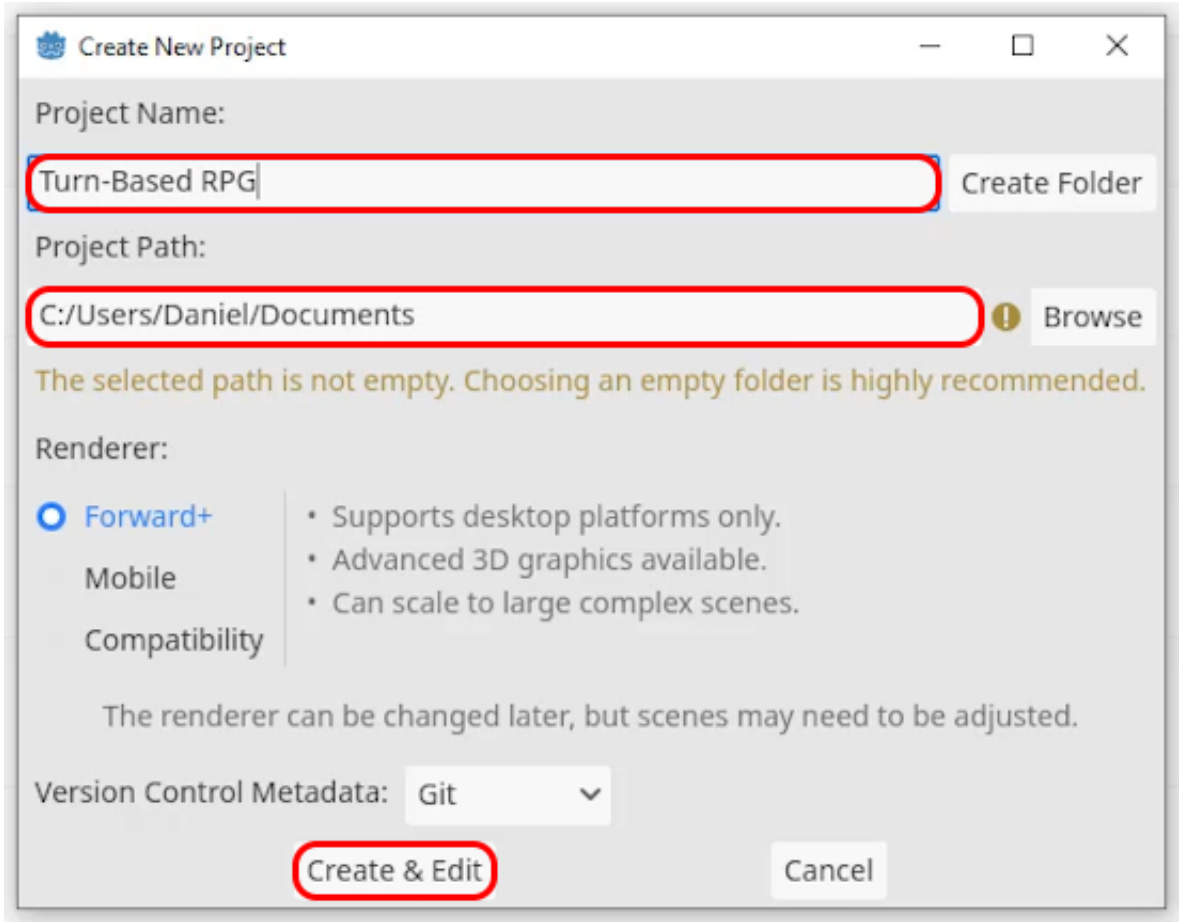
Godot 4.0

This course was made using version 4.0 of the Godot engine. If you are encountering any issues with the course material, that may be due to your version of the engine. Many of the features explored in this course are different or missing from previous Godot versions and may also be different in future versions.

In this lesson, we'll kick off by creating our project and importing the assets we need. To begin with, open up Godot into the *Project Manager.* We will start by pressing the **New Project** button.

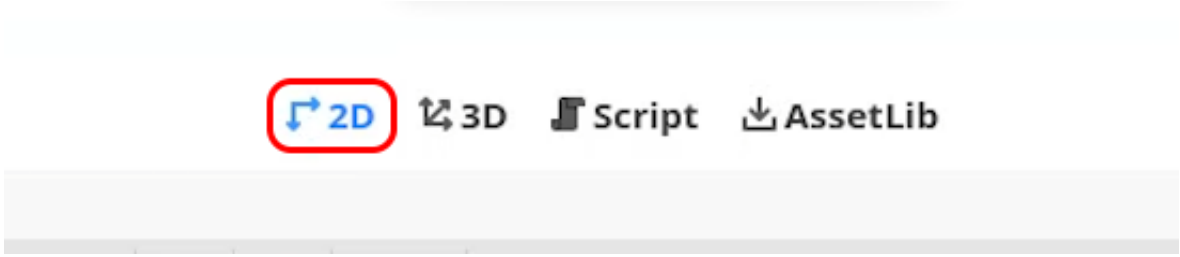


Choose a name for the project, we will call ours *Turn-Based RPG* and select a *Project Path*. If your path isn't empty, press the *Create Folder* button to create a new folder with the project name. With that all in place, hit **Create & Edit** to create the project.

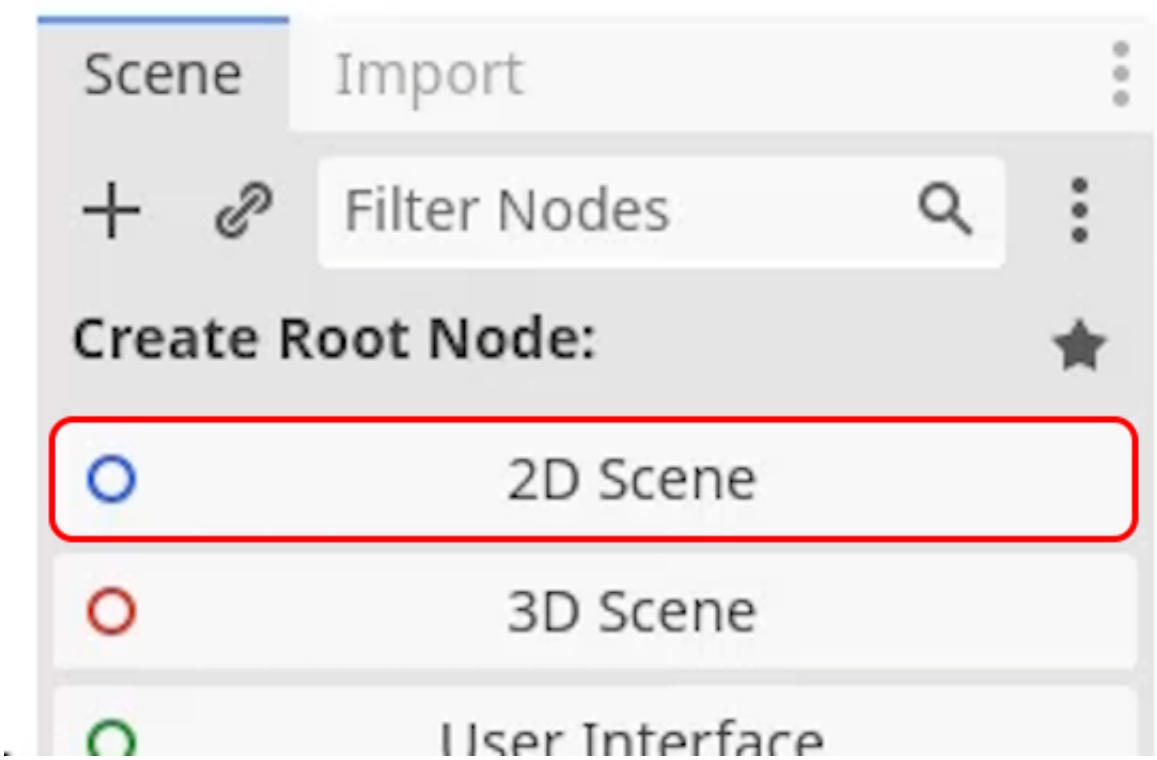


Creating the First Scene

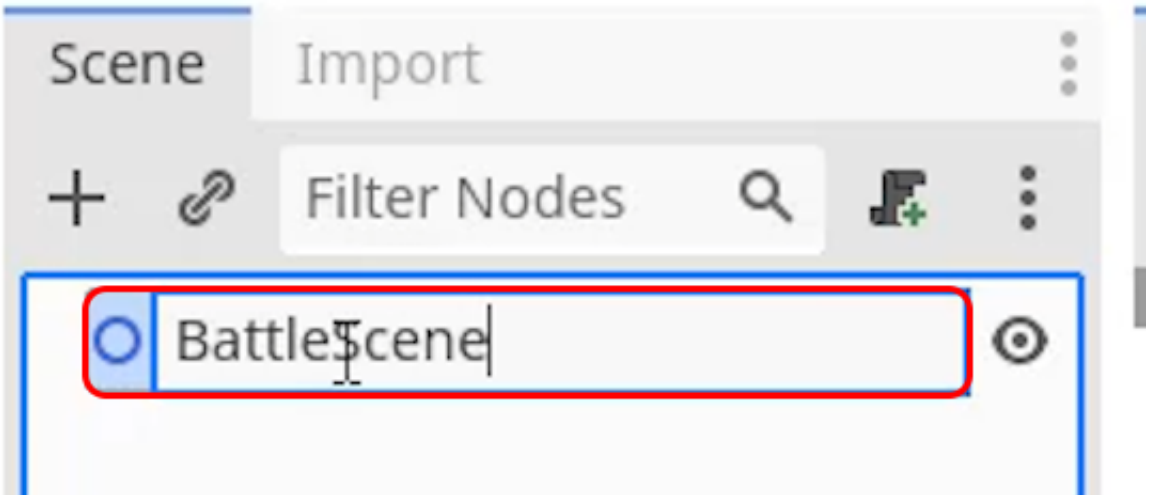
To begin with, we will switch the editor to the **2D Mode** as we will be developing a 2D game.



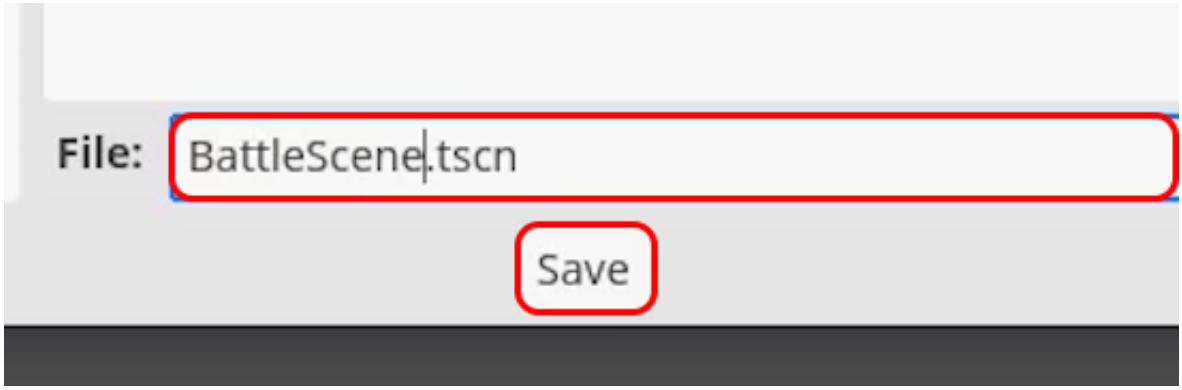
We will choose **2D Scene** to create the root node of our scene.



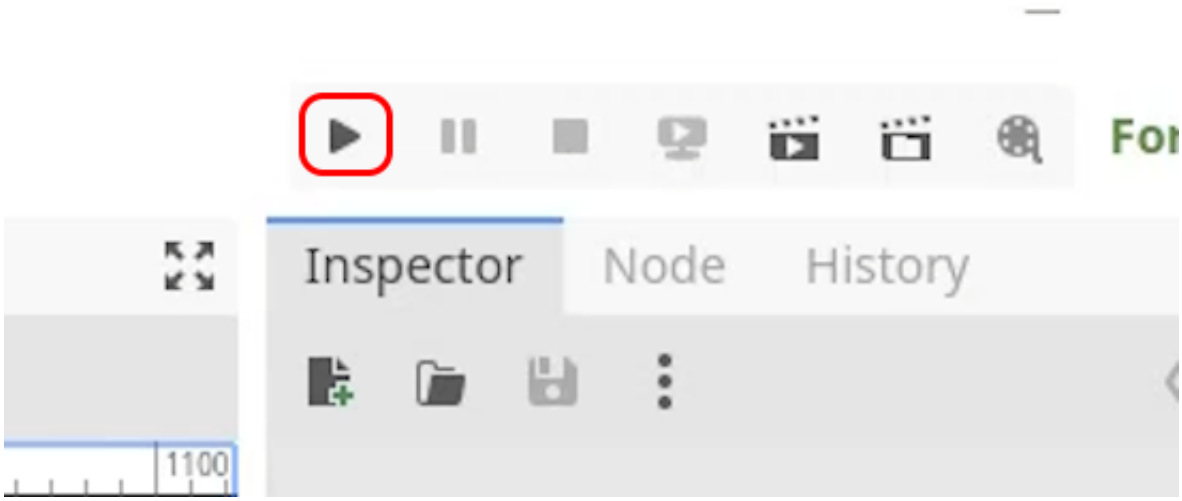
To make things easy to recognize, we can **rename** the root node to *BattleScene*.



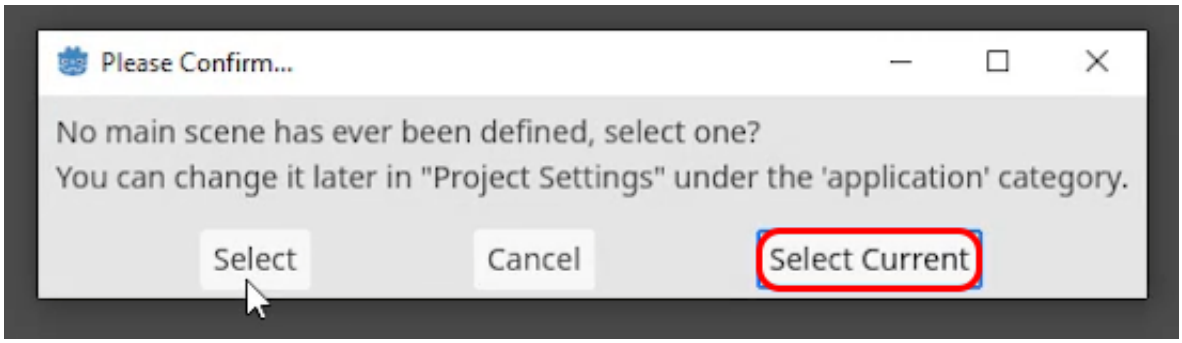
This sets up our scene for adding any nodes as children, such as characters, cameras, or our UI. We can now **save** the scene (**CLTR+S**) and save the file as *BattleScene.tscn*.



Finally, press the **Play** button in the top right of the Godot Editor.

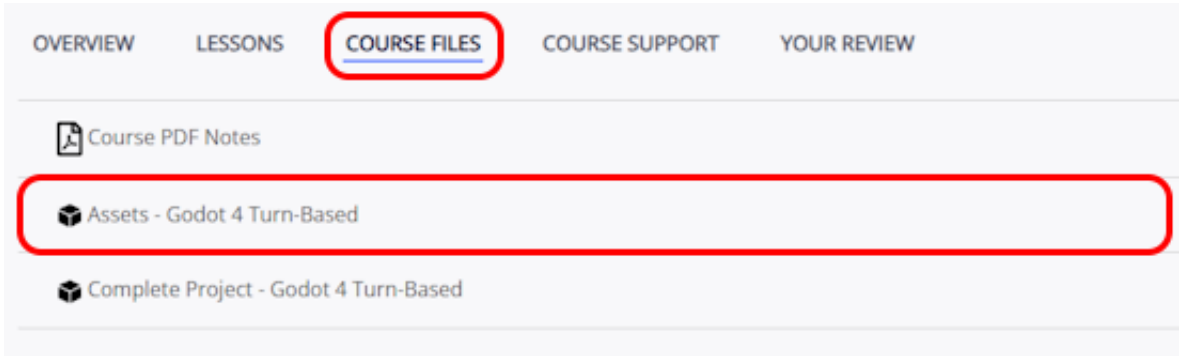


This will allow us to choose **Select Current** for the main scene of the game.

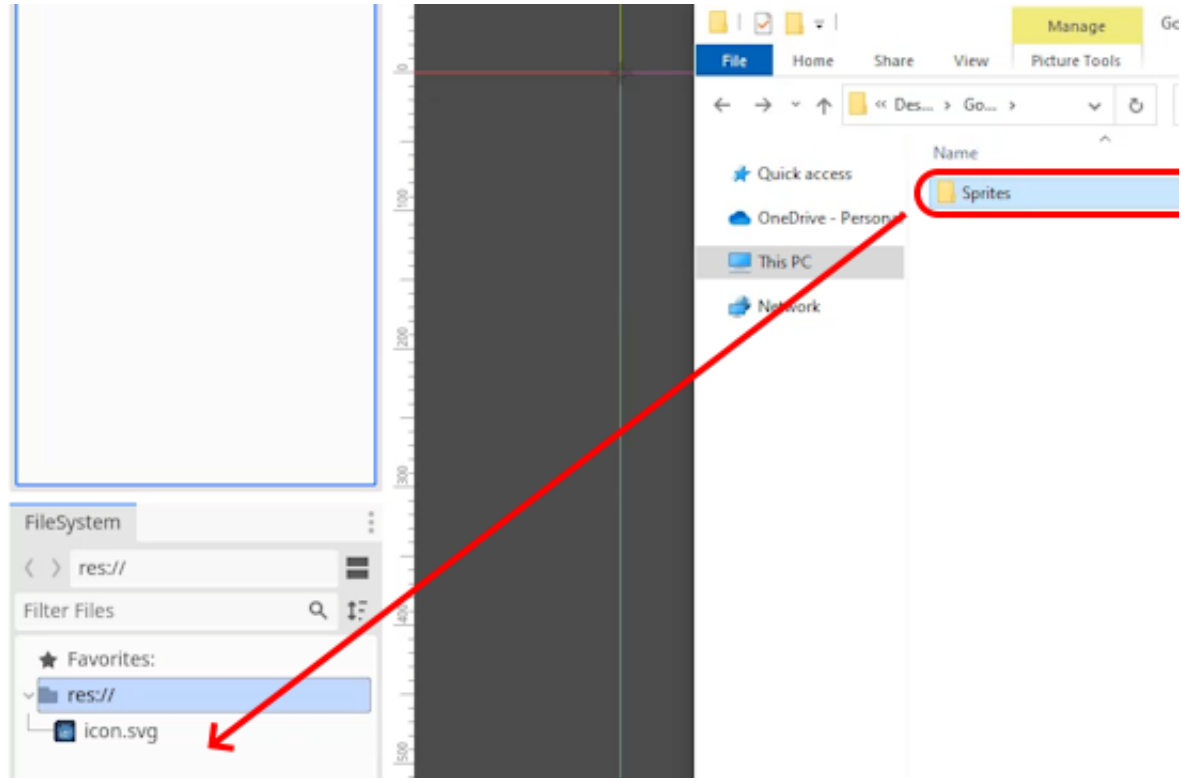


Setting Up the Assets

We also need to set up our assets to use throughout the game. You can of course use your own, however, in the **Course Files** section on *Zenva* we have supplied a zip of all the assets we will be using throughout the lessons.

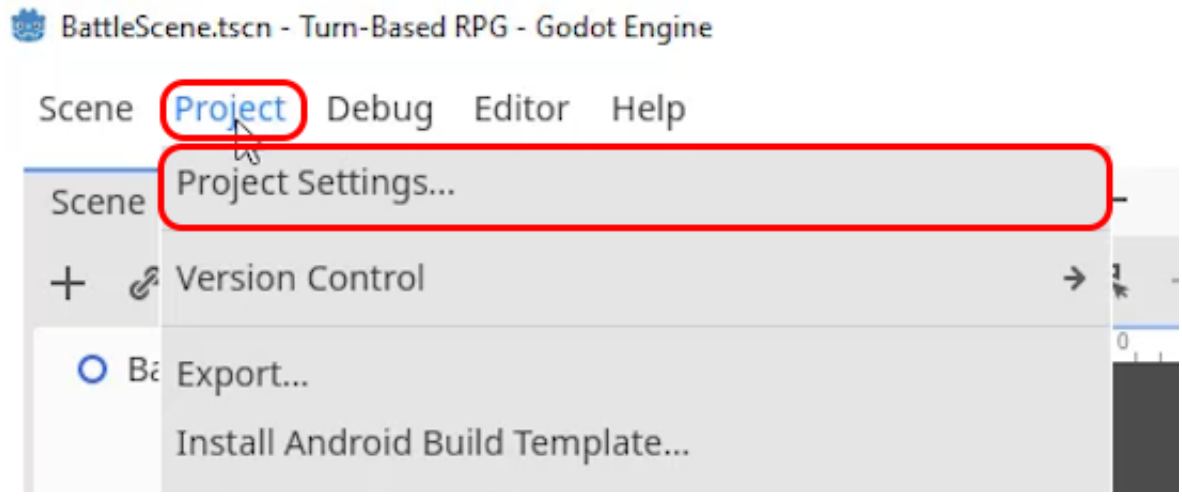


Inside the assets zip file are 4 sprites that we can use as the characters in our game. To add them to our Godot project, simply click and drag the folder into the *FileSystem* window.

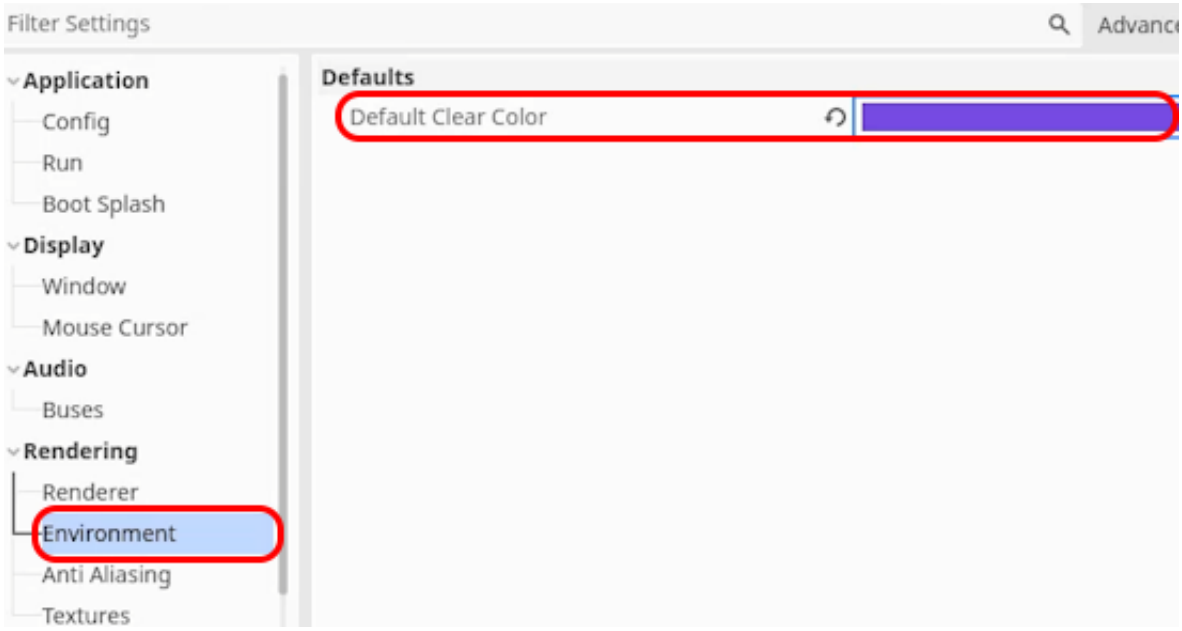


Changing the Background Color

As a final thing, we likely don't want this dim-grey color for our game's background. To change this, we will go to the **Project Settings** window.

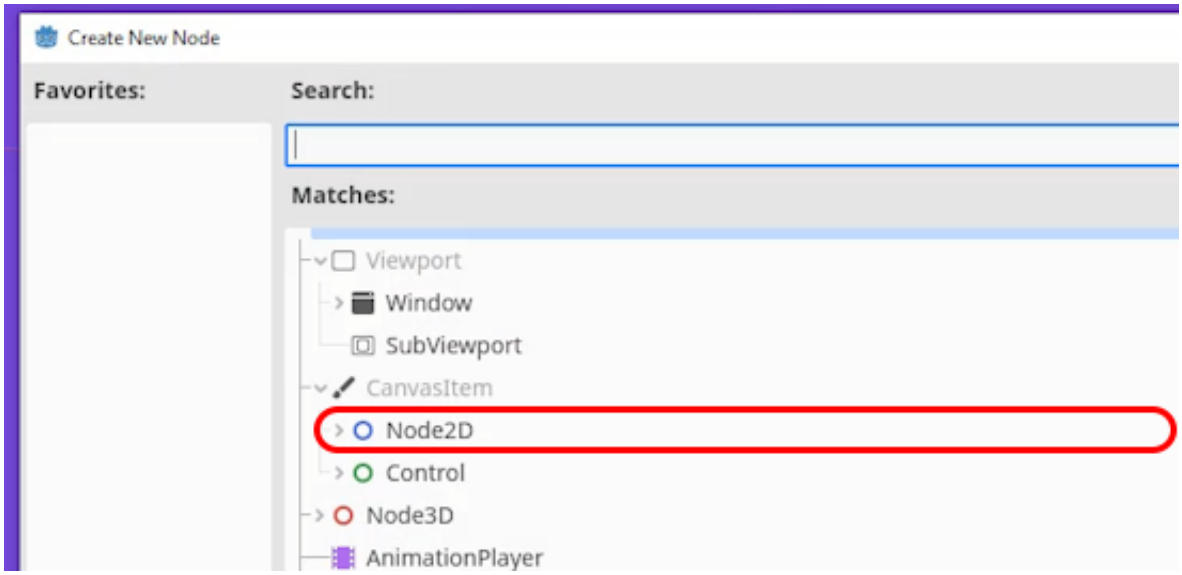


From here, we will select the **Environment** tab, and change the **Default Clear Color** value.

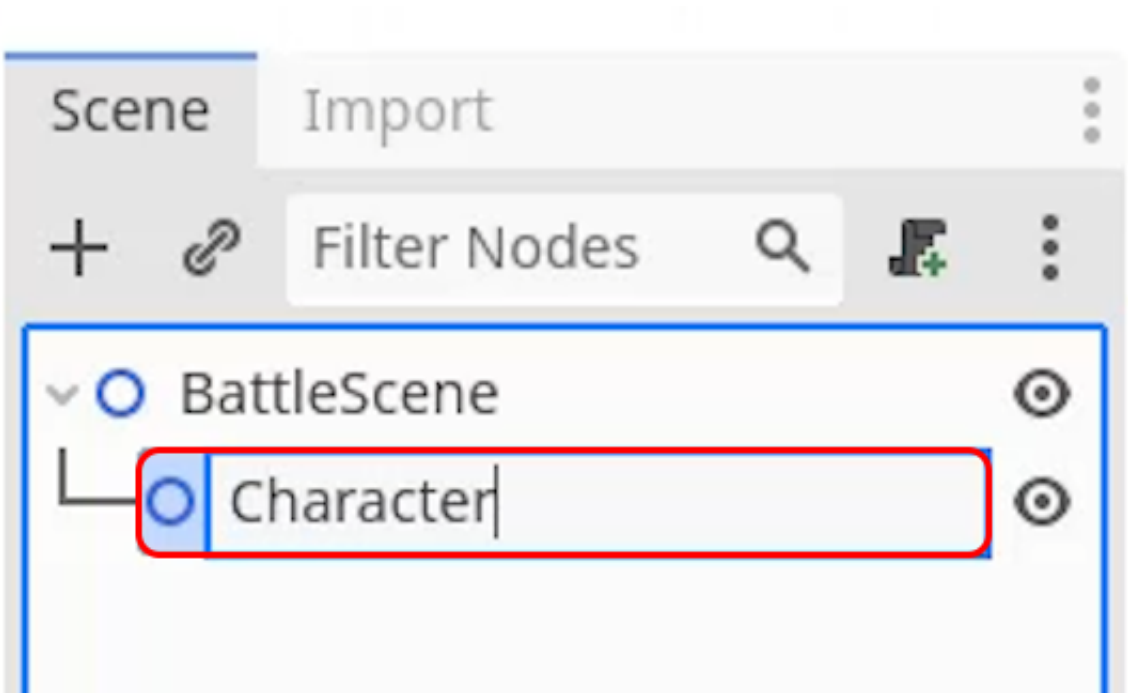


With this done, we are ready to begin setting up our characters in the next lesson.

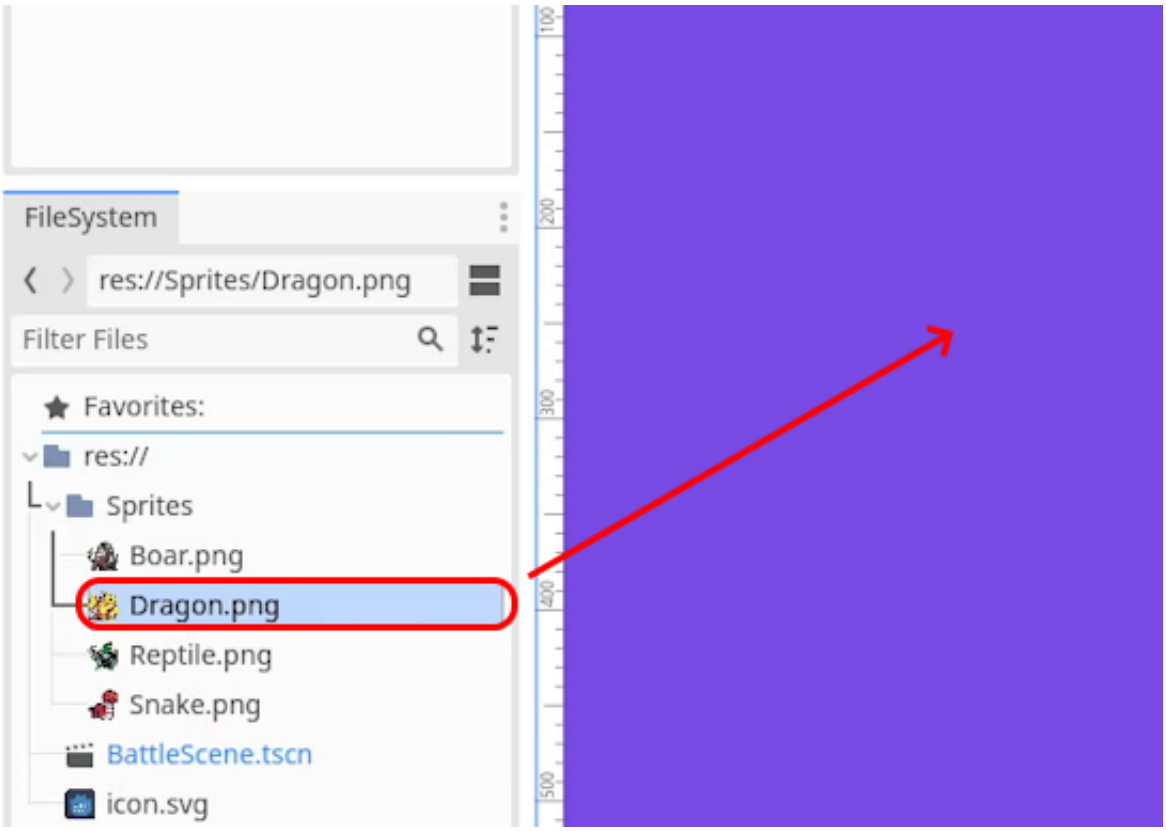
With our project all set up, in this lesson, we are ready to begin creating our character. Unlike with other games, our characters won't need to move and interact with the physics systems, so we won't be using a CharacterBody2D node, as you may have previously. Instead, we will use a new **Node2D** as the root node of our character.



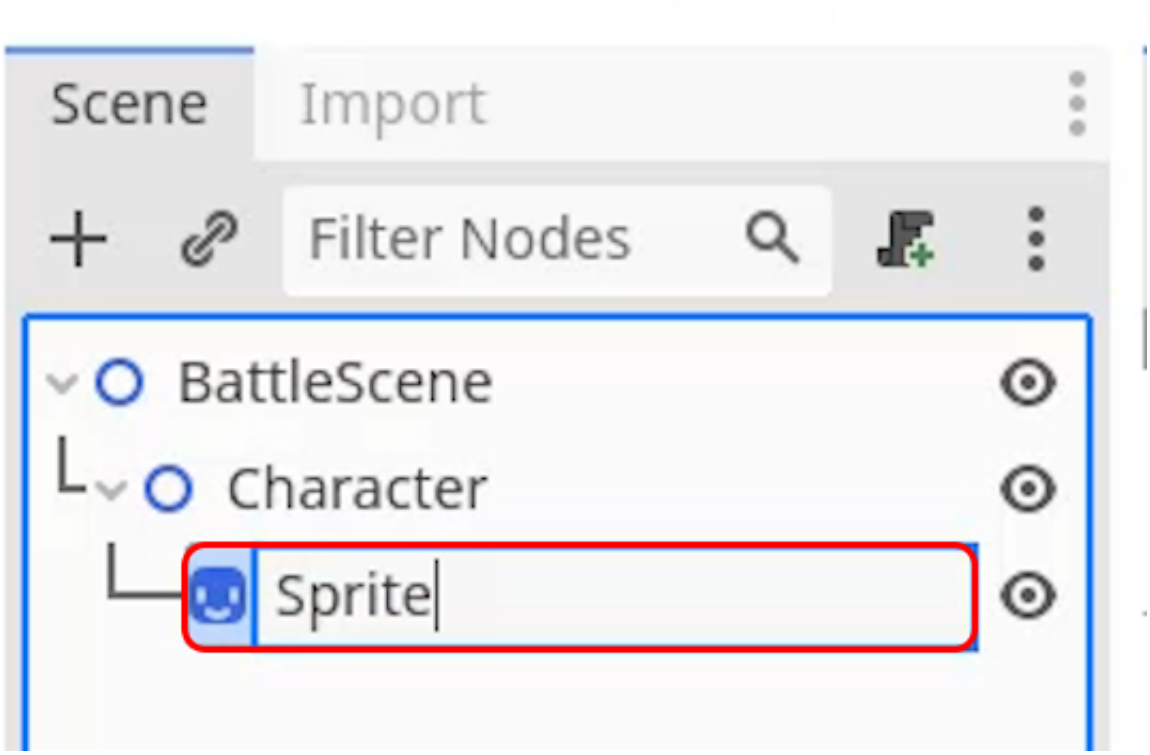
We can **rename** this to *"Character"*.



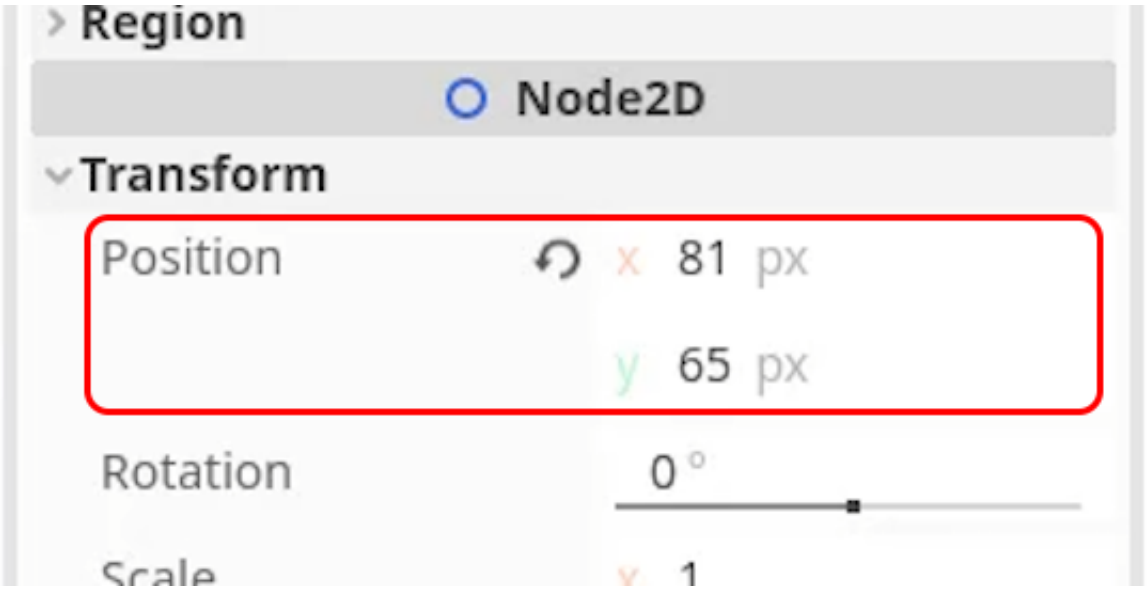
We can then add a visual element to our character by using one of the assets we added in the previous lesson. For this character, we will be using the *Dragon.png* asset.



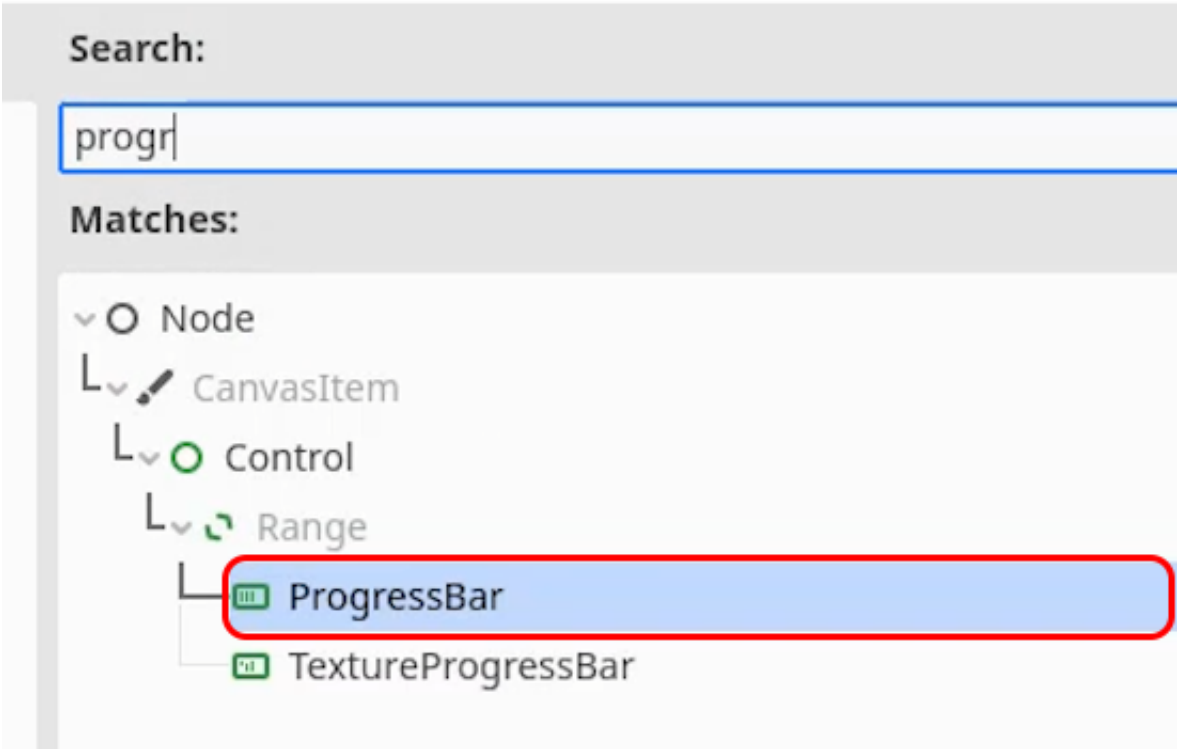
Make sure this node is a child of our *Character* root node, and **rename** it to *"Sprite"* so that we can identify it.



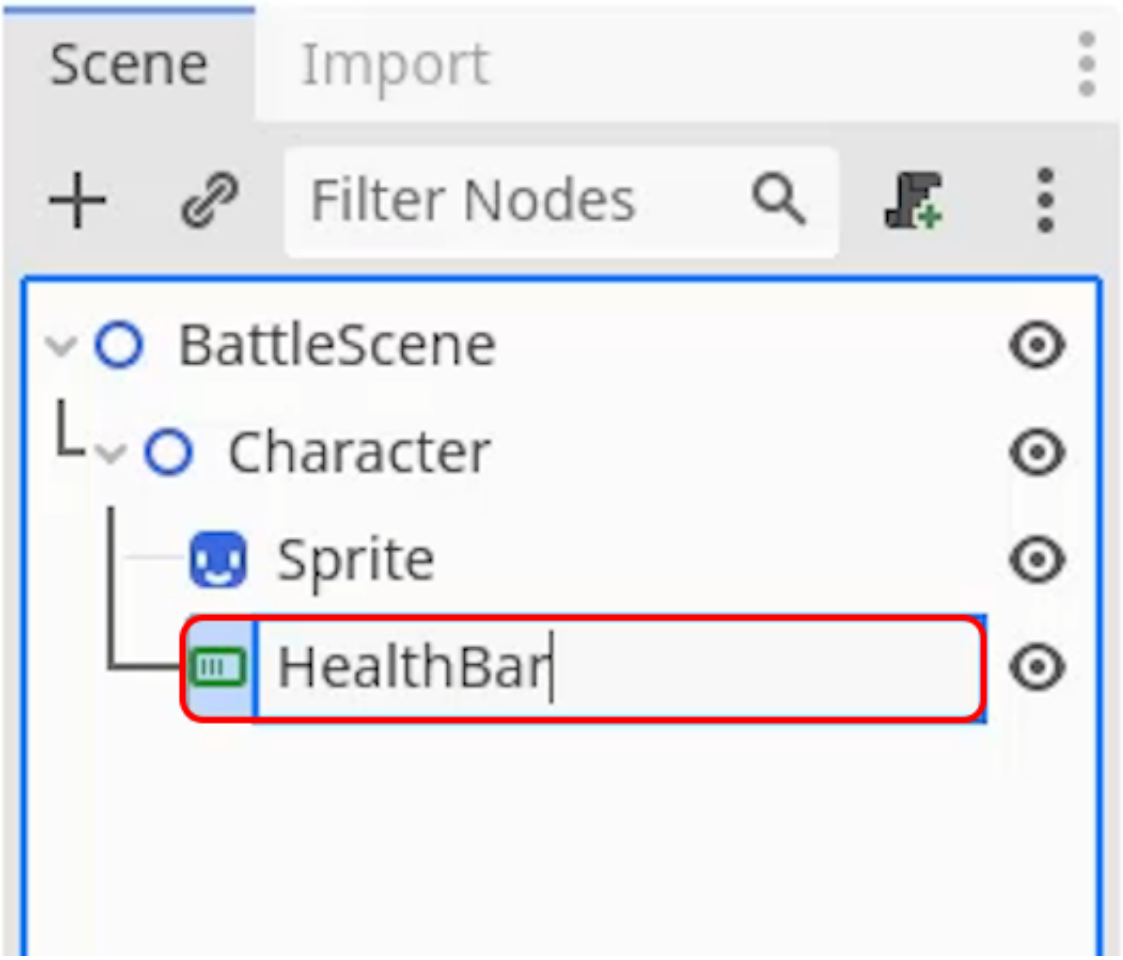
We also need to set the **Position** of the *Sprite* to **(0, 0)** to make sure it's centered on the *Sprite* node.



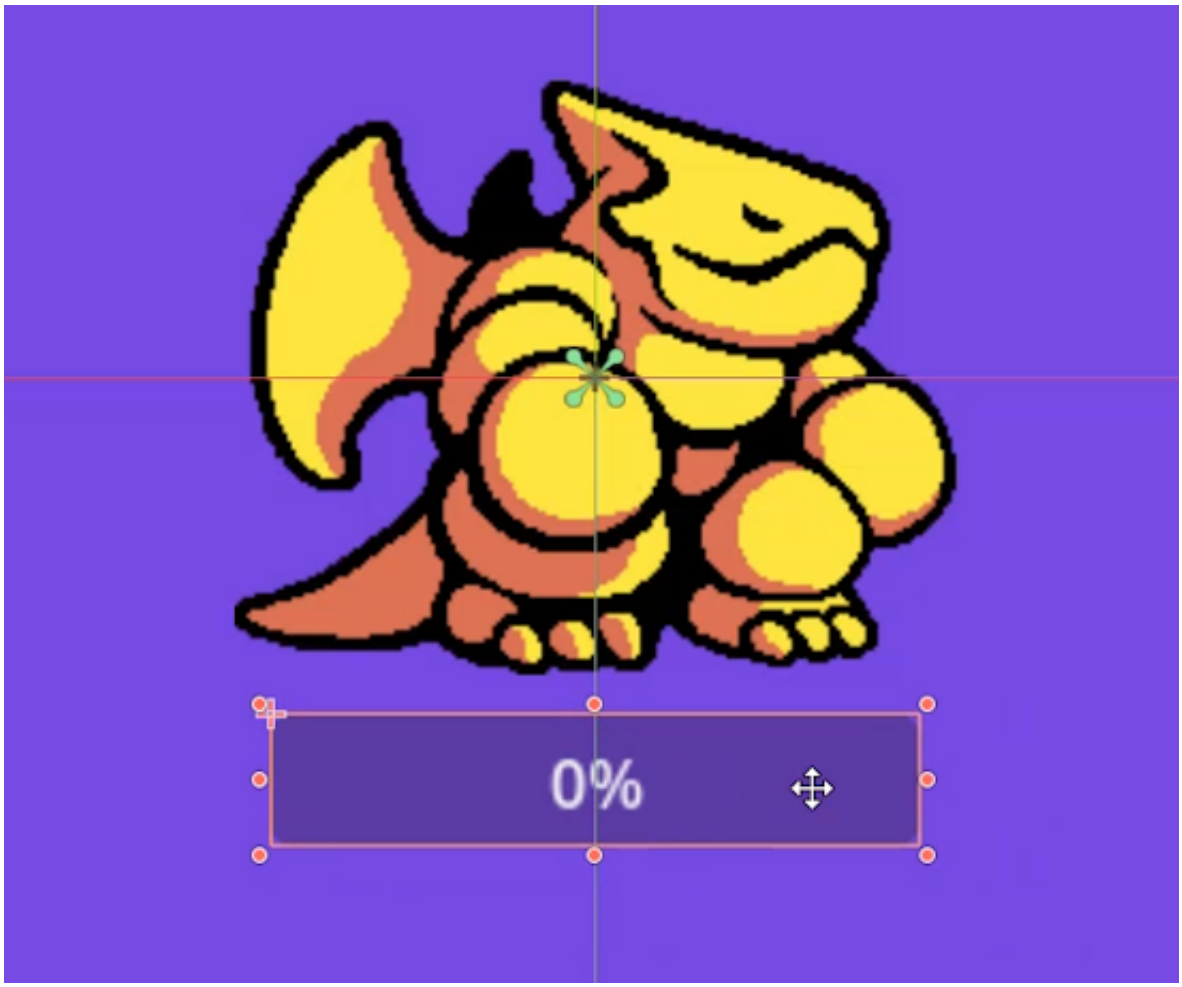
Alongside this sprite node, we also want our character to have a health bar that can adapt to the character's current health. To do this, we will add a **ProgressBar** as a child node of our *Character*.



This can then be **renamed** to *"Health Bar"*.

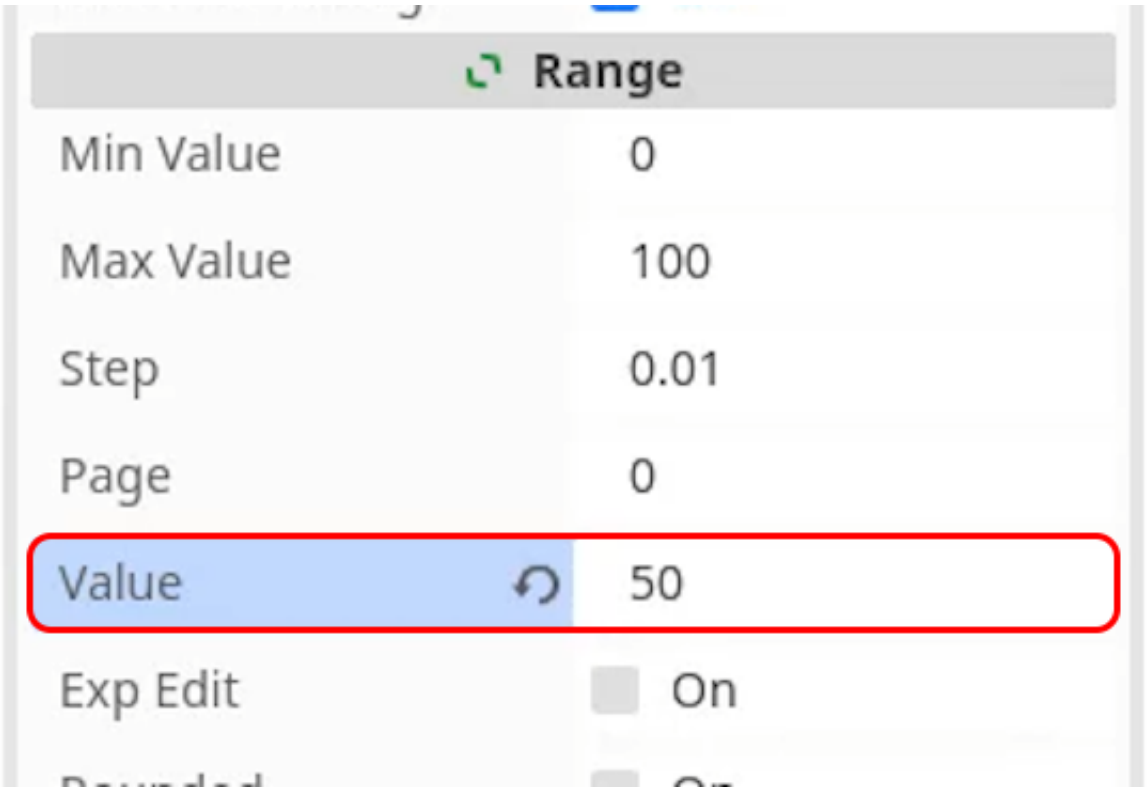


Use the orange circles to resize the element, as we will place it just beneath the *Sprite* element.

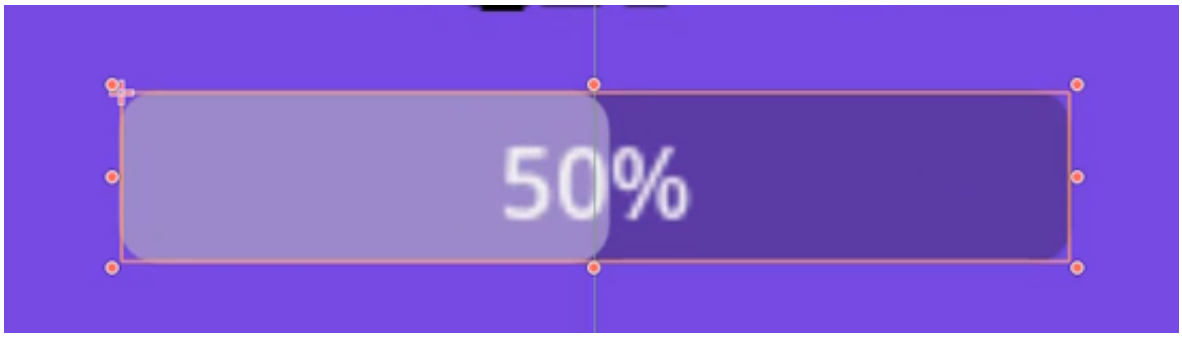


Changing the Health Bar

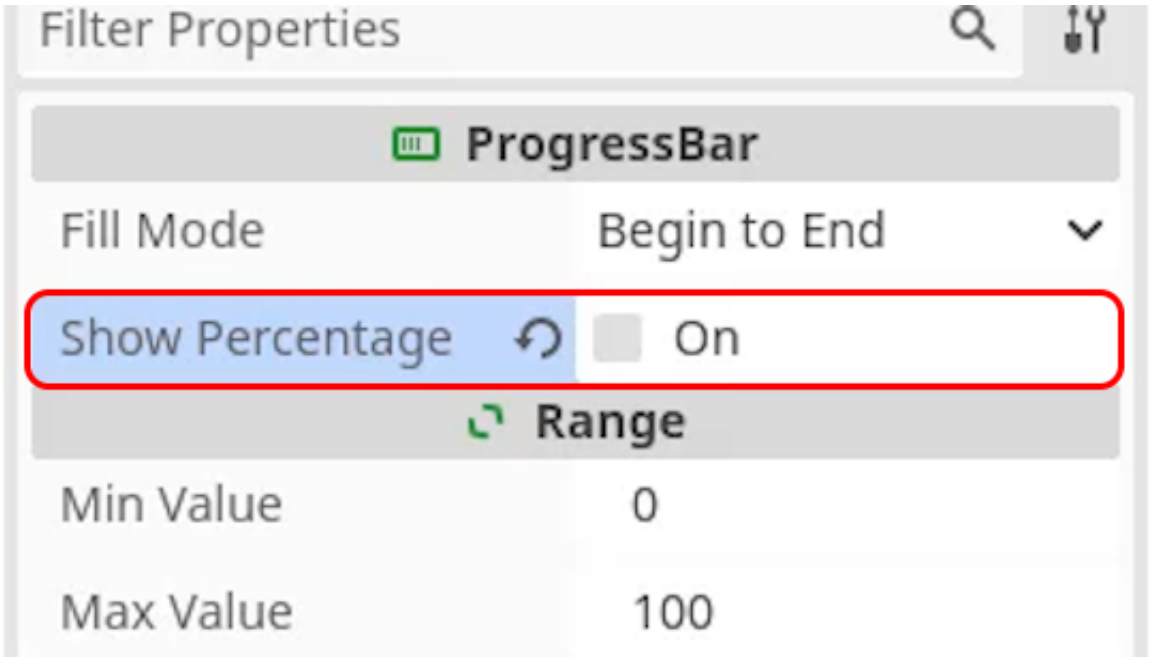
By default, a *ProgressBar* node doesn't look quite like a health bar, but there are lots of values we can customize to make it exactly how we like. To help us see the changes, we can change the **Value** property of the *Healthbar* node to **50**.



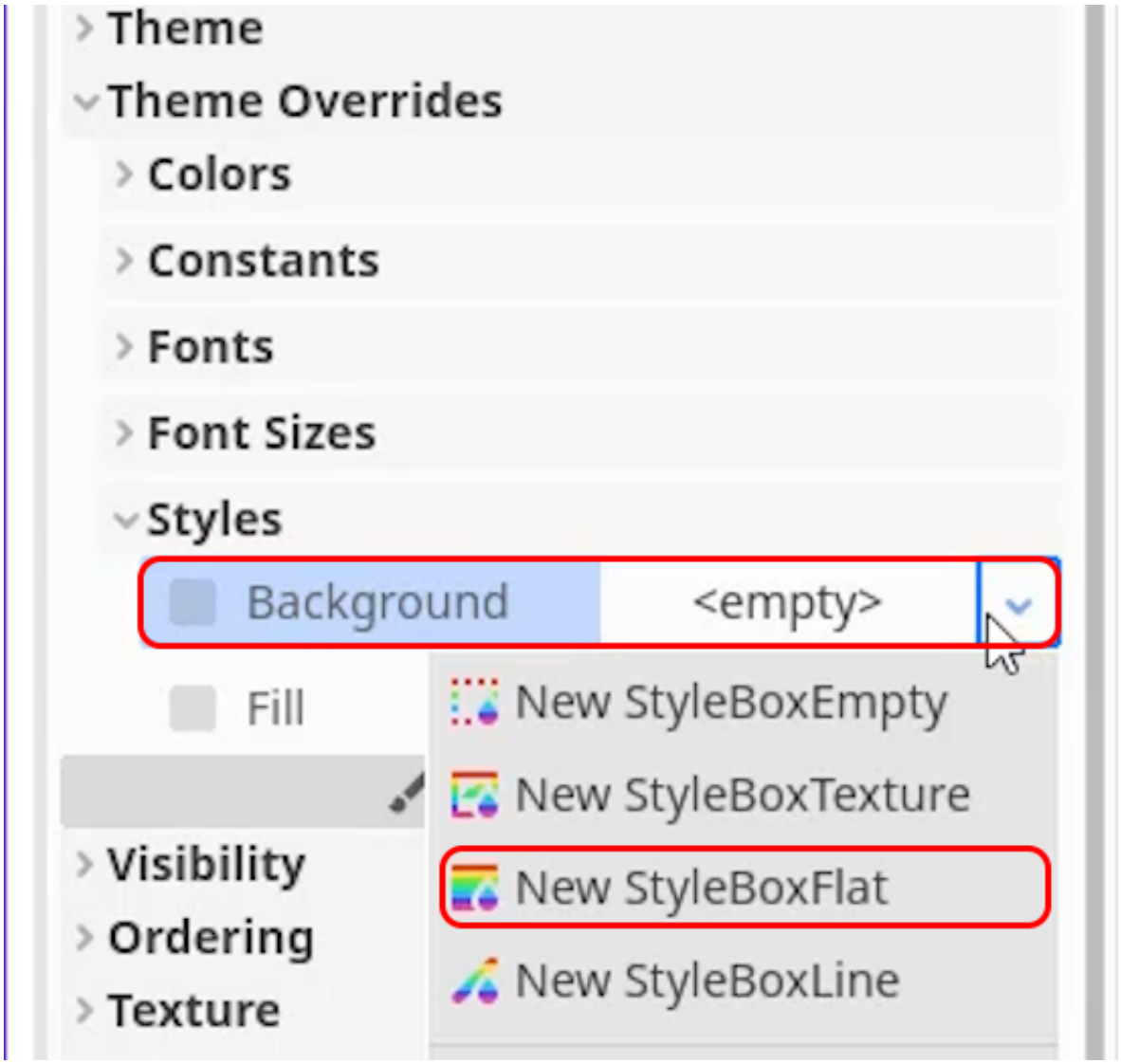
This will show the bar at half full in the scene view to see how any customizations affect it.



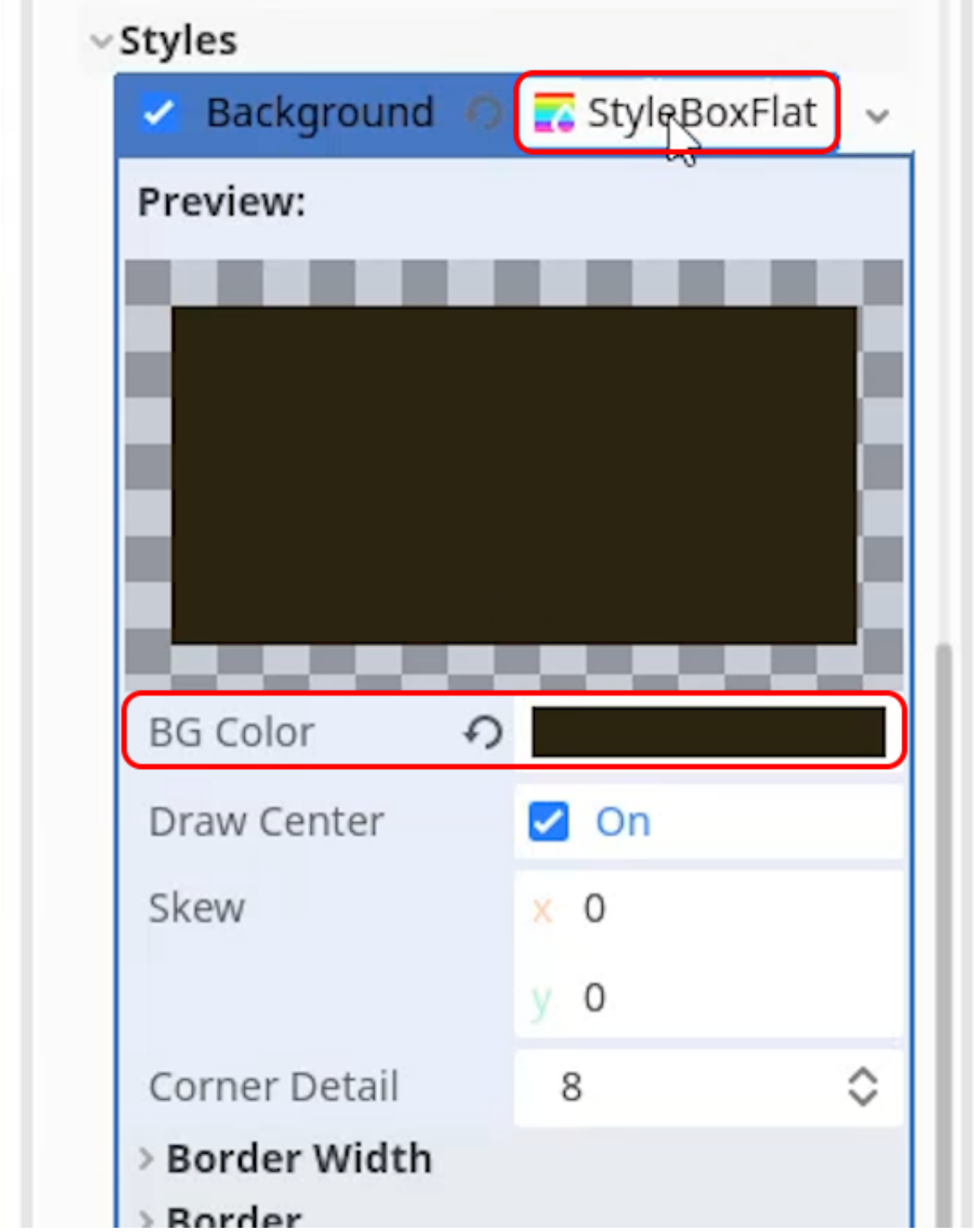
Our first change will be to disable the **Show Percentage** property, which will remove the text from the *HealthBar,* as we will be adding our own later.



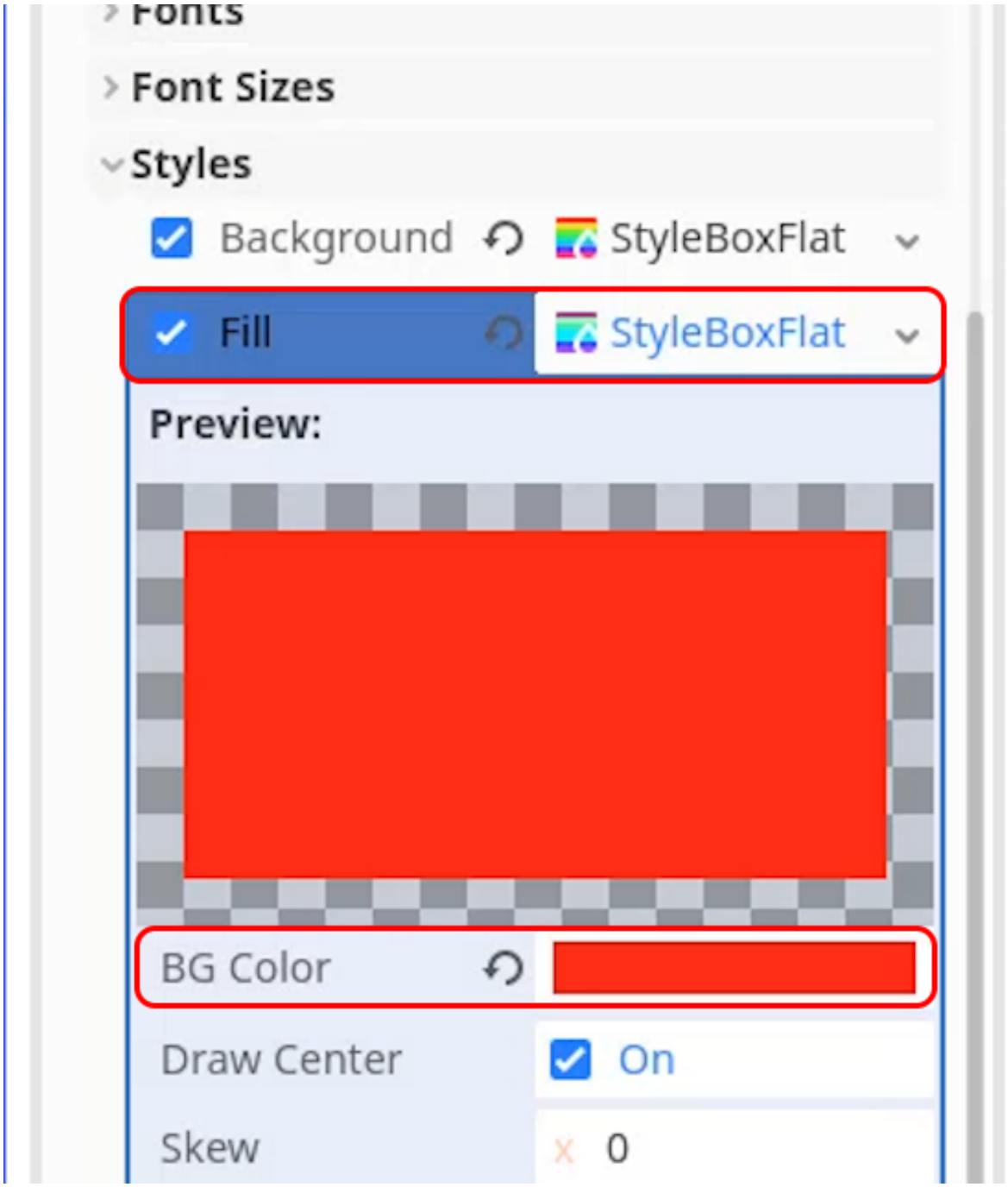
We can then begin changing the colors of our *HealthBar*. To do this, we will add new *Styles* in the *Theme Overrides* section. To change the **Background** color, select **New StyleBoxFlat** from the dropdown next to the property.



You can then press the new **StyleBoxFlat** value and change the **BG Color** property to a dark grey, to match the background color that we want.

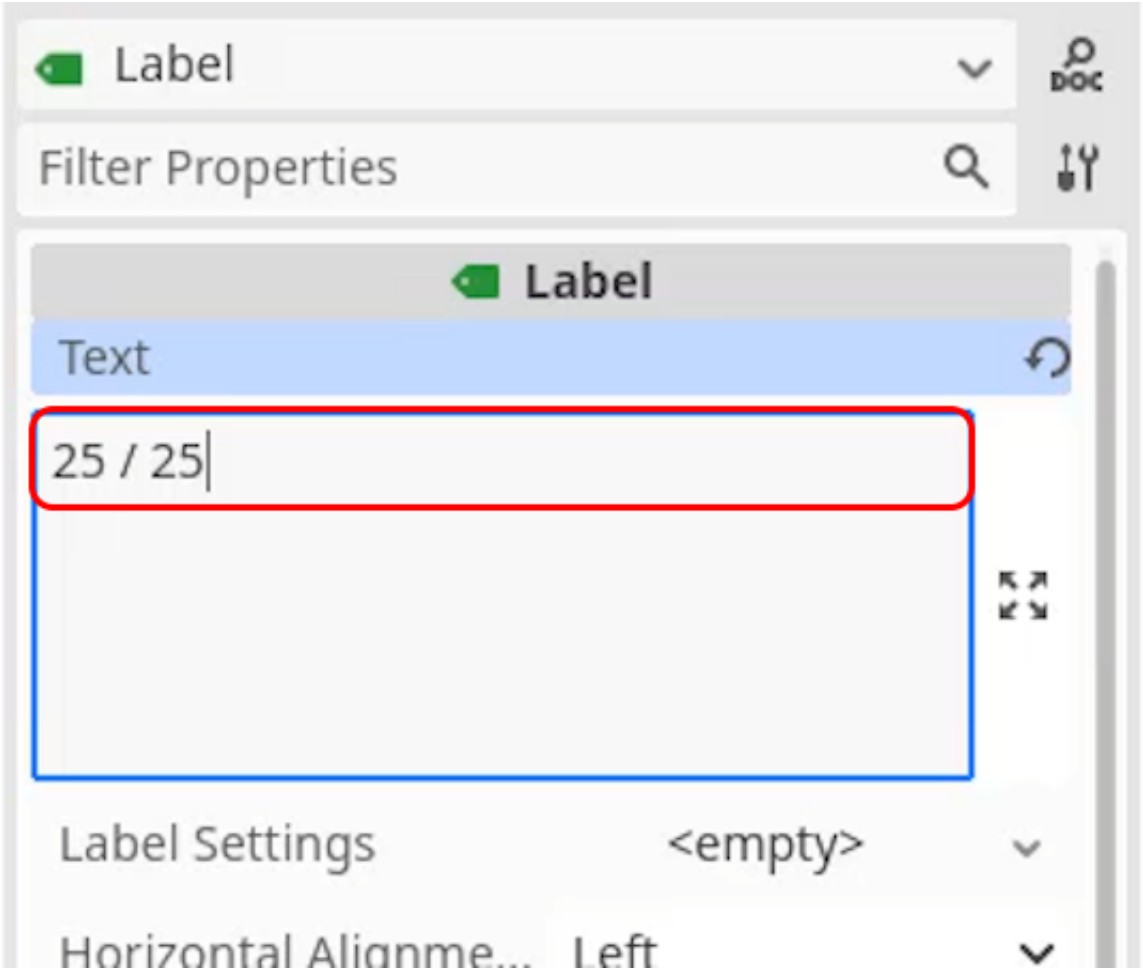


We can then do the same thing for the **Fill** property, except this time give the *BG Color* a red value.



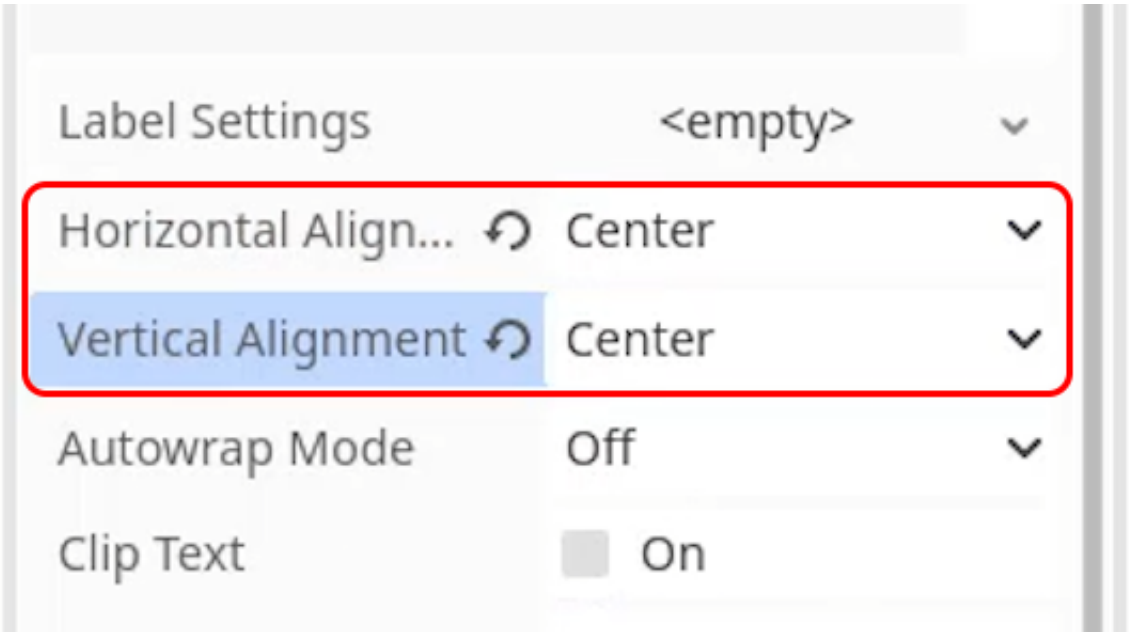
We also want to add text to our health bar, however, instead of a percentage, we want to represent our character's health as a fraction. We will do this using a new **Label** node, as a child node of our *HealthBar*.

So that we can get the looks right, we will give the *Label* a default **Text** value of *"25 / 25"*.

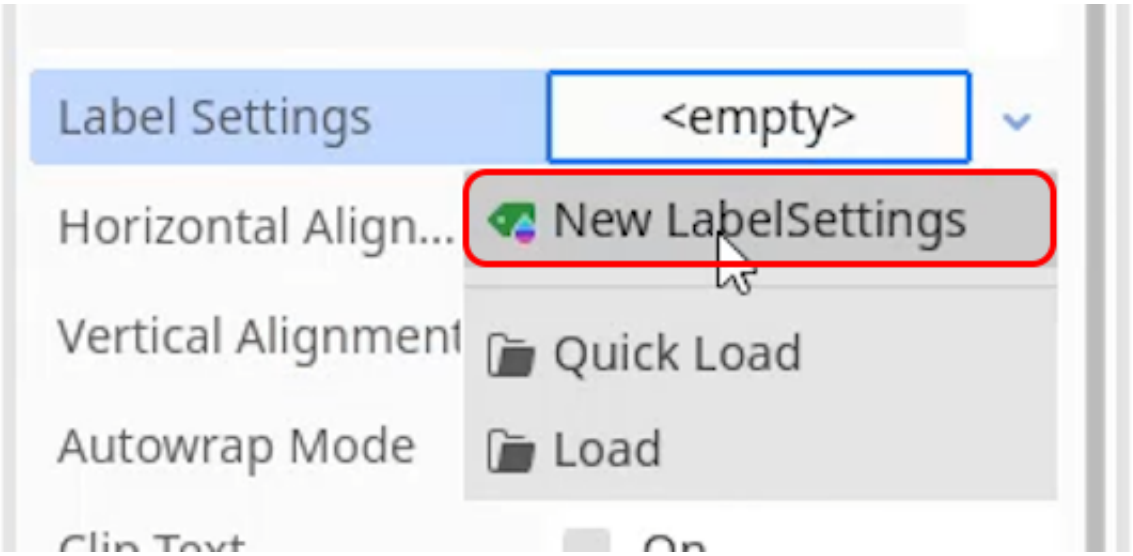


Next, we will resize the text box to fit the bounds of the *HealthBar* element.

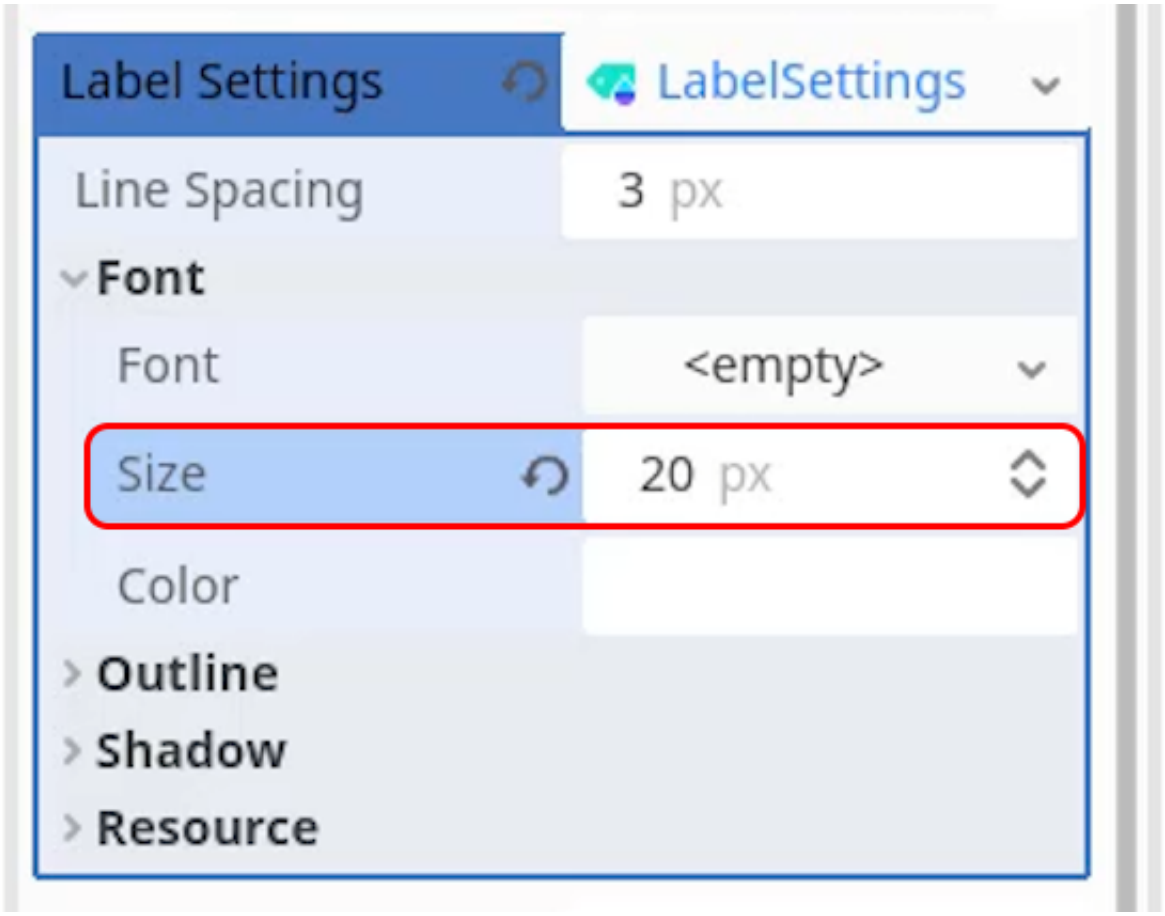
We can then change the **Horizontal Alignment** and **Vertical Alignment** to **Center** so that our text is centered within the *HealthBar* area.



We can also update the visuals of our *Label* element, by adding a **New LabelSettings** object to the node.

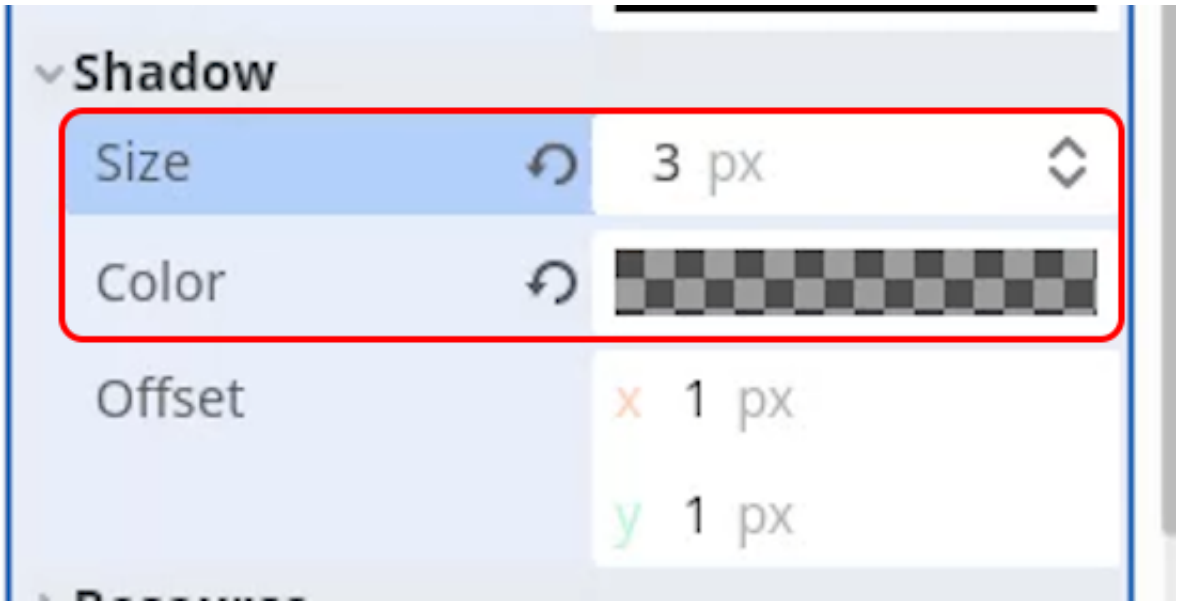


With this open, we can modify lots of values to change how the text is displayed. For this label, we will want to **increase** the **Font Size** value slightly. You may need to re-center the *Label* within the *HealthBar* after this.

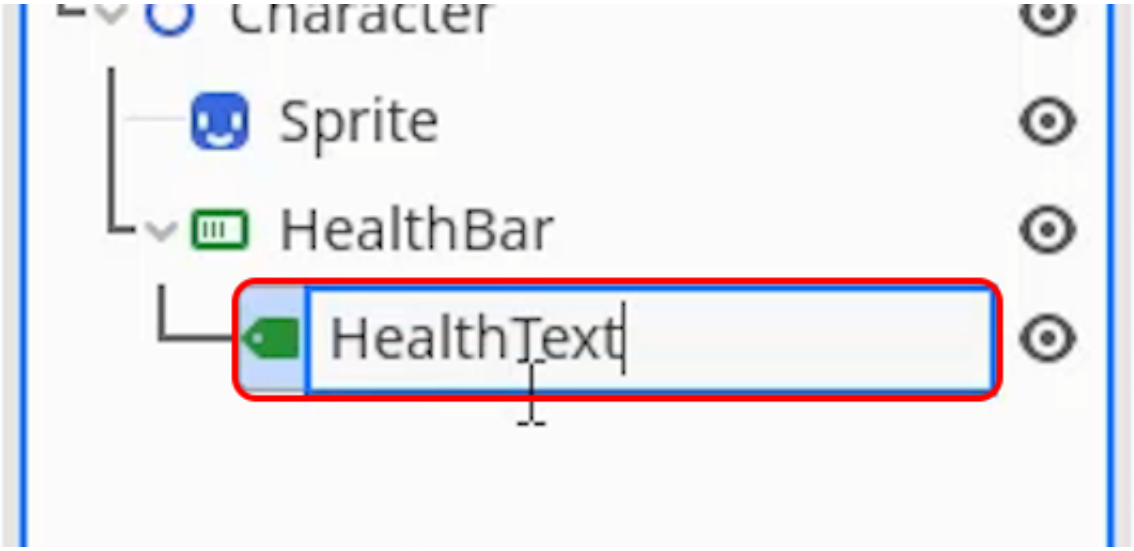


We can also add an **Outline** to the text to make it visible when the *HealthBar* is full. We will be using *3* pixels for the *Size* value and *black* for the *Color* value, but feel free to try experimenting with this yourself.

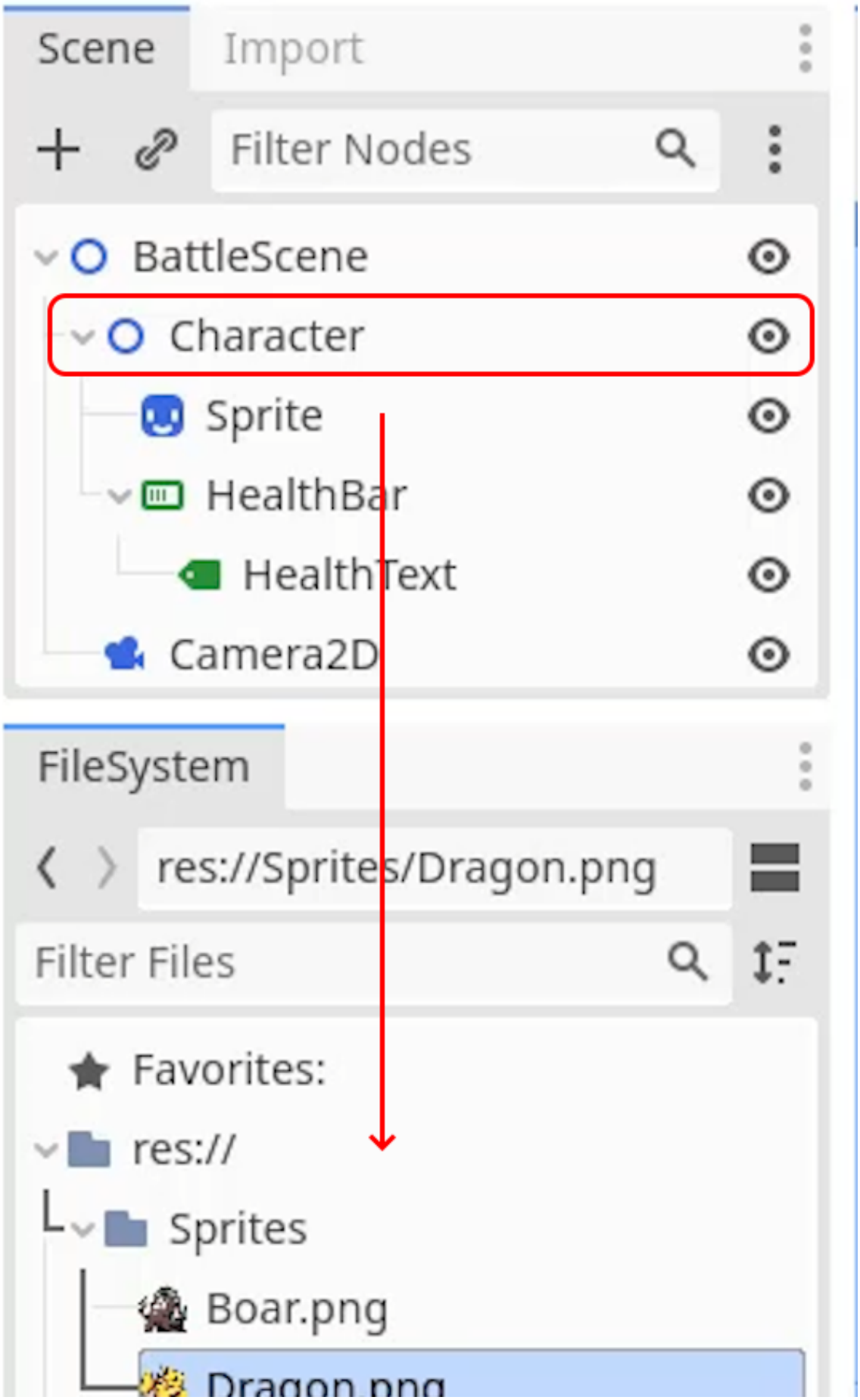
Finally, we can also change the **Shadow** property with some different values to make the text pop.



Finally, we will **rename** the *Label* to **“HealthText”** to make it more identifiable.

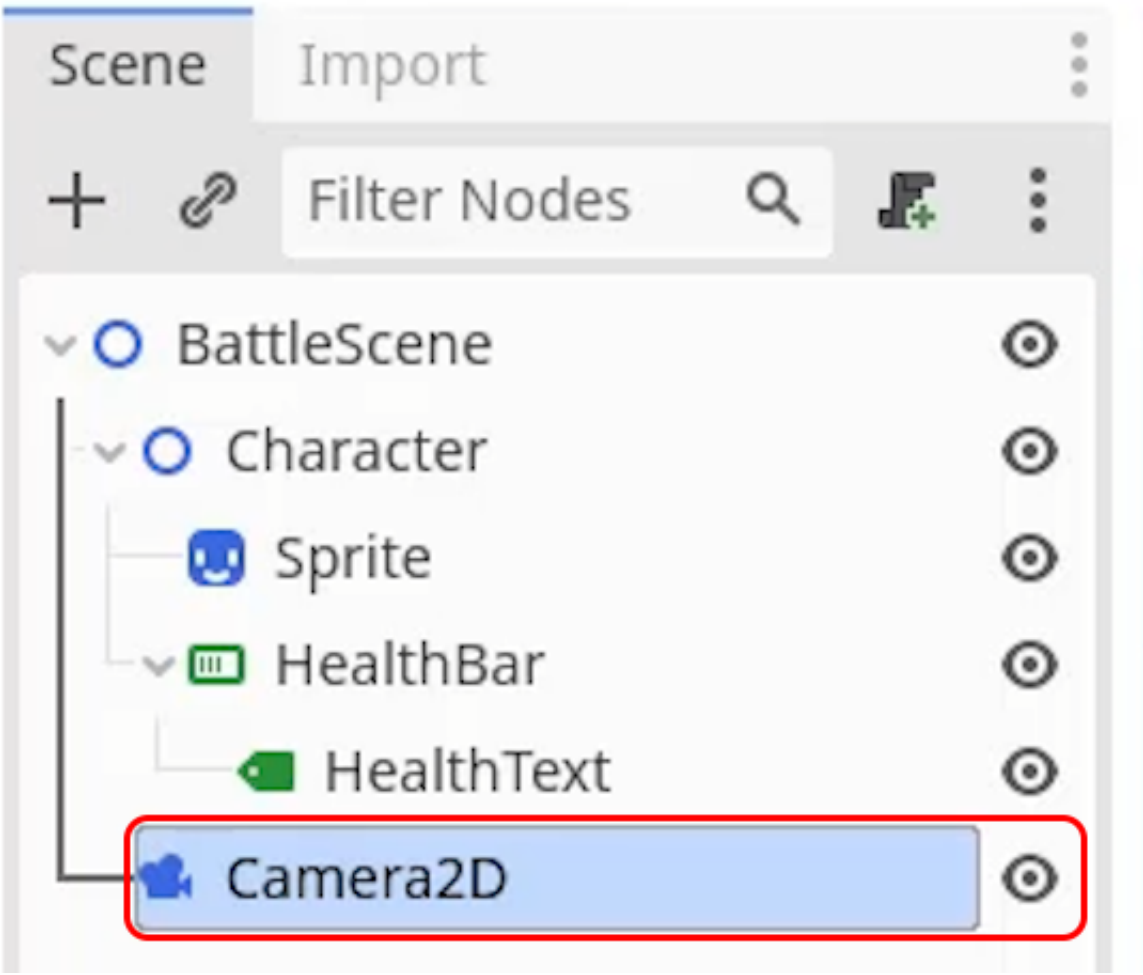


To complete our *Character*, we will drag the root node into the *FileSystem* to turn it into its own scene for use later in the project.

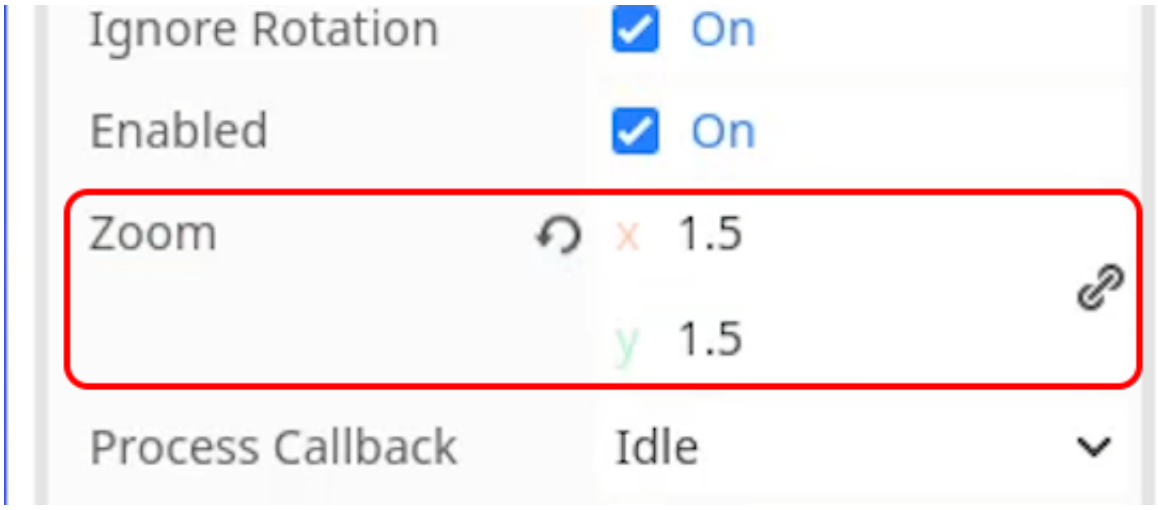


Adding a Camera

With these nodes in place, our character is ready to begin adding functionality to. However, if you press **play** at the moment, the character is stuck in the top-left corner of the screen. To fix this, we can add a **Camera2D** node to our scene tree.



By default, the *Camera2D* node is a little large for our use case, so we can change the **Zoom** value to **1.5** on both axes to zoom the camera in on our sprite.



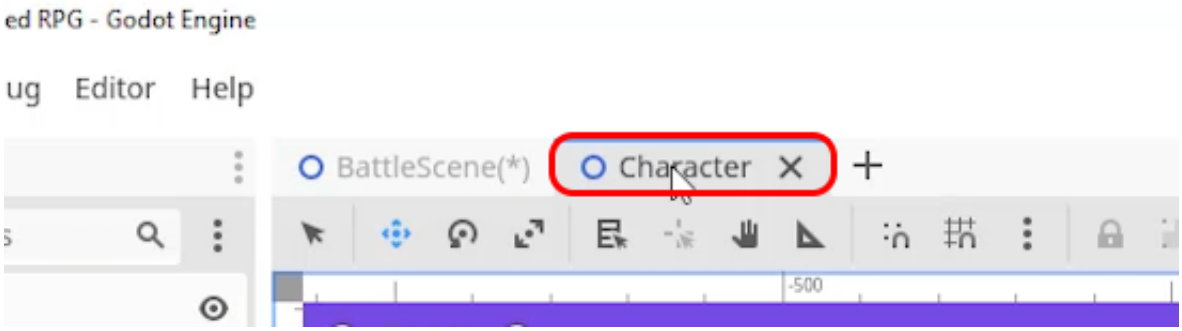
In the next lesson, we will continue working on our character and adding further functionality to the nodes we have set up.

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

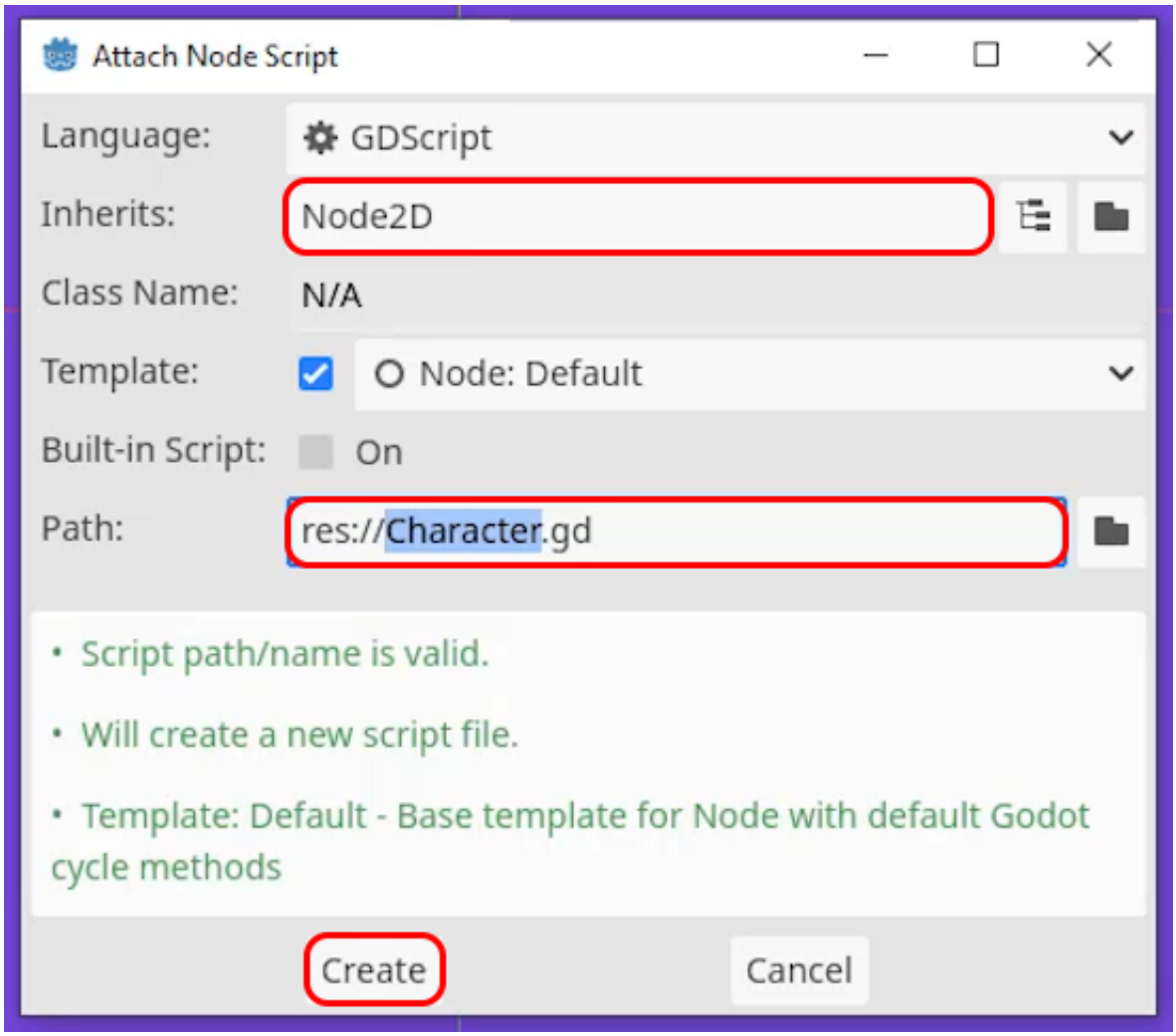
In this lesson, we will be setting up our character script. This will likely be the largest script in our game as it contains crucial functionality such as controlling the character's health and handling combat actions. We will be implementing combat actions later in this course, so for this lesson, we will focus on setting up the script and implementing the character's health systems.

Setting up the Character Script

To begin with, make sure you have the **Character scene** that we created in the previous lesson open, as we want to edit our character here to apply the changes across the project.



We can then add a new script to the node in the *Inspector* window. We will name this *Character.gd* and ensure that it inherits from *Node2D*. Finally hit **Create** to generate the script.



The first thing we need to do is define our *class* directly under the *extends Node2D* line.

```
class_name Character
```

This line defines our script as a class called *Character*, which will allow us to store this script as a variable in other scripts. Additionally, we will now be able to make use of our *Character* script as a base to extend other scripts from and add extra functionality to.

Creating the Variables

As with most scripts, we will need a few variables to use with our *Character*. The first variable will track whether the current character instance is the player, we will call this **is_player**, make it of type **bool,** and definite this with an **export** tag to make it visible for editing in the Godot *Inspector* window.

```
@export var is_player : bool
```

The next variable will track the character's current health, we will call this **cur_hp**, and give it a type of **int** with a default value of **25**. This will also be **exported** as it will change based on the character.


```
@export var cur_hp : int = 25
```

We can copy and paste this line for the maximum health variable, although we will change the name to **max_hp**.

```
@export var max_hp : int = 25
```

We will also need a variable that can keep track of our *combat actions*. We won't be implementing this yet, but we will create the variable for use later in the course. This variable will be called **combat_action**, be of type **Array**, and also have an **export** tag to make it visible in the *Inspector*.

```
@export var combat_action : Array
```

Additionally, we will make use of a variable to track our **opponent**, which will be of type **node** and also be **exported**.

The following code has been updated, and differs from the video. If you are following along with both the video lesson and lesson summary, there is a typo at 3:03 on the variable name for the opponent. This should not make a difference to your code, just remember to change the spelling across any future use of this variable.

```
@export var opponent : Node
```

Finally, we will need to access the *HealthBar* and *HealthText* nodes inside our script. We won't be exporting these, instead, we will use the **onready** tag along with the **get_node** function to assign the variables when the script starts. These variables will be called **health_bar** and **health_text** and be of type **ProgressBar** and **Label** respectively.

```
@onready var health_bar : ProgressBar = get_node("HealthBar")
@onready var health_text : Label = get_node("HealthBar/HealthText")
```

Implementing the Functions

With our variables in place, we are ready to begin implementing some of the functions that we will be making use of across the project. One function we will use quite a lot will update the *HealthBar* and *HealthText* values. We will call this function **_update_health_bar**.

```
func _update_health_bar():
    health_bar.value = cur_hp
```

This code will update the current *Value* for our *HealthBar* however we also need to set the *Max Value* of our *HealthBar* node so that it has something to work from. To do this, we will add some code

to our *_ready* function.

```
func _ready():
    health_bar.max_value = max_hp
```

This will update the *HealthBar* to the correct maximum based on our *max_hp* variable. With this set up, we finally need to update our *HealthText* node to contain the right information. We can add this code to our *_update_health_bar* function as well.

```
func _update_health_bar():
    ...
    health_text.text = str(cur_hp, " / ", max_hp)
```

Here we use the **str** function to combine our *cur_hp* and *max_hp* values into a string that shows them as a fraction. We can also add a new function called **take_damage** with a parameter of **damage**.

```
func take_damage (damage):
    cur_hp -= damage
    _update_health_bar()
```

Here we decrease the current health value, *cur_hp*, by the damage passed through in the parameter. We also need to call the *update_health_bar* function to make this visible to the player. In this function, we can also check to see if the *cur_hp* variable is less than or equal to zero, and if so, handle the character dying.

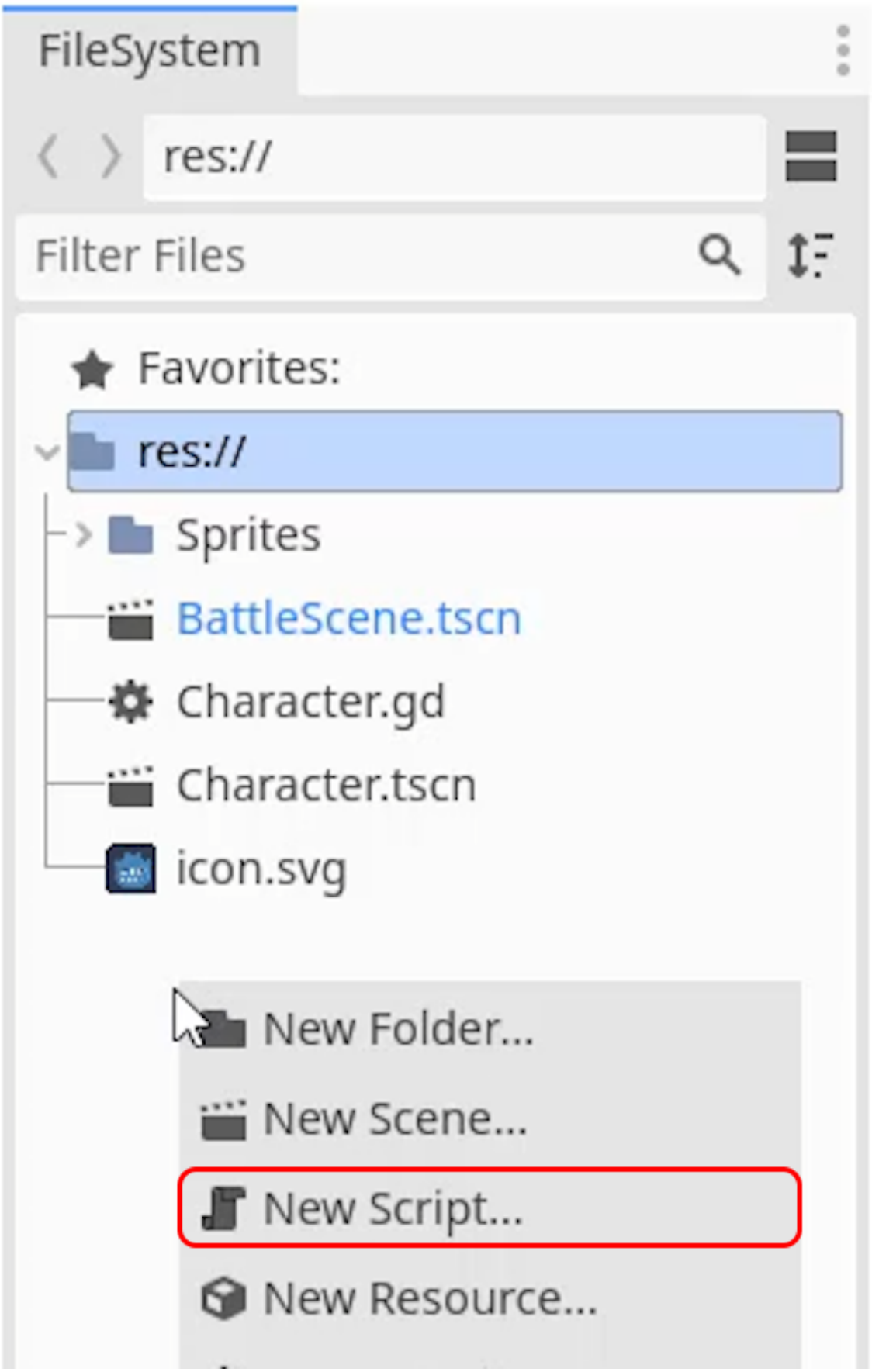
```
func take_damage (damage):
    ...

    if cur_hp <= 0:
        queue_free()
```

To “kill” the character, we use the *queue_free* function, which will destroy the *Character* node in the scene. Later in the course, we will implement a bit more functionality here, but for now, this will work perfectly. Finally, we can also create the *heal* function, which will be the opposite of the *take_damage* function.

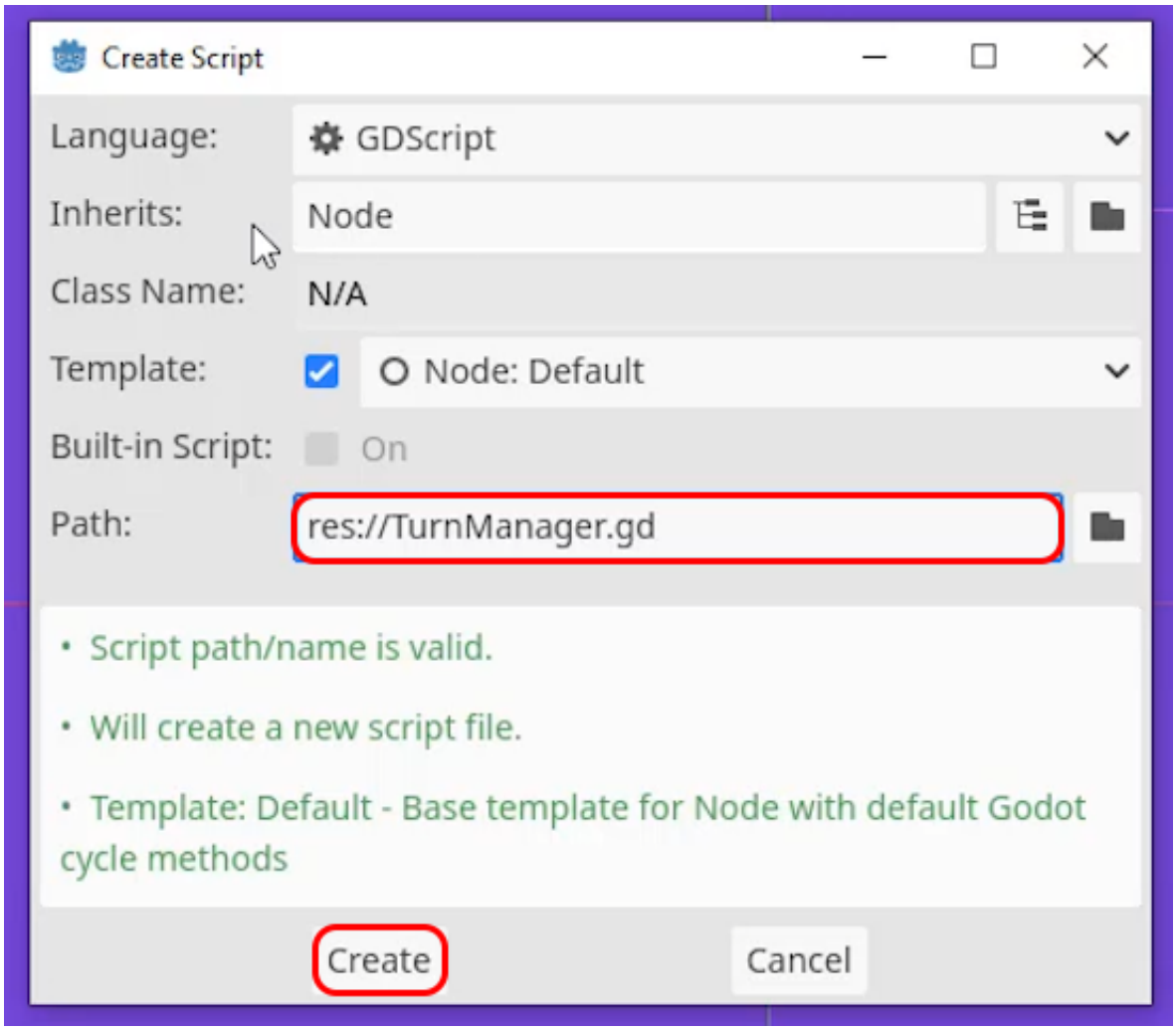
```
func heal (amount):
    cur_hp += amount

    if cur_hp > max_hp:
        cur_hp = max_hp

    _update_health_bar()
```


You will notice that, unlike our *take_damage* function, we also add an additional if statement here. This limits the *cur_hp* value to the maximum hp set by the *max_hp* variable. With this in place, we are ready to begin setting up our *Turn Manager* in the next lesson, which will be used to control the game.

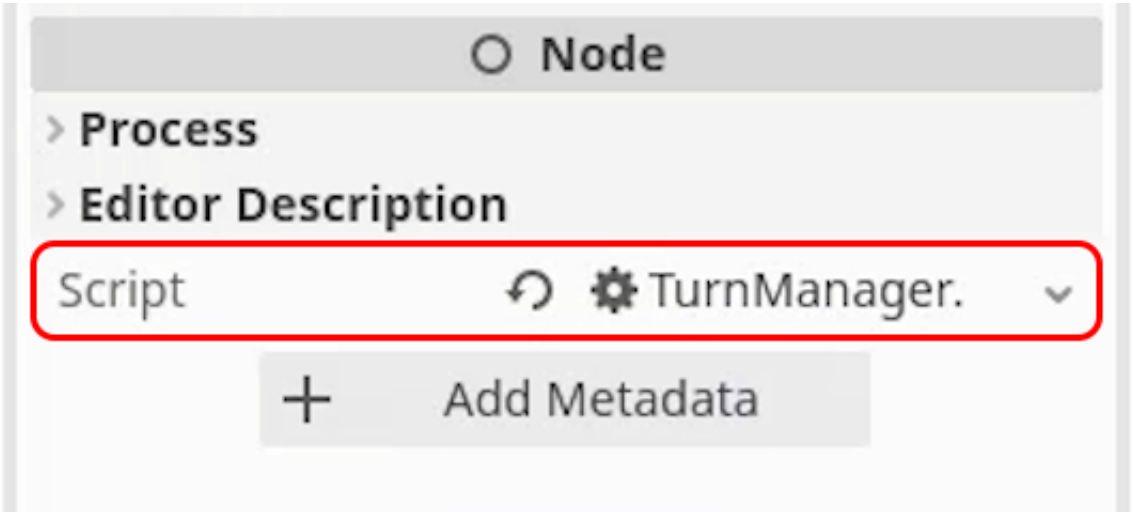
In this lesson, we will begin to create the *Turn Manager* for our game. This is an important component that will focus on deciding which character's turn it is, allowing them to select their combat actions. To begin with, we will create a **new script** by right-clicking in the *FileSystem* window.



We can then call this script *TurnManager.gd* and press the **Create** button.



We made this script inside the *FileSystem* directly, as we won't need to directly modify any specific nodes. This means we can apply the script directly to our **BattleScene** root node.



Creating the Variables

The first variable we need will keep track of the player's *Character*. We can call this **player_char**, with a type of **Node**. To give this a value, we will use the **export** tag so that it visible in the *Inspector* window for us to drag the player node into.

```
@export var player_char : Node
```

We will need the same variable to keep track of our enemy's *Character*, we can call this one **enemy_char**.

```
@export var enemy_char : Node
```

We will also need to keep track of who's turn it currently is, so we will create another variable called **cur_char** (for "current character") with a type of **Character**. This *Character* type is directly referencing our *Character.gd* script, which is made possible as we defined it as a class using the *class_name* line previously. By defining the class it makes it much easier for us to access the *Character's* variables and functions, which will be necessary in our turn manager script.

```
var cur_char : Character
```

Unfortunately, Godot doesn't handle exporting custom classes very well, so we can't change our *player_char* and *enemy_char* types to *Character*. Our next variable will be a **float** called **next_turn_delay**. This will be used to define the delay between ending a turn and starting the next character's turn. We will give this a default value of **1.0** but add an **export** tag as well to make it editable in the *Inspector.*

```
@export var next_turn_delay : float = 1.0
```

We will also need a **game_over** variable to keep track of the current game state. This will be a **bool** and have a default value of **false**. This will be turned to true when one of the characters has defeated the other.

```
var game_over : bool = false
```

Creating the Functions

This script won't need the *_process* function so we can remove that, however we will be making use of the *_ready* function. With the variables in place we can begin defining our functions, although we won't be writing any functionality in them just yet, so instead we can fill them with the **pass** keyword. Our first function will be called **begin_next_turn** which will handle switching characters and informing the next character that their turn has begun.

```
func begin_next_turn():
    pass
```

We will also make use of a function called **end_current_turn**, which will be called when a combat action has been completed.

```
func end_current_turn():
    pass
```

Finally, we can add a function called **character_died** which will need a parameter of **character** and this will be called when either the player's or the enemy's health is zero or less.

```
func character_died(character):
    pass
```

This defines everything our turn manager script will be capable of, so we are ready to begin adding code to our functions. The first step in our **begin_next_turn** function is to check if the *cur_char* variable is our player's *Character* node.

```
func begin_next_turn():
    if cur_char == player_char:
        cur_char = enemy_char
```

If the current character is the player, we can swap the value to the enemy character. Equally we can swap the *cur_char* variable to our *player_char* if it is the other way around.

```
func begin_next_turn():
    ...
    elif cur_char == enemy_char:
        cur_char = player_char
```

Finally, in case our *cur_char* variable hasn't been set, we can set it to the *player_char* as the default. This way our player will go first as intended.

```
func begin_next_turn():
    ...
    else:
        cur_char = player_char
```

We can then call the *begin_next_turn* function in the **end_current_turn** function to switch the characters when the current character calls to end their turn.

```
func end_current_turn():
    begin_next_turn()
```

However, this doesn't cover the character having died. To fix this, we can add a check for

the *game_over* variable before calling the *begin_next_turn* function.

```
func end_current_turn():
    if game_over == false:
        begin_next_turn()
```

We also want to use the *next_turn_delay* variable to delay the next turn beginning. This can be done using a *Timer* to pause this function and wait a certain amount of time before continuing to the following lines. We will do this in our **end_current_turn** function before the *begin_next_turn* function is called.

```
func end_current_turn():
   await get_tree().create_time(next_turn_delay).timeout
   ...
```

This code uses the **await** keyword, which effectively pauses the code execution until the event given is completed. In this case the event given is the **timeout** event of our new timer, which will occur when the time given by our *next_turn_delay* variable has passed. Finally, we can update the *game_over* variable in the **character_died** function.

```
func character_died(character):
    game_over = true
```

Here, it doesn't matter if our character is the enemy or the player as the game will be over in either case. However, we do need to check if the dead player is the player character, and if so print a *"Game Over!"* message to the player, as they have lost the fight.

```
func character_died(character):
    ...
    if character.is_player == true:
        print("Game Over!")
```

If the character isn't the player, this means they have defeated the enemy, and therefore we can print a message saying *"You Win!"*.

```
func character_died(character):
    ...
    else:
        print("You Win!")
```

Finally, we will call the **begin_next_turn** function from the **_ready** function to begin the game.

```
func _ready():
    begin_next_turn()
```


We can add a delay of 0.5 seconds to this by copying the code from the *end_current_turn* function and place this before our *begin_next_turn* call.

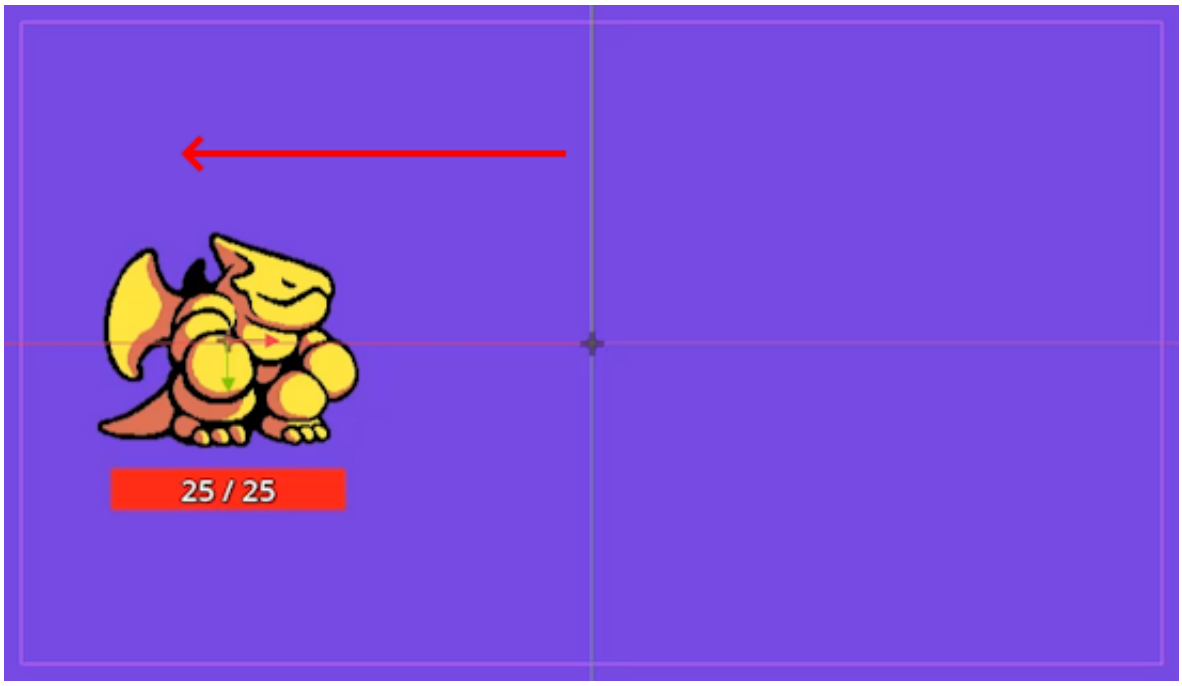
```
func _ready():
    await get_tree().create_timer(0.5).timeout
    ...
```

This concludes the functionality for our turn manager script, however, it doesn't actually communicate with our player and enemy characters. We will fix this in the next lesson, by setting up the systems to communicate to the characters if it is their turn or not.

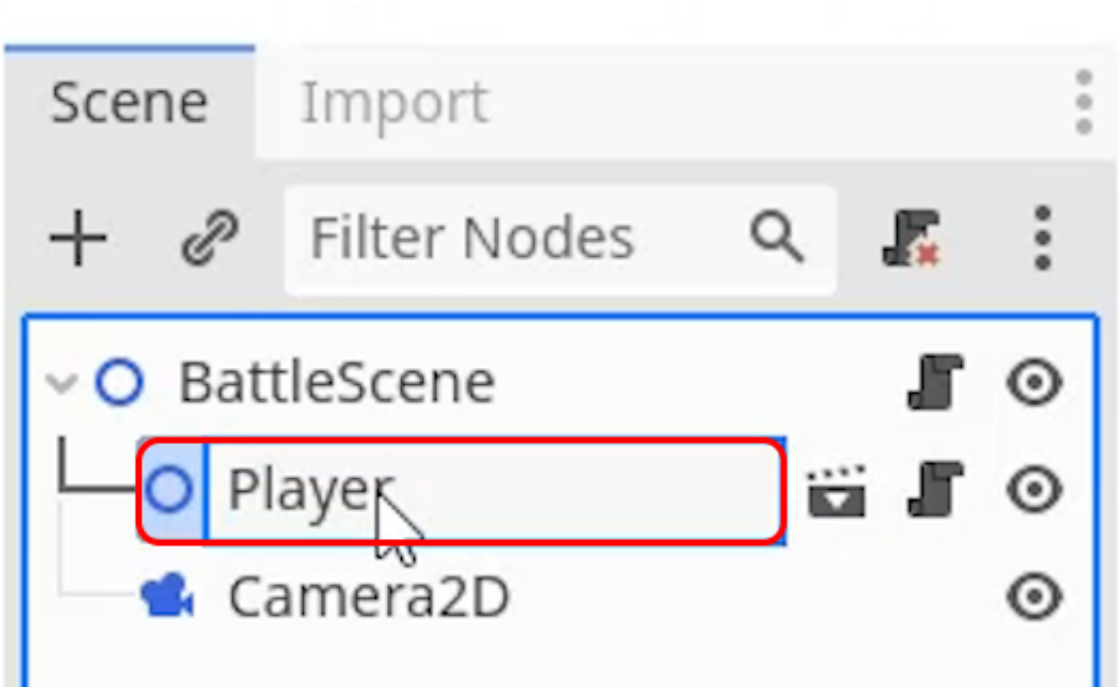
In this lesson, we will continue setting up our *TurnManager* script and implement the systems to communicate with the *Character* instances. This will allow us to let both the player and the enemy know when their turn has begun and when it has ended.

Adding Two Characters

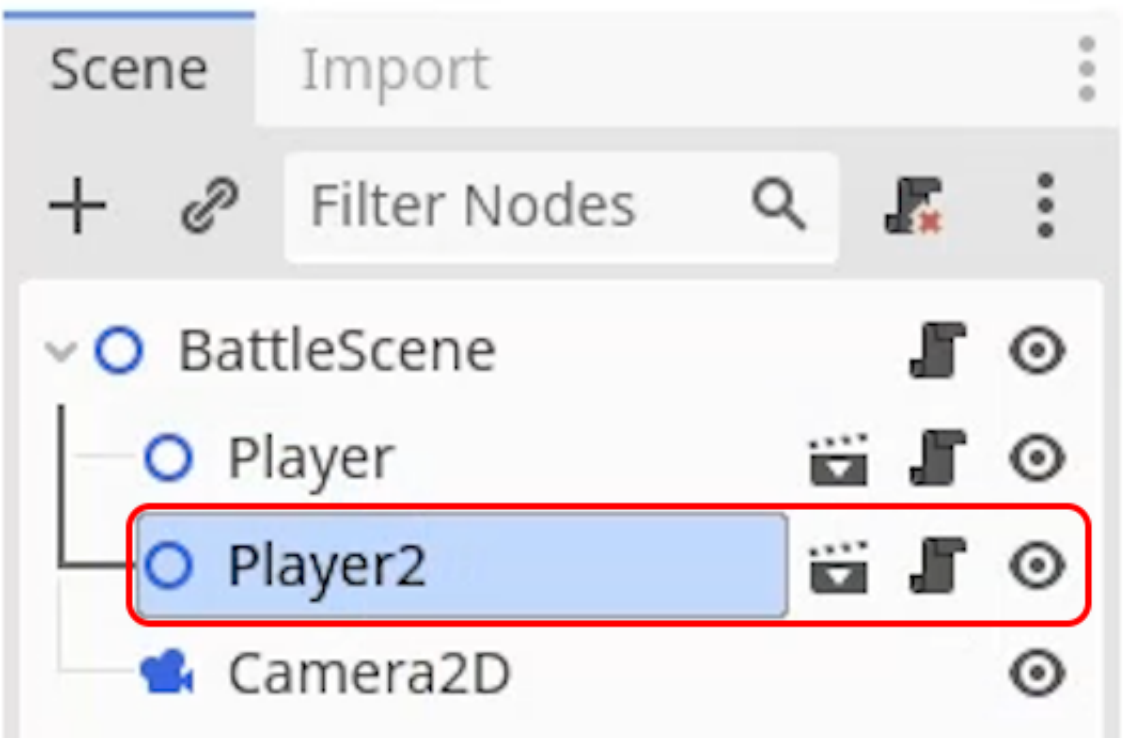
Before we can begin adding extra code to the *TurnManager* script, we first need to add both our player and our enemy characters to the scene. To begin with, we will move the existing *Character* object to the left side of the screen. Remember you can see the size of the screen defined by the purple rectangle shown in the *2D* scene view.



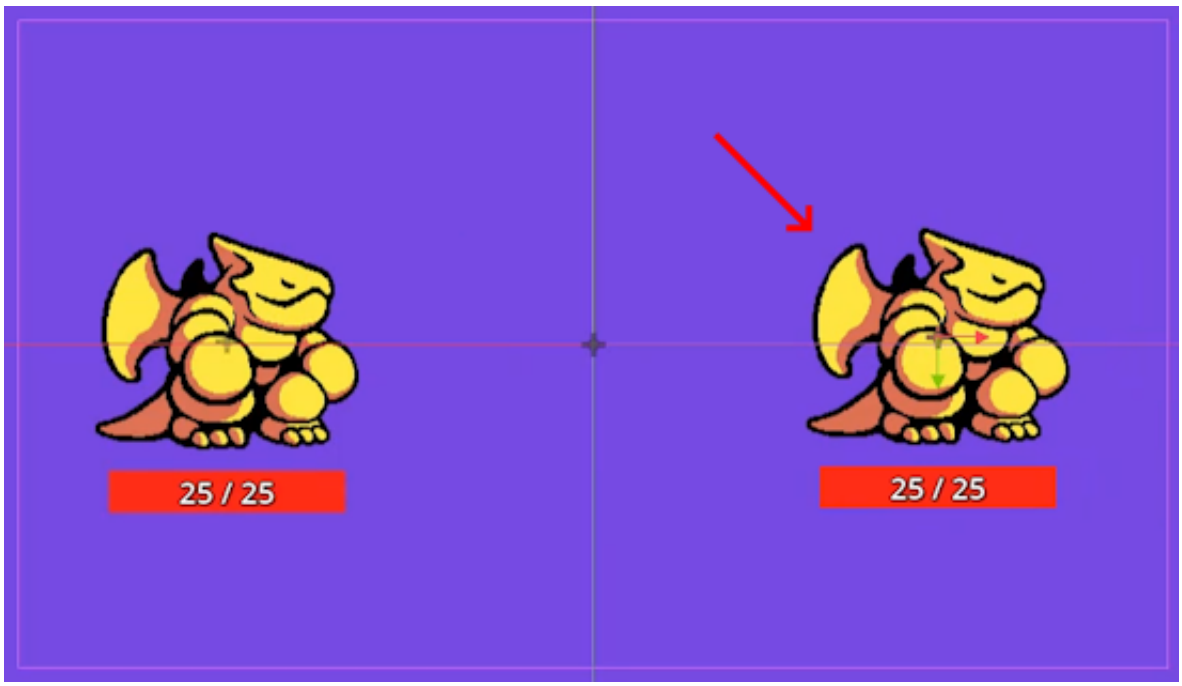
We can then **rename** this *Character* instance to **“Player”**.



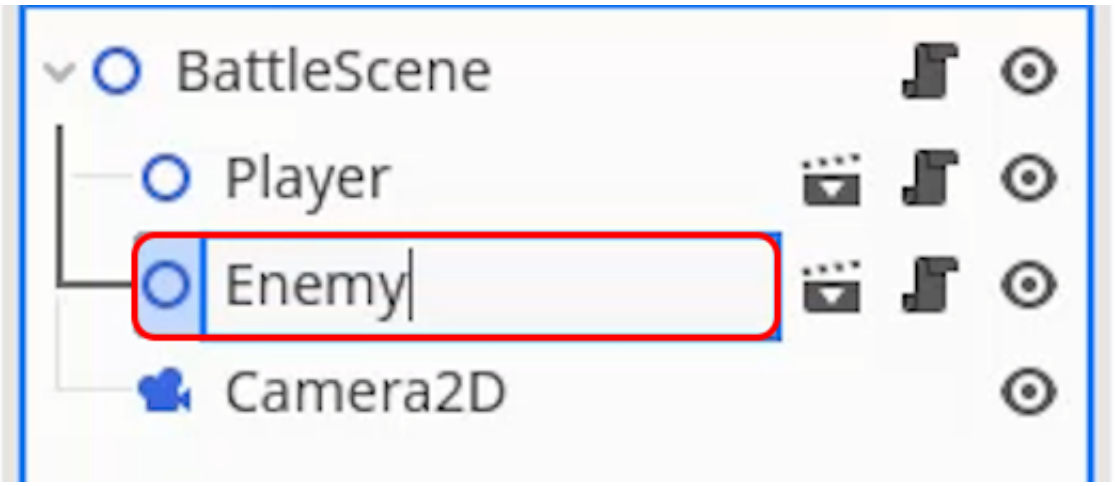
Because we have previously turned our *Character* into its own scene, to create our enemy character instance we can simply **duplicate** the *Player* instance (**CTRL+D**).



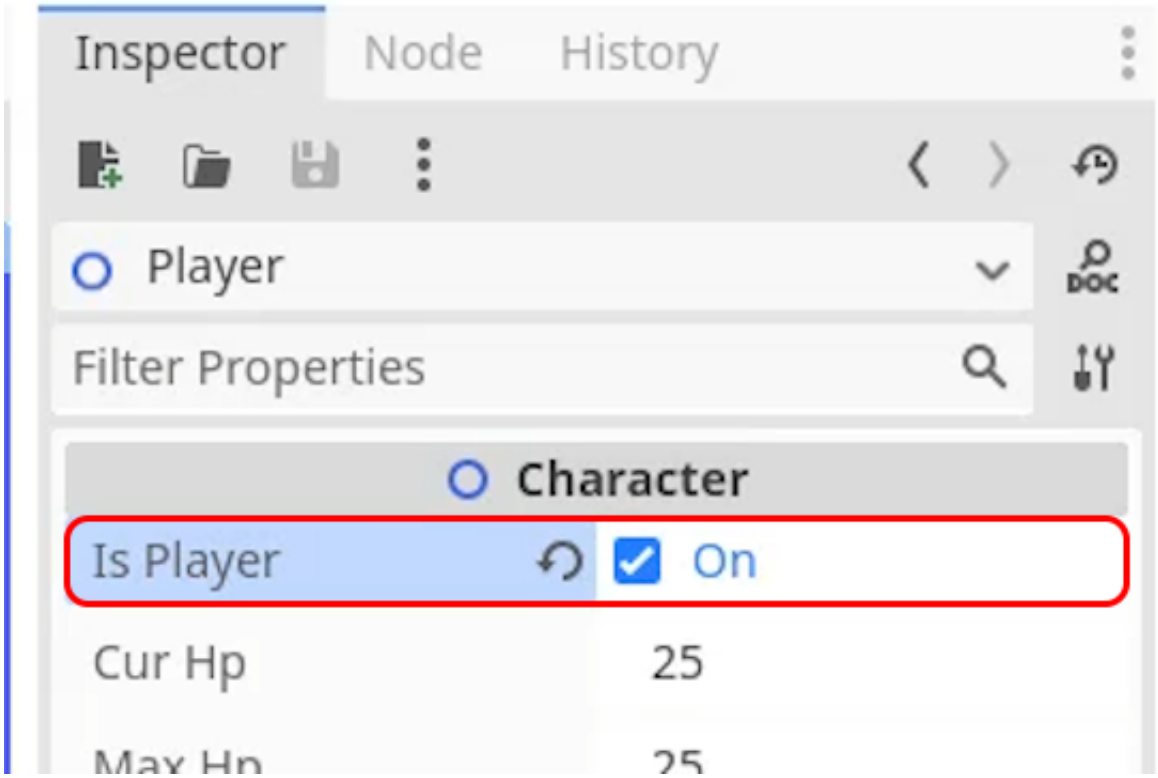
Then, move the duplicated character to the right side of the screen.



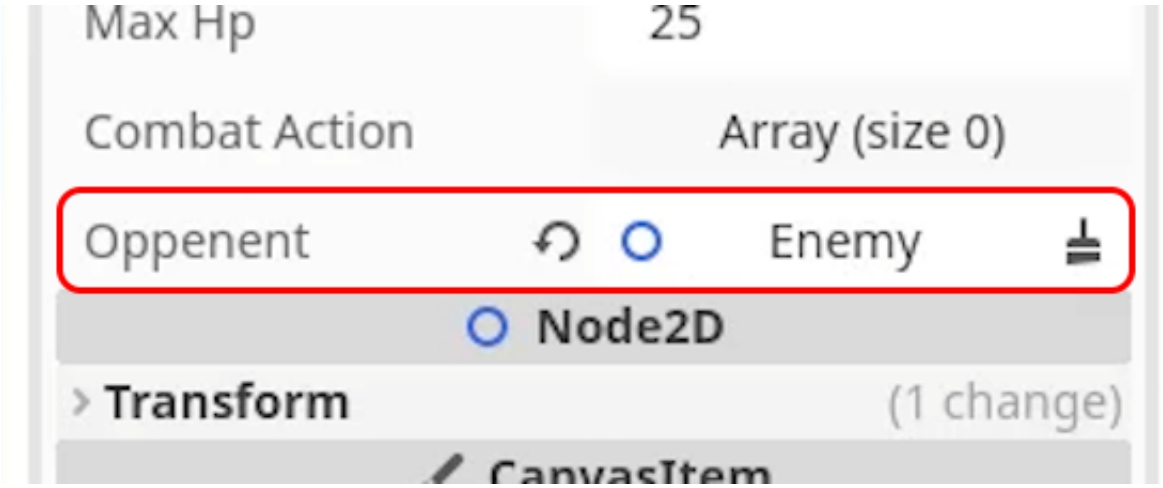
We can then **rename** the second character instance to **“Enemy”** so that it can be identified easily in the scene.



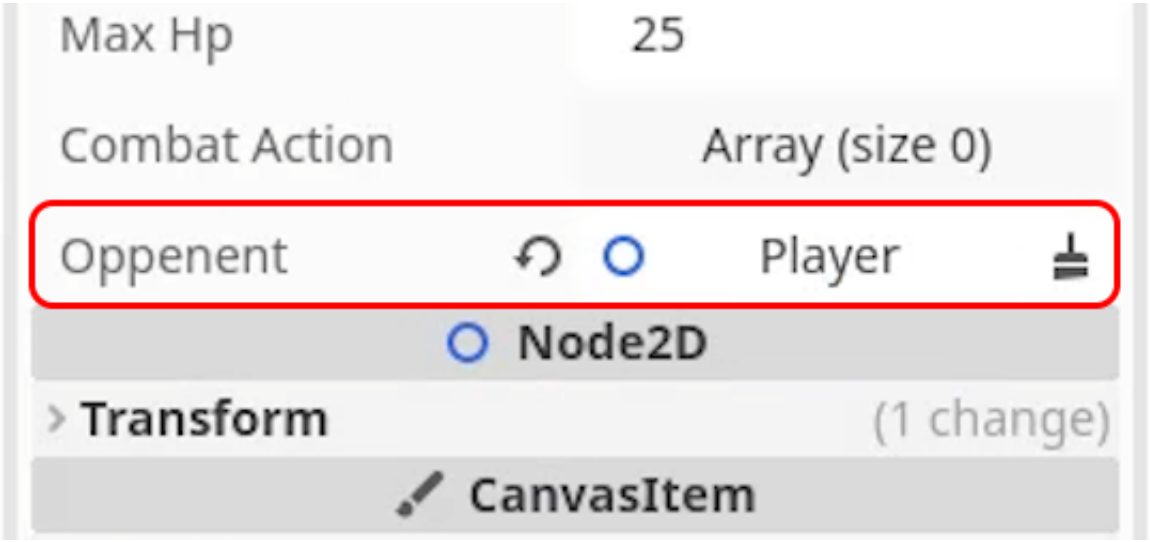
Finally, with the **Player** instance selected, we can enable the **Is Player** property in the *Inspector* window. This should not be enabled on the *Enemy* instance.



The *Character* script also needs the **Opponent** variable to be set in the *Inspector*. For the **Player** instance, we will drag and drop the **Enemy** instance as the value for this variable.



Inversely, for the **Enemy** instance, we will set the **Opponent** to be the **Player** instance.



Adding the Code

With both characters set up, we are ready to begin sending signals between the *TurnManager* script and the two *Character* instances. To do this, we will need a new function in the **Character.gd** script called **_on_character_begin_turn** with a parameter called **character**.

```
func _on_character_begin_turn(character):
    pass
```

For now, we will fill this function with the **pass** keyword so that we can add functionality to it later. Currently, we have nothing in place to call this function, to do this we will create a **signal** in the **TurnManager** script. In Godot, a *signal* is a type of event that functions can subscribe to, when the event is triggered (known as *invoking* the signal) the functions will then be called. Godot supplies some signals by default in the *Node* window. Our custom signals can be placed just below the variables in the script.

```
signal character_begin_turn(character)
```


This will define a signal called **character_begin_turn** with a parameter of **character**, which can be linked to the function we created in the *Character* script. We will also want to define another signal with the name of **character_end_turn** and a parameter of **character** again.

```
signal character_end_turn(character)
```

With the signals in place, we can call the *character_begin_turn* signal from within the **begin_next_turn** function. This can be done using the **emit_signal** function that comes as part of Godot.

```
func begin_next_turn():
    ...
    emit_signal("character_begin_turn", cur_char)
```

Here we pass two parameters to the *emit_signal* function. The first parameter is the *name* of the signal, as we defined it at the top of our script and the second parameter is our *character* parameter defined in the signal. For the *character* parameter, we pass through the *cur_char* variable. We can do the exact same with the **end_current_turn** function and the **character_end_turn** signal.

```
func end_current_turn():
    emit_signal("character_end_turn", cur_char)
    ...
```

It is important to place this before the *begin_next_turn* call in the *end_current_turn* function, otherwise, the current character will be set to the wrong value. We now need to connect the *Character* script to the *TurnManager* signals. We will do this by getting the root node of our scene, called **BattleScene**, as this is where our *TurnManager* script is attached, and connecting the **on_begin_character_turn** function to the **character_begin_turn** signal. This needs to be done in the **_ready** function to ensure it is run at the start of the game.

```
func _ready():
    get_node("/root/BattleScene").character_begin_turn.connect(_on_character_begin_tu
rn)
    ...
```

This will ensure that whenever the *TurnManager's character_begin_turn* signal is emitted, the *on_begin_character_turn* function in the *Character* scripts will also run. We also need to set up the code to update the *TurnManager* when a character dies. We will do this in the **take_damage** function of the *Character* script. Before we call the **queue_free** function we need to let the *TurnManager* know the character has died, to do this we will access the *BattleScene* node again, using **get_node**, and call the **character_died** function.

```
func take_damage(damage):
    ...
    if cur_hp <= 0:
```



```
        get_node("root/BattlScene").character_died(self)
        queue_free()
```

We use the **this** keyword to send the current *Character* instance as the character parameter for the function.

Changing the Enemy Visuals

The final thing we can do in this lesson is change the visual asset our enemy *Character* instance is using. Because our enemy and player characters are both instances of the same *Character* scene, we can't change the enemy instance directly, so instead we will add some code to do it for us. The first step will be to add two new variables to the **Character** script. The first will be called **visual** and be of type **Texture2D**, which we will **export** so that it is editable in the *Inspector* window. This will be the sprite asset that we replace our *Sprite* node with.

```
@export var visual : Texture2D
```

The second variable will be called **flip_visual**, be of type **bool**, and also be **exported**. This will flip the *Sprite* if true by modifying the *Flip H* property. This will be useful for making the enemy face the player.

```
@export var flip_visual : bool
```

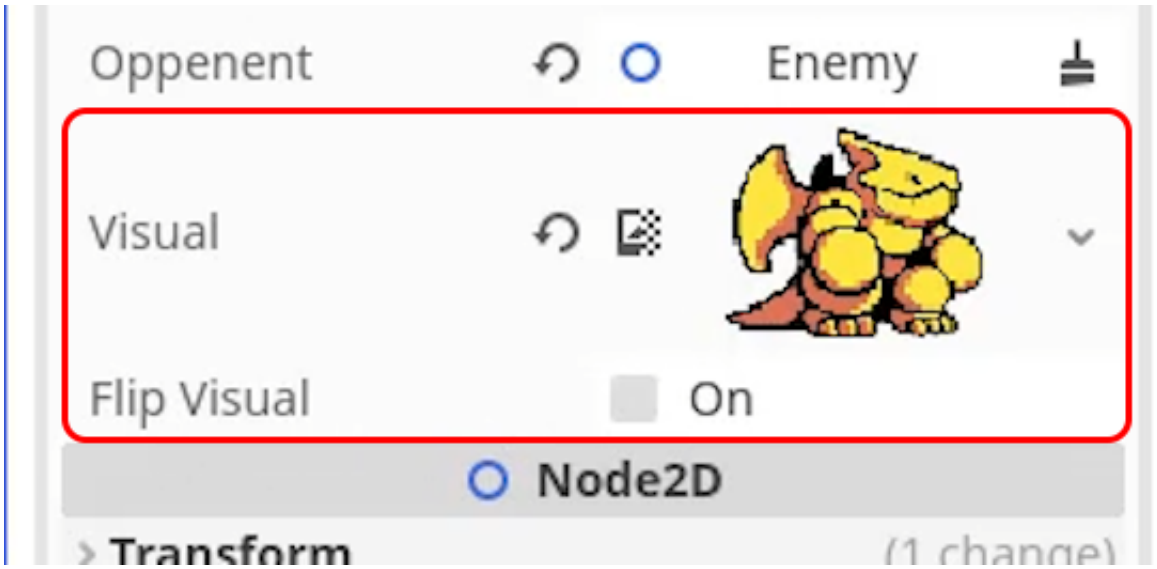
We can then use these variables inside the **_ready** function. First of all we will assign the texture of the *Sprite* node to our new *visual* variable.

```
func _ready():
    $Sprite.texture = visual
    ...
```

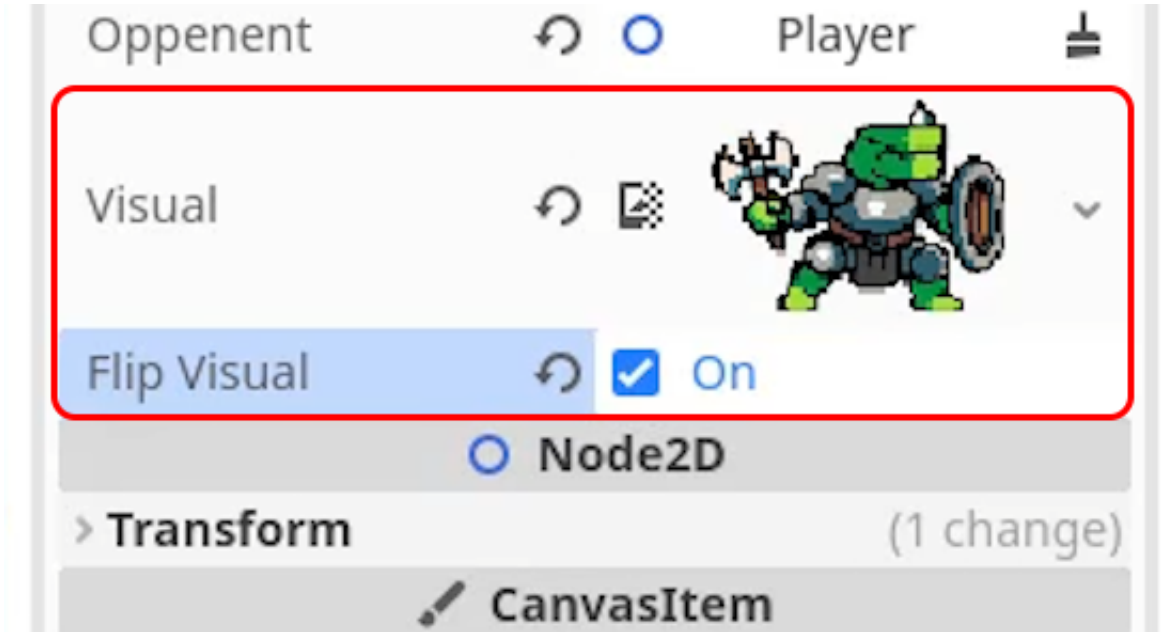
As we aren't going to be accessing the *Sprite* component outside of the *_ready* function, there's no need to set it up as a variable, so we will make use of Godot's built-in **$** access method to access it. We can then assign the **flip_h** variable of the *Sprite* to be the same as the **flip_visual** variable.

```
func _ready():
    ...
    $Sprite.flip_h = flip_visual
    ...
```

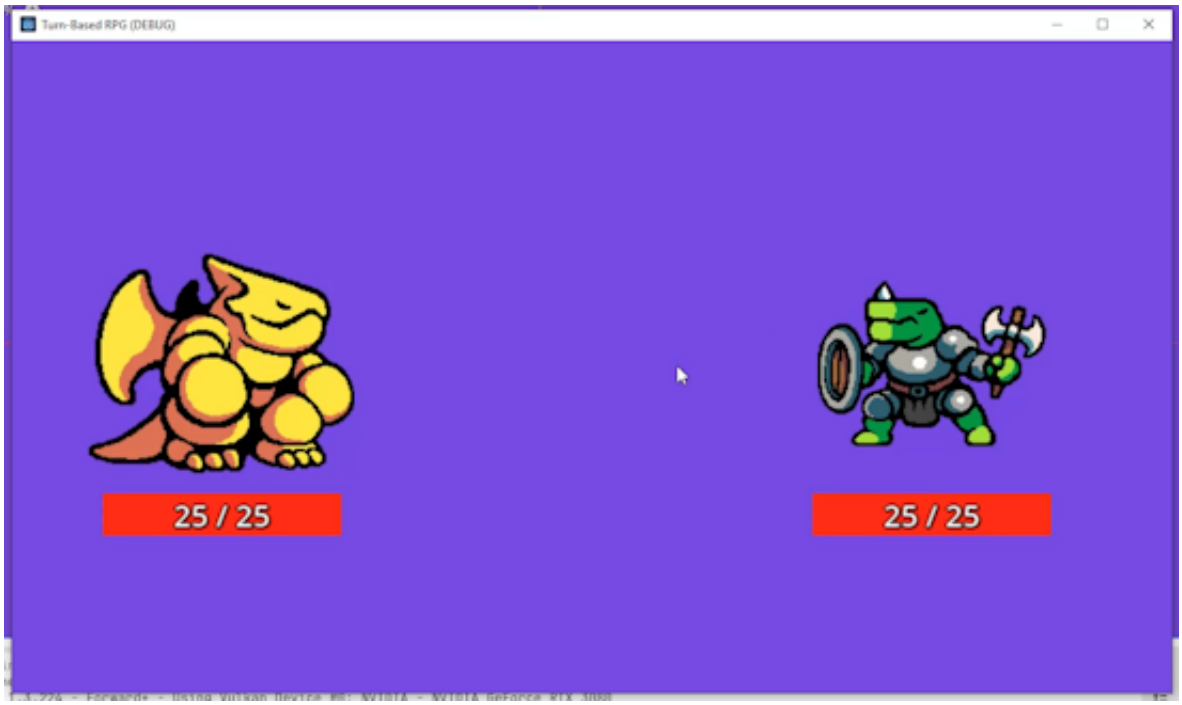
With this in place, we can begin assigning the variables in the **BattleScene** scene. For our **Player** instance, we want to assign the *Dragon.png* asset as our *Visual,* and make sure *FlipVisual* is still untagged.



On the **Enemy** instance we will set the *Visual* to *Reptile.png* and tick the *FlipVisual* property so that the enemy will face the player.



Now if you press **play**, you will see there are two character instances, and our enemy has a different sprite and faces the correct direction.



In the next lesson, we will begin setting up our combat actions for the *Player* instance, so that we can see the functions we have created in action.

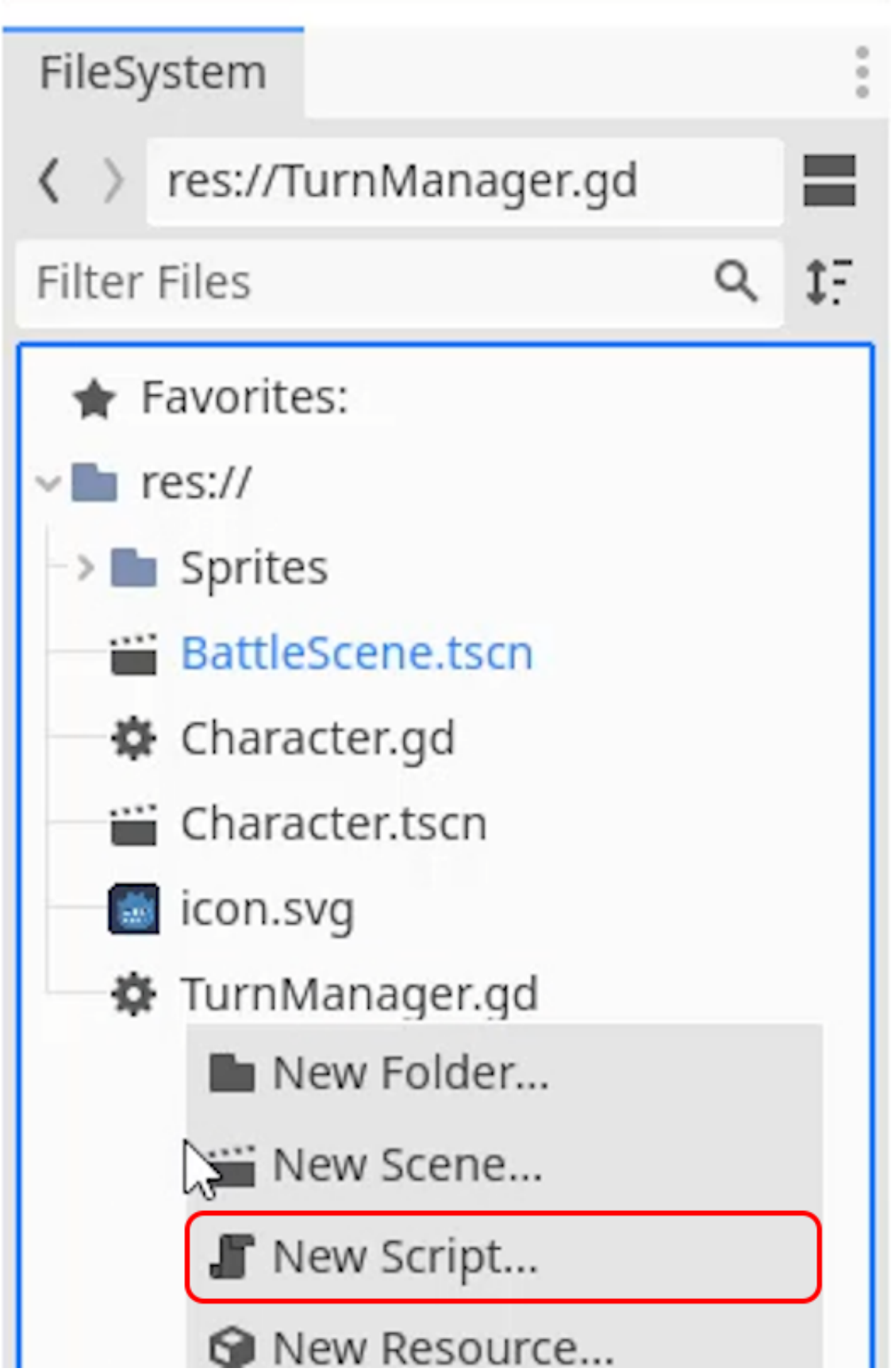
With our *Character* instances and our *TurnManager* set up, we are ready to begin implementing the combat actions that will make up a major part of this game. In our game, a *Combat Action* will be an action the player or the enemy can do when it is their turn. These actions will be used to deal damage to the other *Character* or heal the current *Character,* however, we will be making the system in such a way you can expand it to add more features if you would like. Our game will include three basic combat actions:

- Basic Attack, which deals damage.
- Heavy Attack, which deals more damage than the basic attack.
- Heal, which heals the character who is using the action.

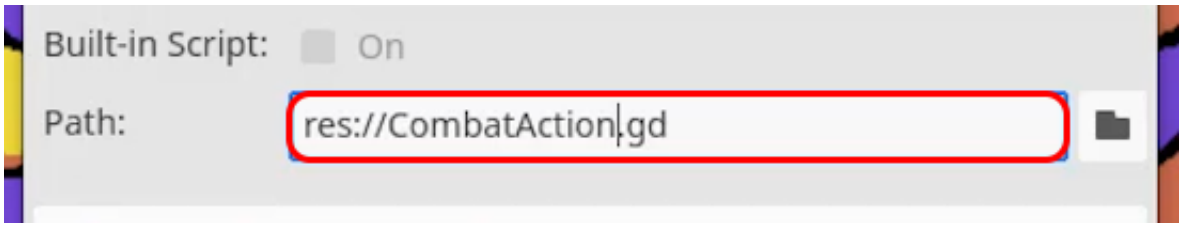
All of these actions also need to be interchangeable so that we can give the player different actions to the enemy.

Using Resources

Because we want to have multiple, interchangeable combat actions, we don’t want to hard code each action into the *Character* script. This is where *Resources* come into play. A *Resource* in Godot is effectively a custom asset that will allow us to store the information needed for each combat action. To create a new custom *Resource*, we first need to create a **New Script** by right-clicking in the *FileSystem* window.

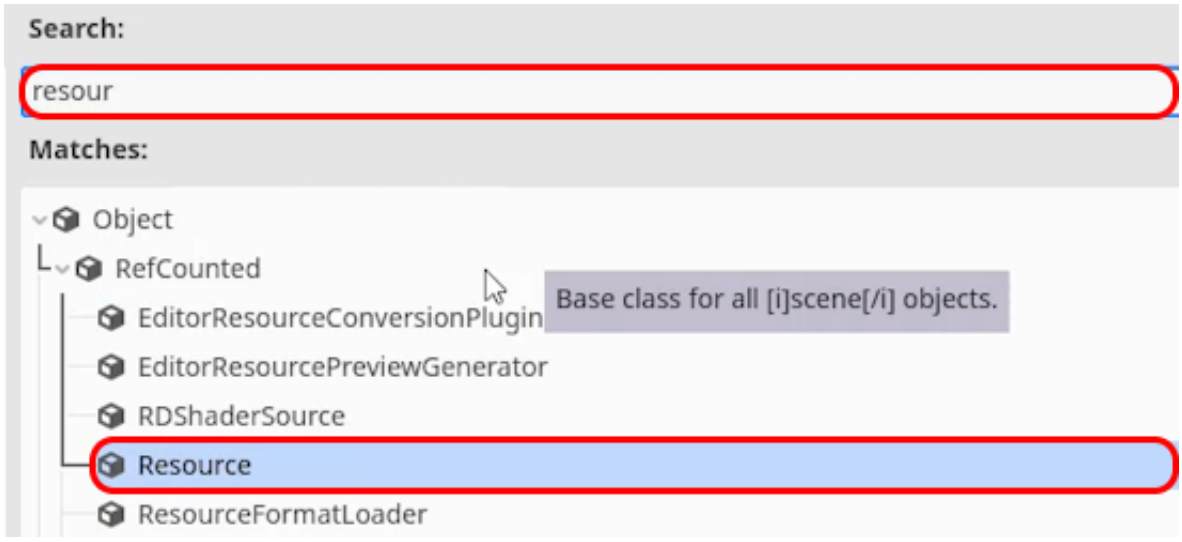


We will call this “**CombatAction”**.



This script also needs to inherit from the *Resource* class by selecting the **tree structure button**.

Select the **Resource** class as the class to inherit from.



With these values selected, we are ready to choose **Create** and open up the script for editing. Before we can begin adding information to the new resource, we first need to give the script a **class_name** definition. This will make it possible to create the *Resource* instance in the *FileSystem* window along with identifying it in scripts. For this class name, we will use the name **CombatAction**.

```
class_name CombatAction
```

Creating the Variables

We are now ready to set up the properties for this *Resource.* Each of these variables will be **exported** so that we can edit them across each instance of the *CombatAction* resource. The first variable will be called **display_name**. This will define the name that is shown to the player, so we can put a placeholder of **“Action (x DMG)”** in place as a default value.

```
@export var display_name = "Action (x DMG)"
```


Next, we will create a variable called **damage**, which will have a type of **int** and have a default value of **0**. This will be the damage that the *Character* applies to their opponent when the action is used.

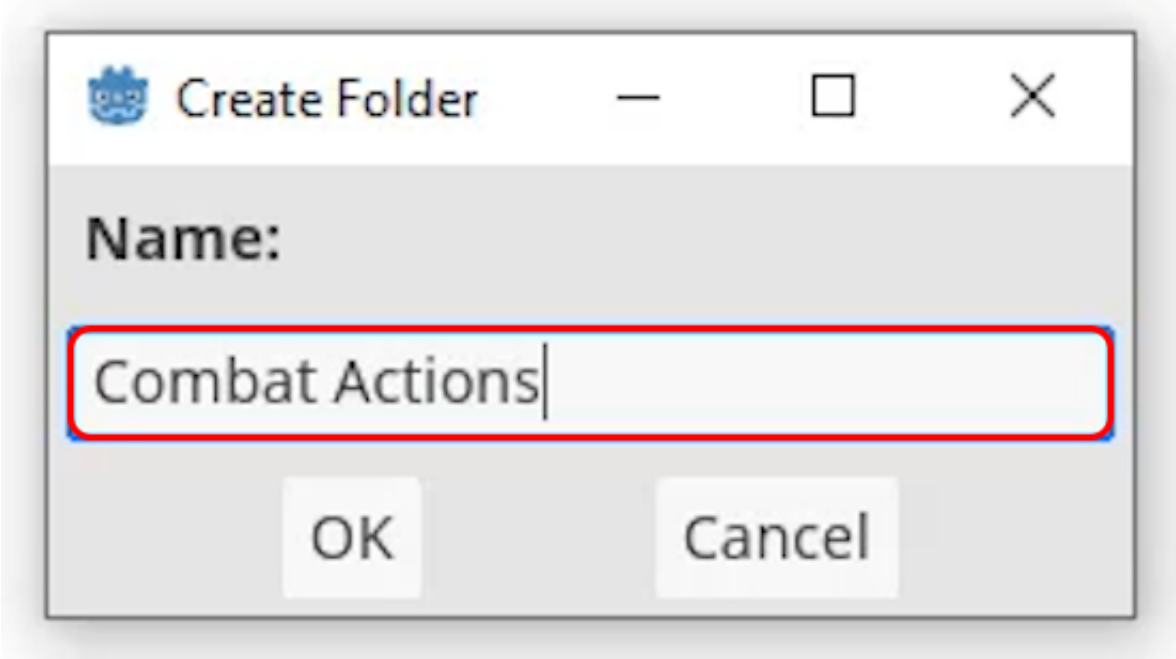
```
@export var damage : int = 0
```

We can copy this variable for the **heal** property.

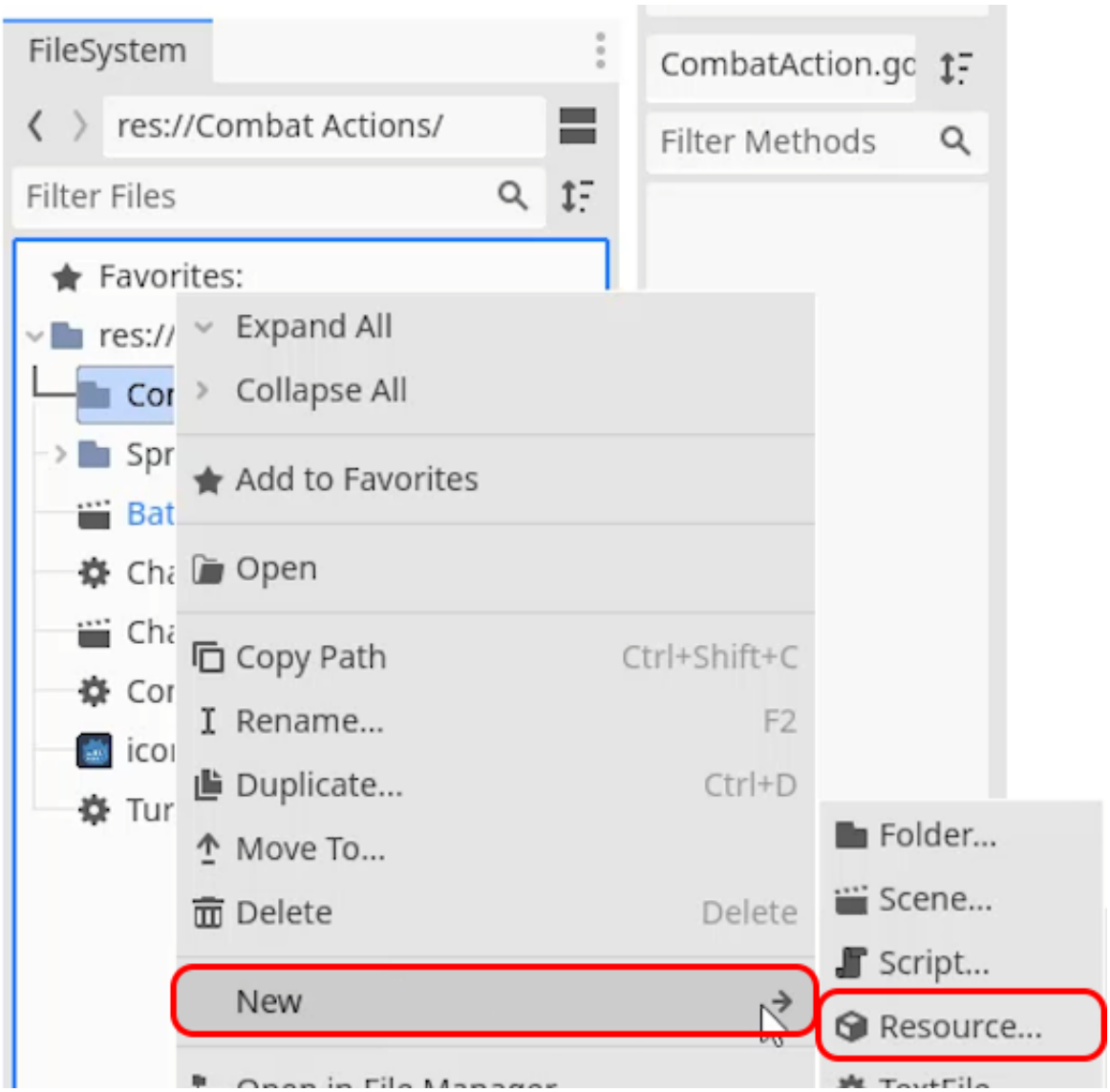
```
@export var heal : int = 0
```

Creating Combat Actions

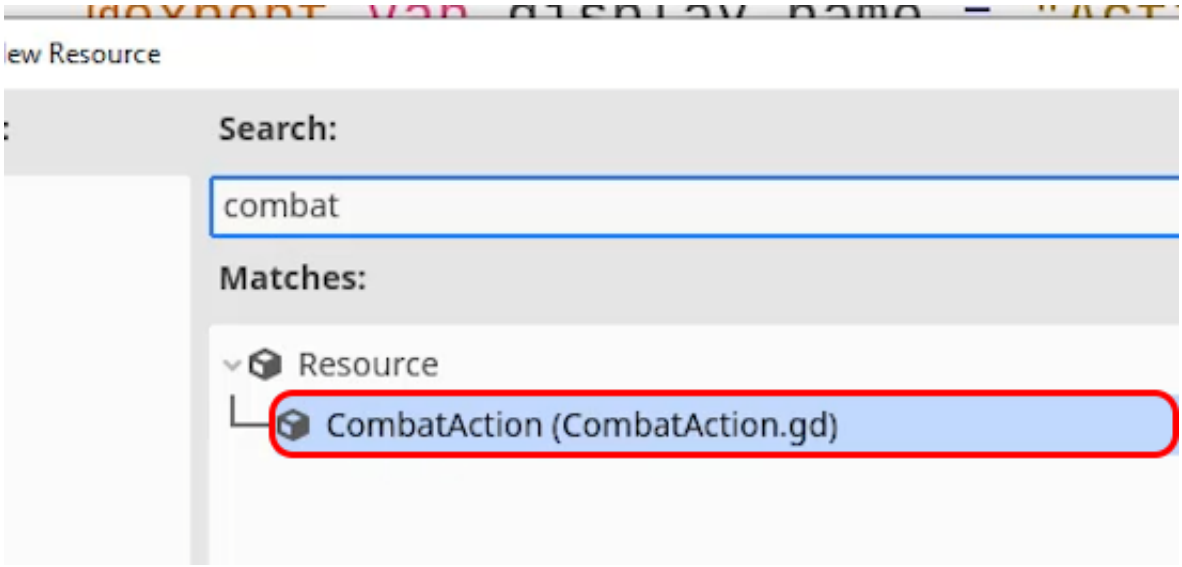
With the variables in place, we can begin creating each instance of the new *CombatAction* resource. We will do this in a **new folder** inside of the *FileSystem* window to keep the project organized.



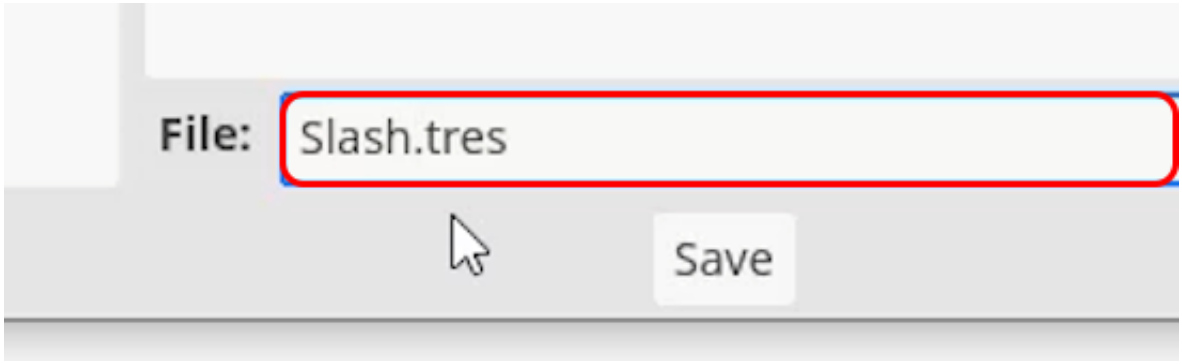
Inside this folder, we can create our *Resources*. To do this select the **new -> Resource** option in the right-click menu.



A *Resource* is just another name for an asset in our project. Although we often won't access assets like scenes and sprites from this menu, you can create any type of *Resource* here. Because our *CombatAction* inherits from *Resource*, we are able to create them as resources within the *FileSystem*. This means that we don't need to instantiate the *CombatAction* as a node inside the scene to use it, instead, we can access it directly from the asset. To create the resource, simply select **CombatAction** from the resource tree.

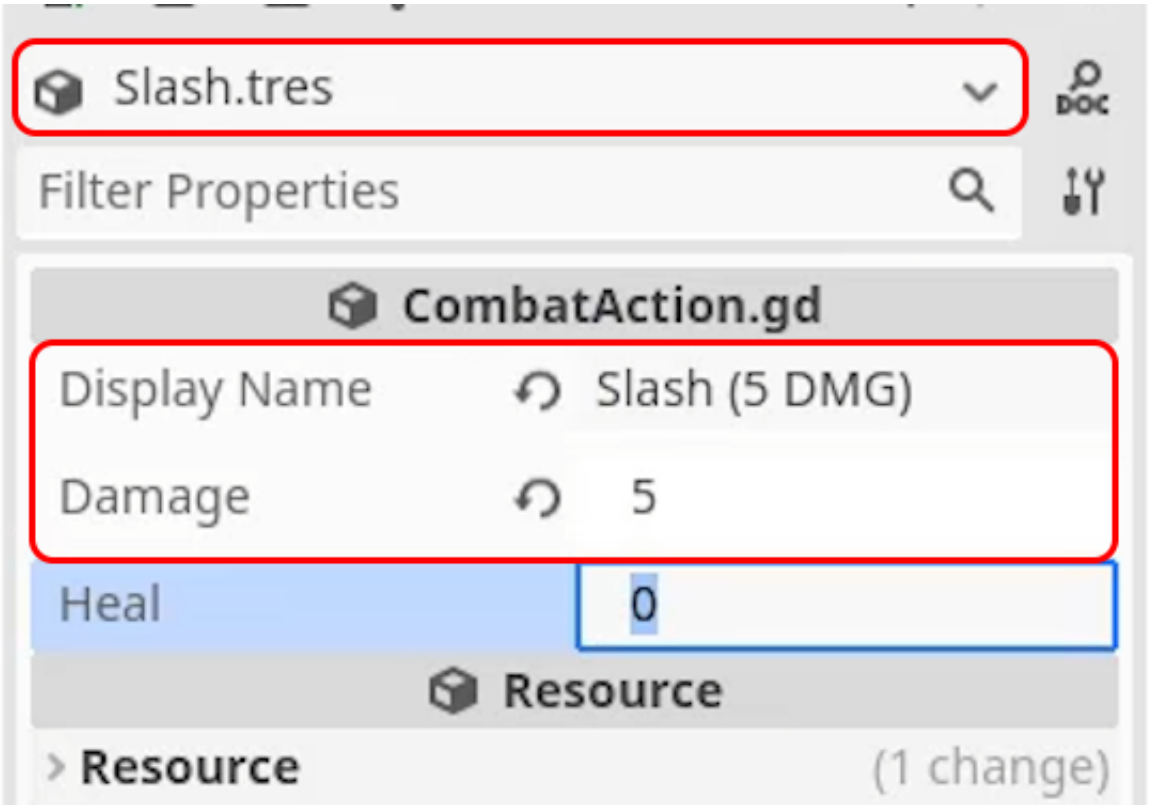


We can name this “**Slash.tres”** as that will be the name for our base attack action.

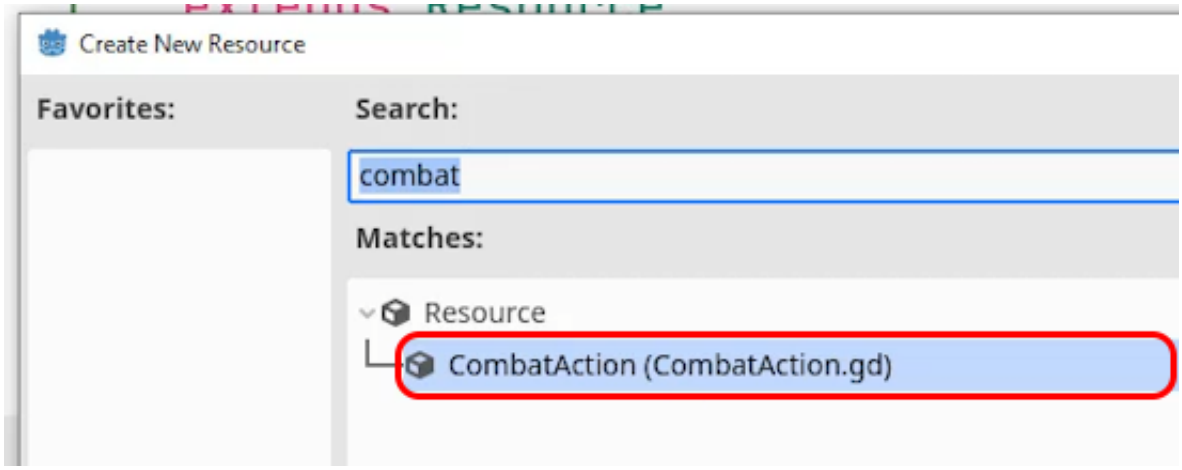


With the **Slash.tres** resource select in the *FileSystem*, you will notice the properties we have created are editable in the *Inspector,* similar to node instances in a scene. For our **Slash** resource, we want the following values:

- **Display Name:** Slash (5 DMG)
- **Damage:** 5
- **Heal:** 0

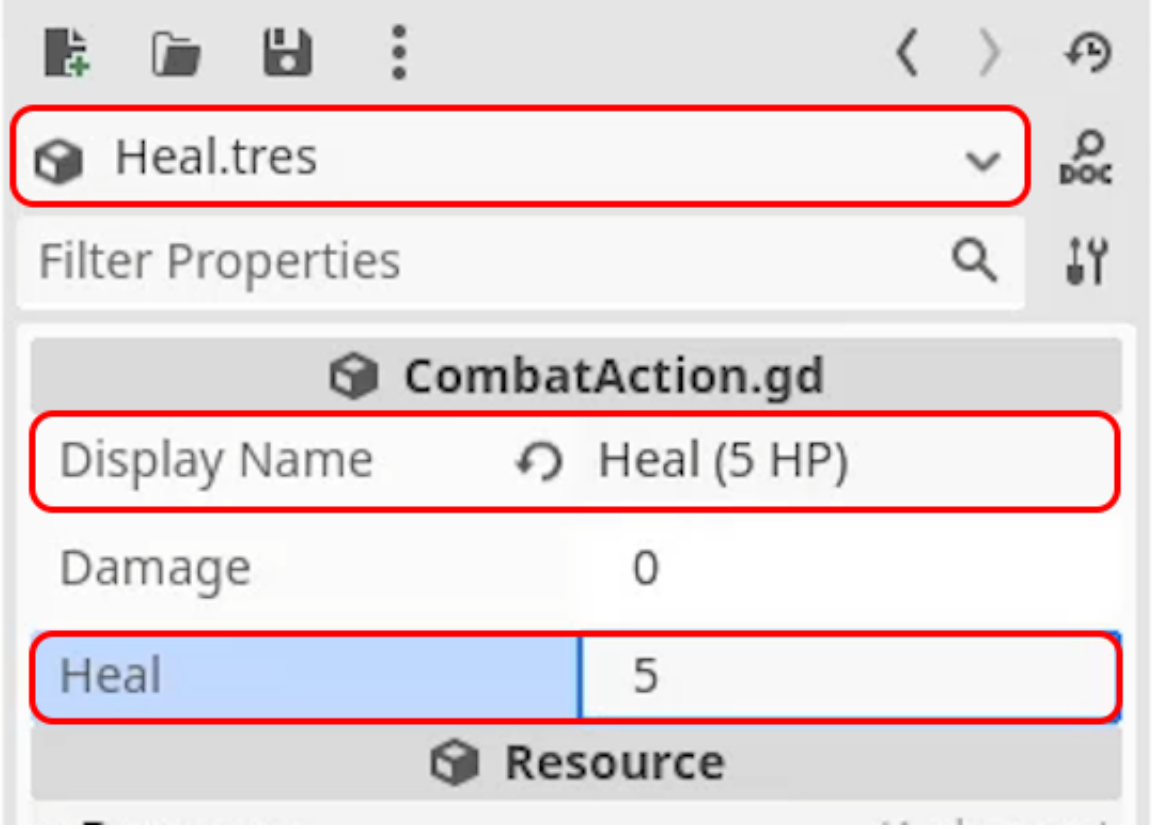


We can now easily create a second **CombatAction** resource with different values. This can be called **“Heal.tres”**.



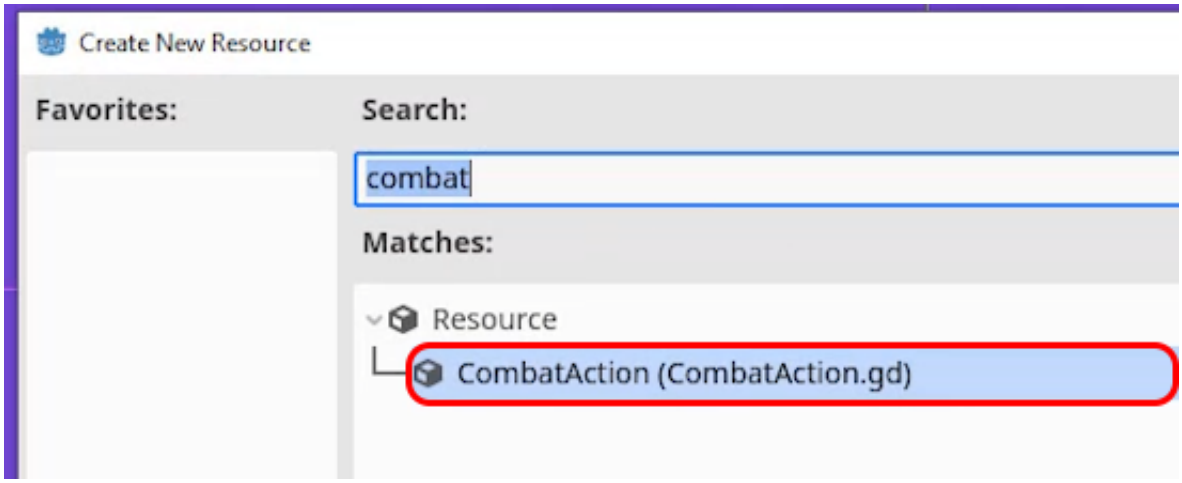
For the **Heal** resource, we will use the following values:

- **Display Name:** Heal (5 HP)
- **Damage:** 0
- **Heal:** 5



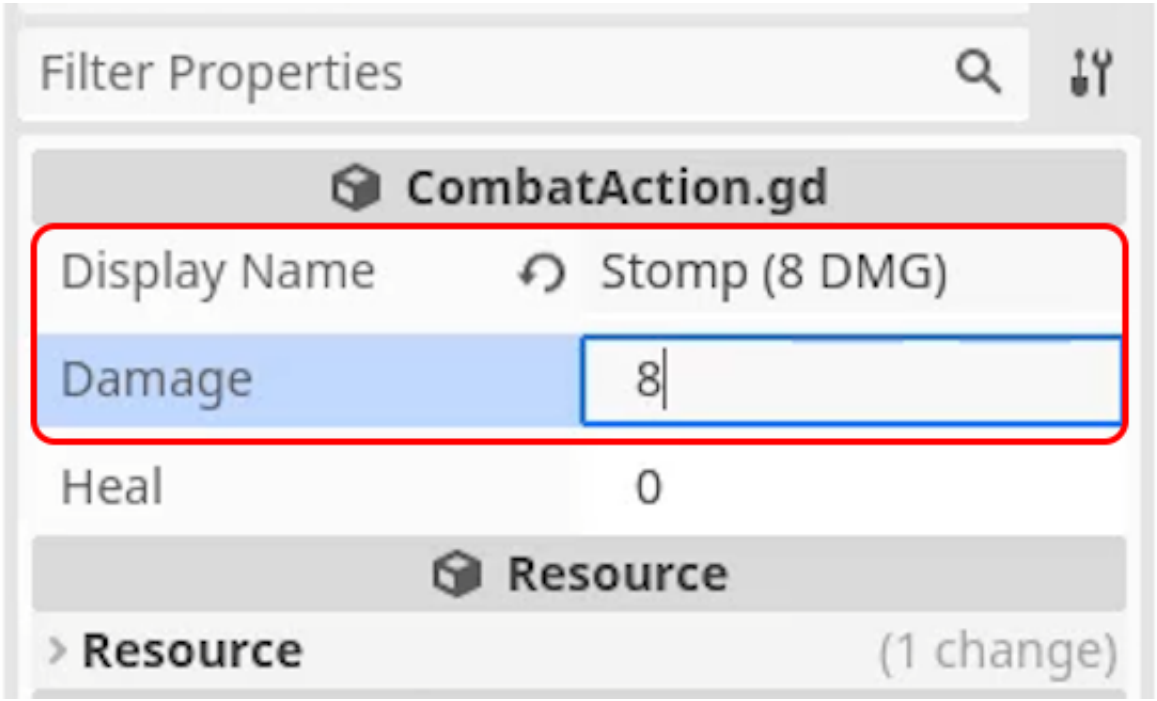
As a challenge, before the next lesson try creating another combat action called **Stomp**. This action will deal **8 damage** to the opponent.

Hopefully, you had a good go at the challenge set in the previous lesson! To create our **Stomp** combat action, we will begin by creating another *CombatAction* resource.



We can call this **“Stomp.tres”** to follow the naming scheme set up in the previous lesson. For the properties of this action, we will use the following:

- **Display Name:** Stomp (8 DMG)
- **Damage:** 8
- **Health:** 0

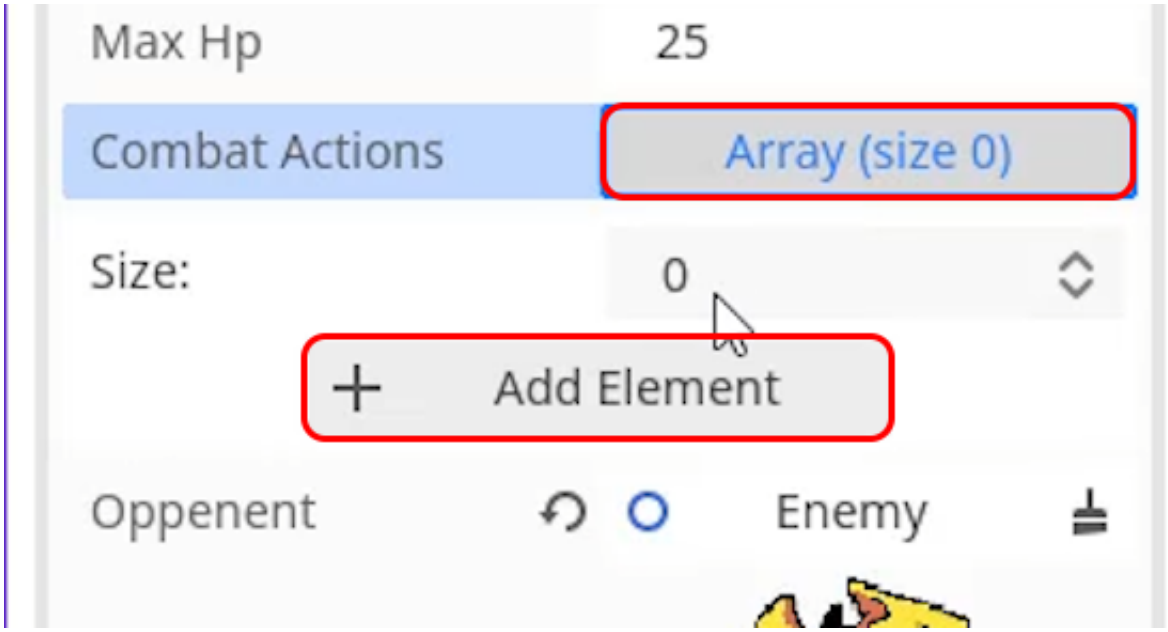


Modifying the Character Script

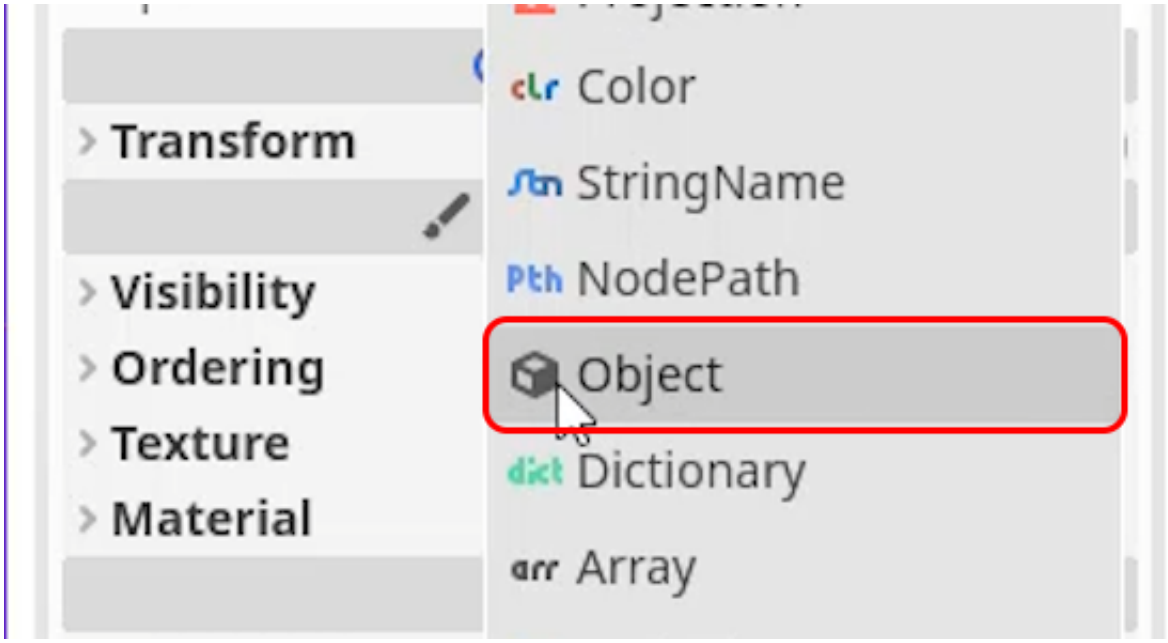
Currently, we have no way to assign our combat actions to our *Character*. Luckily, as we have created them as *Resources* this will be no problem. Previously, we created a variable called **combat_action** in the *Character* script. This should be renamed to **combat_actions** to better show there are multiple elements in this variable.

```
@export var combat_actions : Array
```

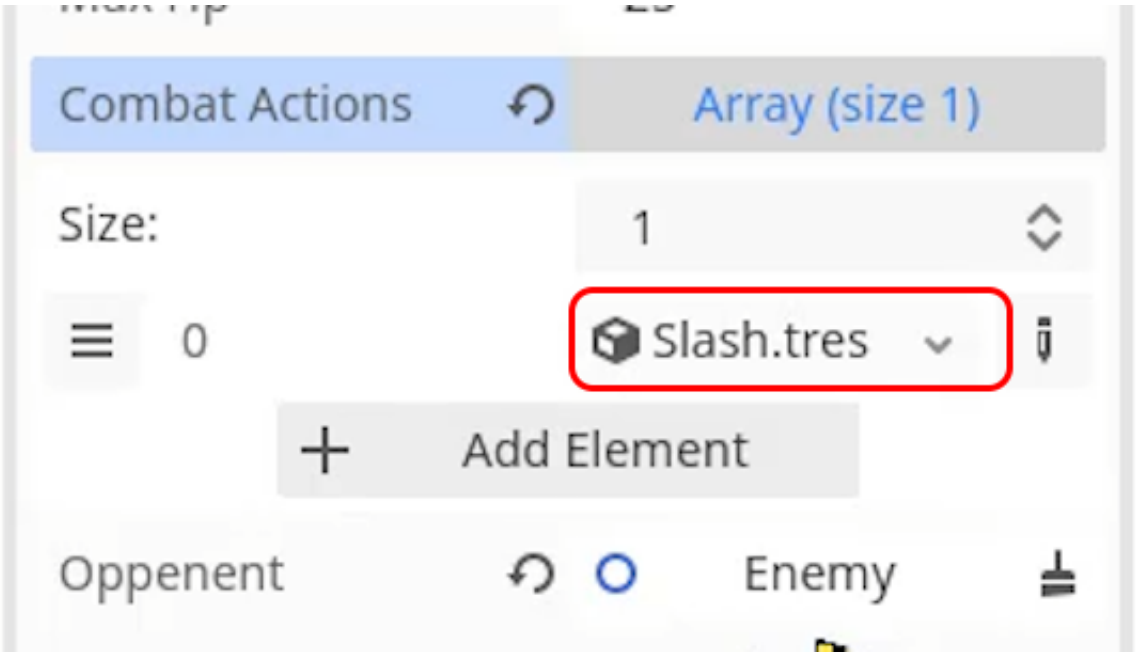
The *combat_actions* variable is given a type of **Array**. This type allows us to store multiple items inside one variable in a list format. With this in mind, we can select the **Player** node and begin adding elements to our *Combat Actions* array from the *Inspector* window. Press the **Add Element** button to add room for new properties in the array.



By default, Godot does not know we want our *Array* to contain *CombatActions*. To fix this, select the **pencil button** to the right of the new element and select the **Object** type.

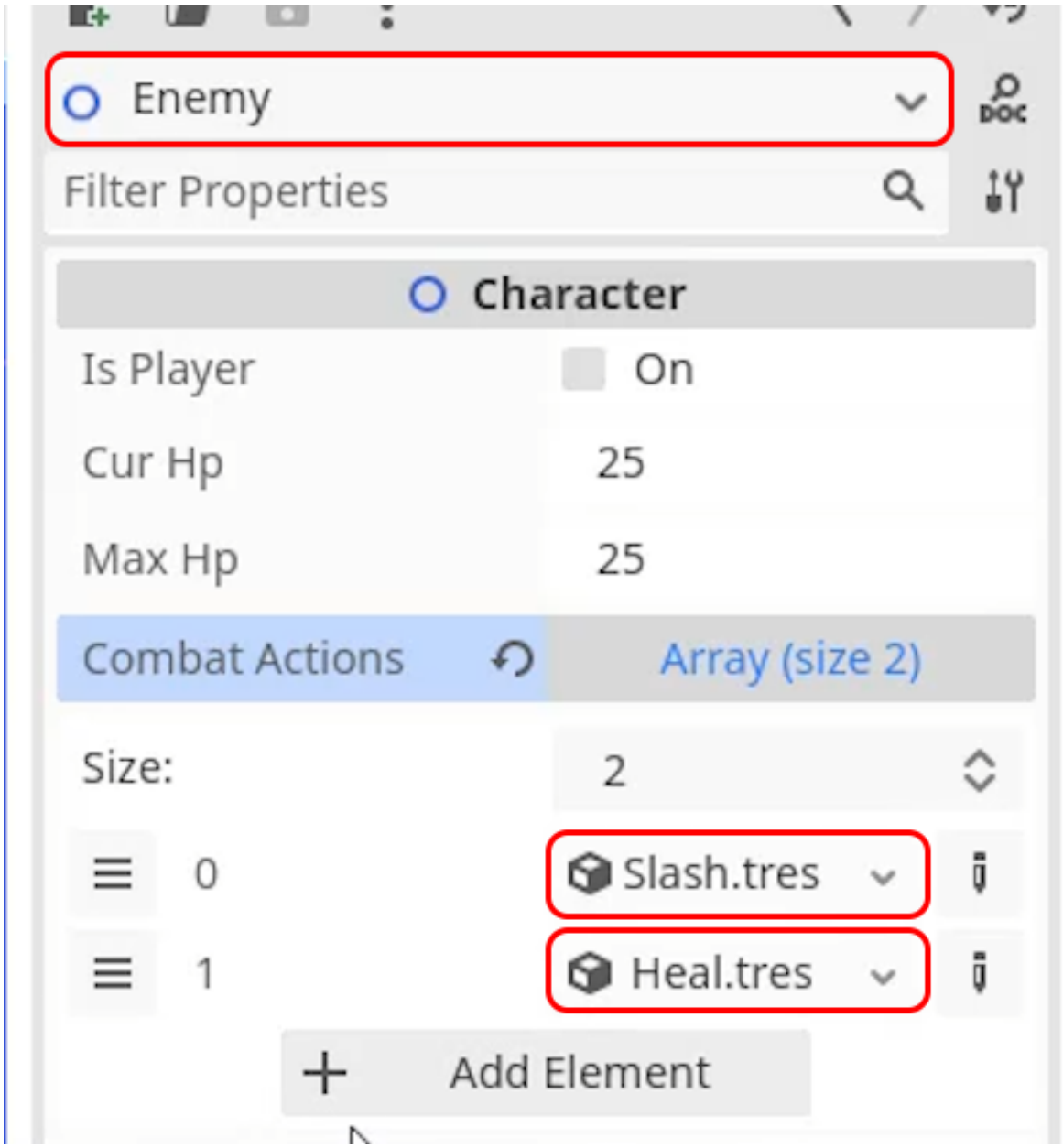


This will then allow us to drag our combat action into the new property.



This process can then be repeated for both the **Heal** and the **Stomp** combat actions.

We can then repeat this process for the **Enemy** character. However, to make the game more playable, we don't want to give our enemy the **Stomp** combat action, so we can simply not add this *Resource*.

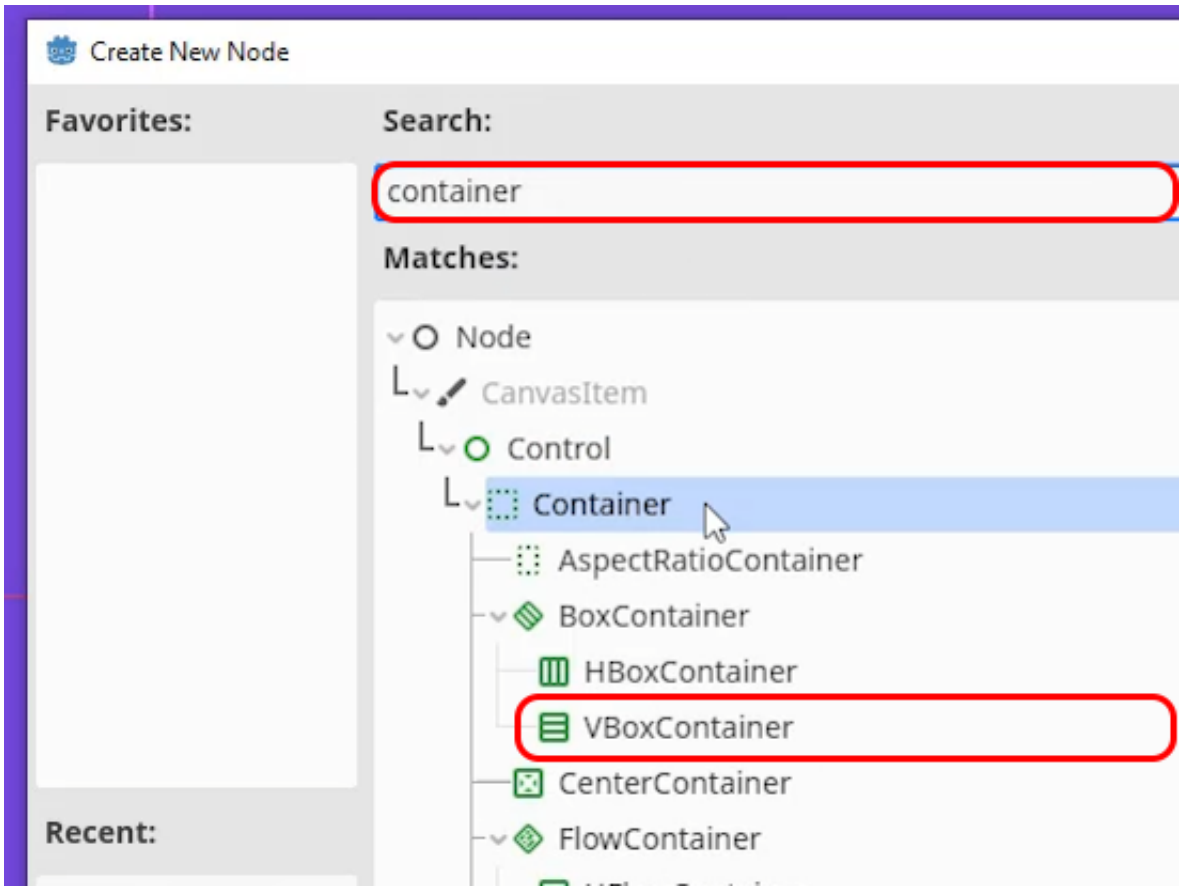


This means we now have all the code in place to create the gameplay systems. In the next lesson, we will begin creating the Player's UI for selecting the combat actions.

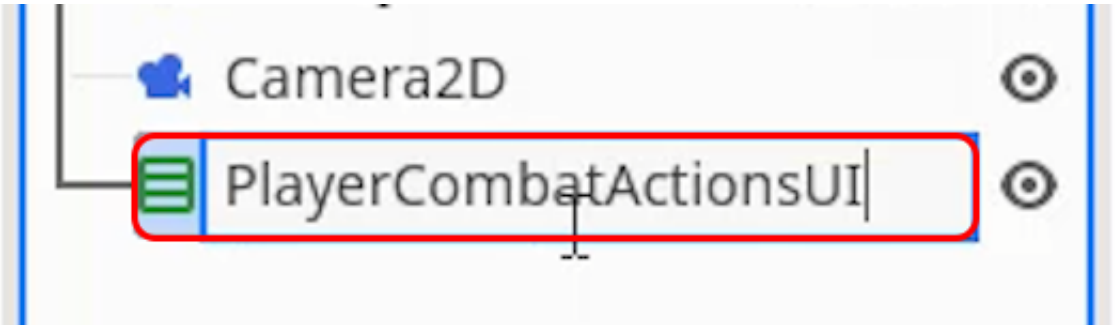
Our game now needs a user interface to allow our players to select the combat action they want to use. The idea is to create a list of buttons that appear below the *Player* character instance whenever it is their turn, when the button is clicked, this will trigger the combat action to be used.

Creating the UI

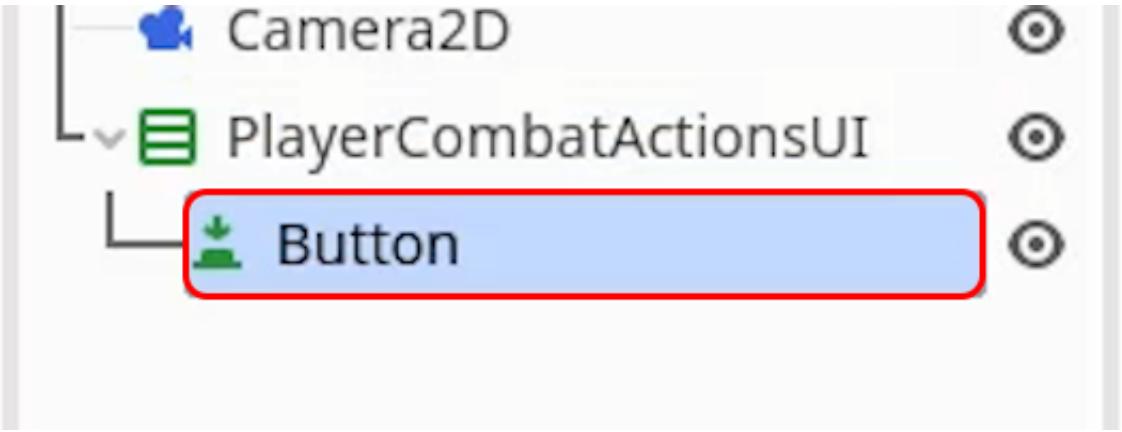
To begin with, we will create a new **VBoxContainer** node in the **BattleScene**. This will act as a container for our UI, that automatically arranges our buttons into a vertical list.



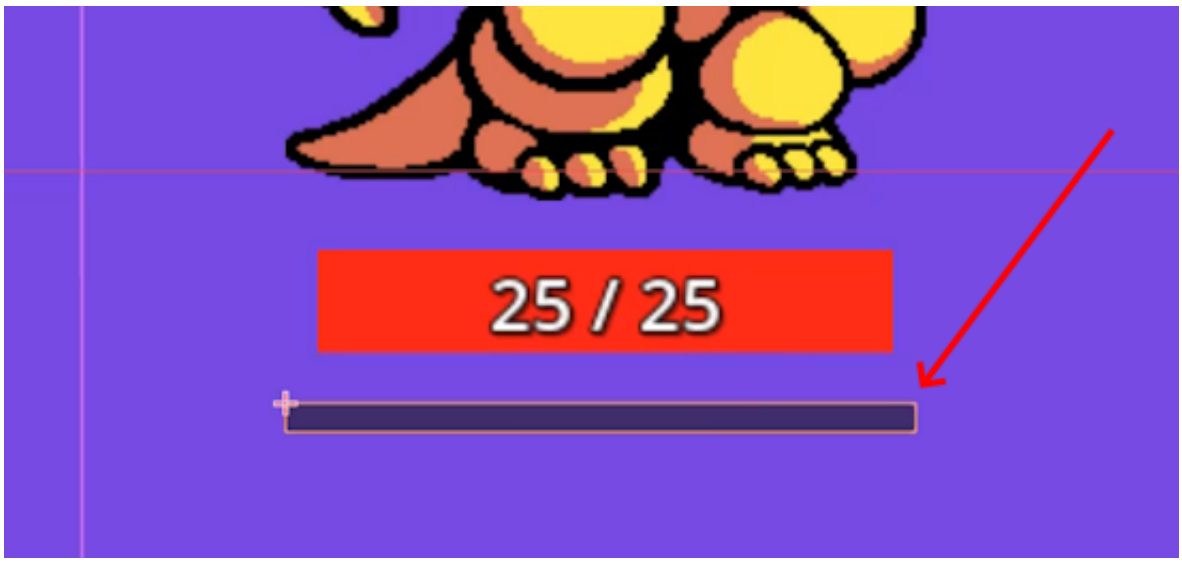
We can **rename** this node to **“PlayerCombatActionsUI”**.



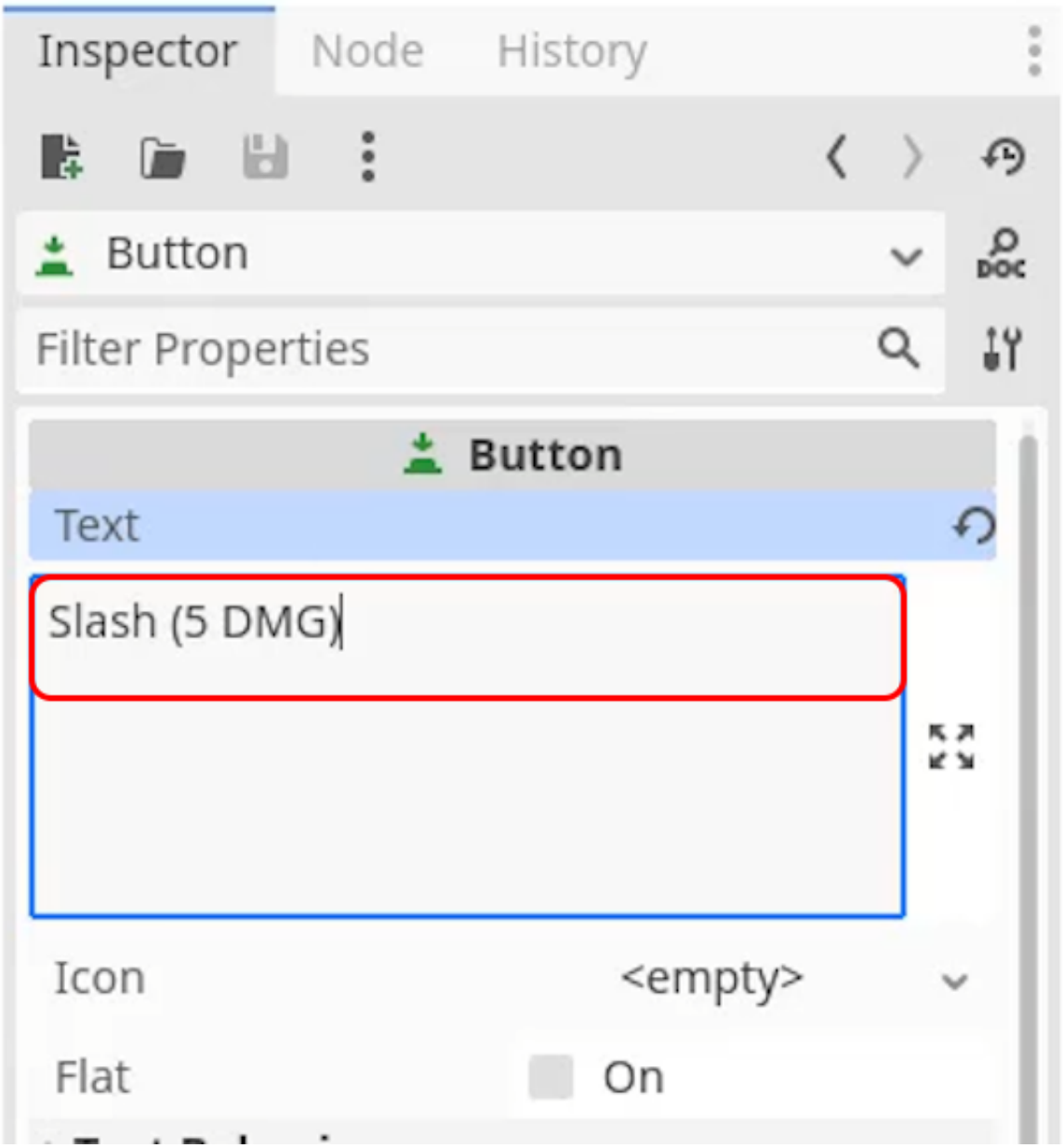
We can then add a **Button** as a child node to this container.



As you will see, because we use a *VBoxContainer* to handle the UI positioning, the button is automatically scaled to the width of the container.



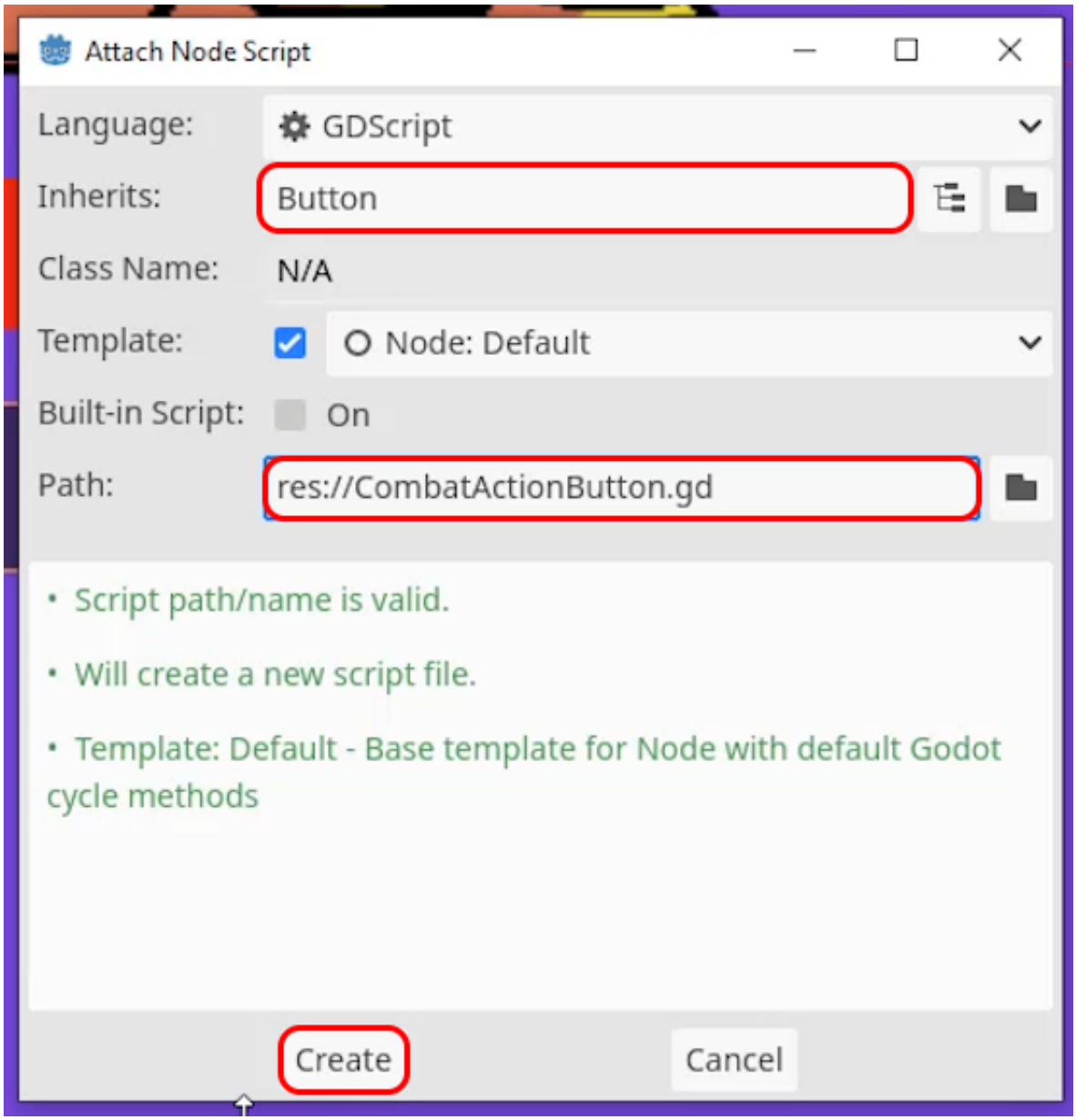
We probably want to make this a bit taller. The best way to do that is to enter some default text on our button, such as *“Slash (5 DMG)”*.



This will allow us to visualize the buttons in the scene, however, we will be changing this text through the code later on.

Button Script

We can now implement some of the functionality for our button. The best way to do this will be to add a new script directly to the *Button* node. We can name this **CombatActionButton**, ensure it inherits from **Button** and press **Create**.



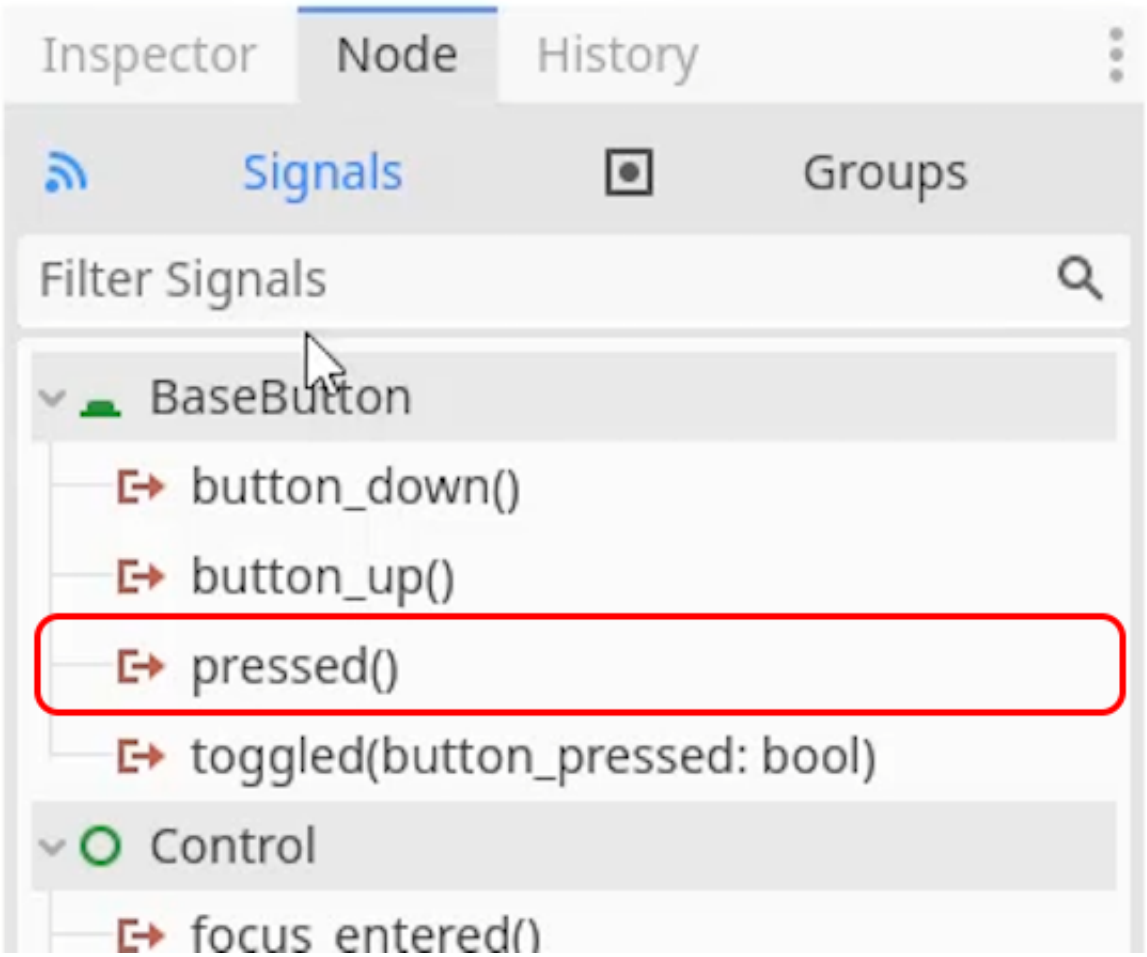
Inside this script, we can remove the *_ready* and *_process* functions as we won't be needing them. Our first variable will be called **combat_action** and have a type of **CombatAction**. This will allow us to assign the button with a specific *CombatAction* resource, allowing our script to know the effects that should be applied when the button is pressed. Our buttons will be passed the combat action when it is the player's turn, so we don't need to give a default value.

```
var combat_action : CombatAction
```

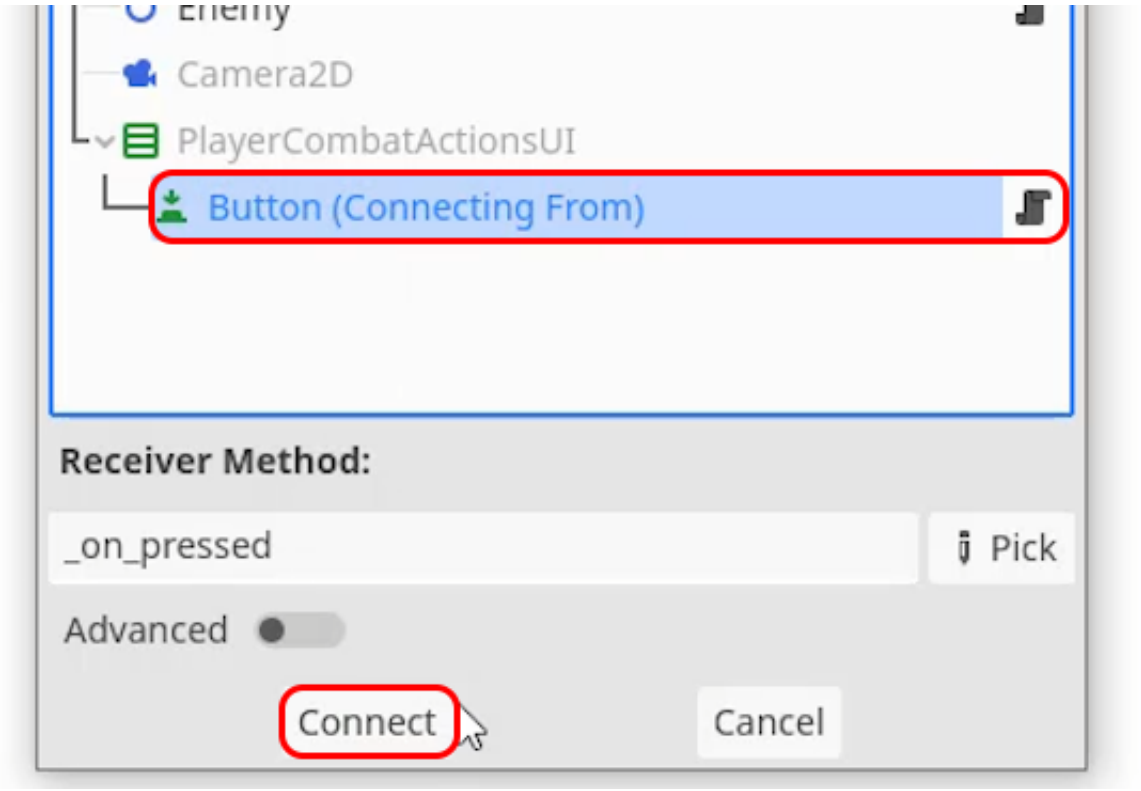
Before handling the click events on our button, we first want to set up a function in our **Character** script that can be called when the button is clicked. This will be called **cast_combat_action** and have a parameter of **action**, which will be the *combat_action* value of the button that was pressed.

```
func cast_combat_action (action):
    pass
```

We can now handle button clicks inside the **CombatActionButton** script. To listen for the button being clicked, we can add a **signal** to the node. We will use the **pressed** signal for this.



Make sure to assign the *pressed* signal to the **Button** node before clicking **Connect**.



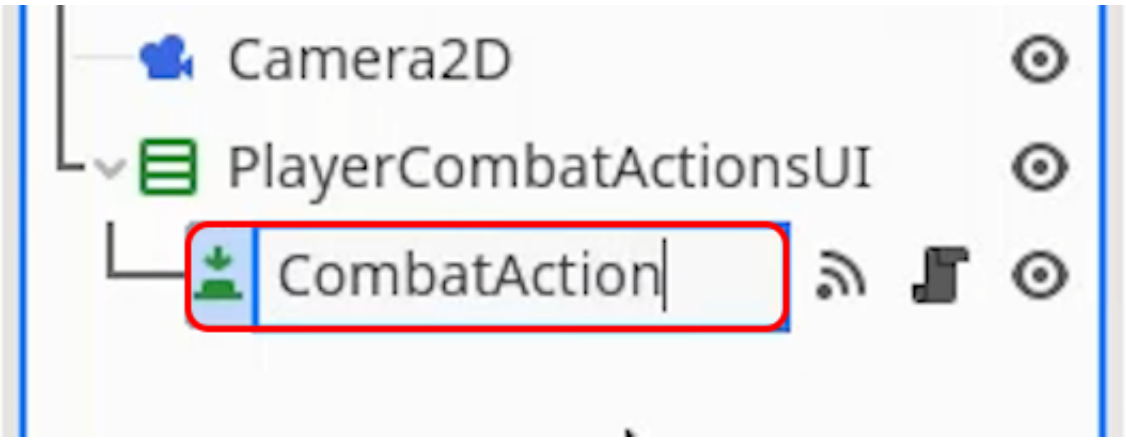
This will generate a function called **_on_pressed** inside our button script, which will be automatically called whenever the button is pressed by the player. Here, we can call the **cast_combat_action** function on the current character.

```
func _on_pressed():
    get_node("/root/BattleScene").cur_char.cast_combat_action(combat_action)
```

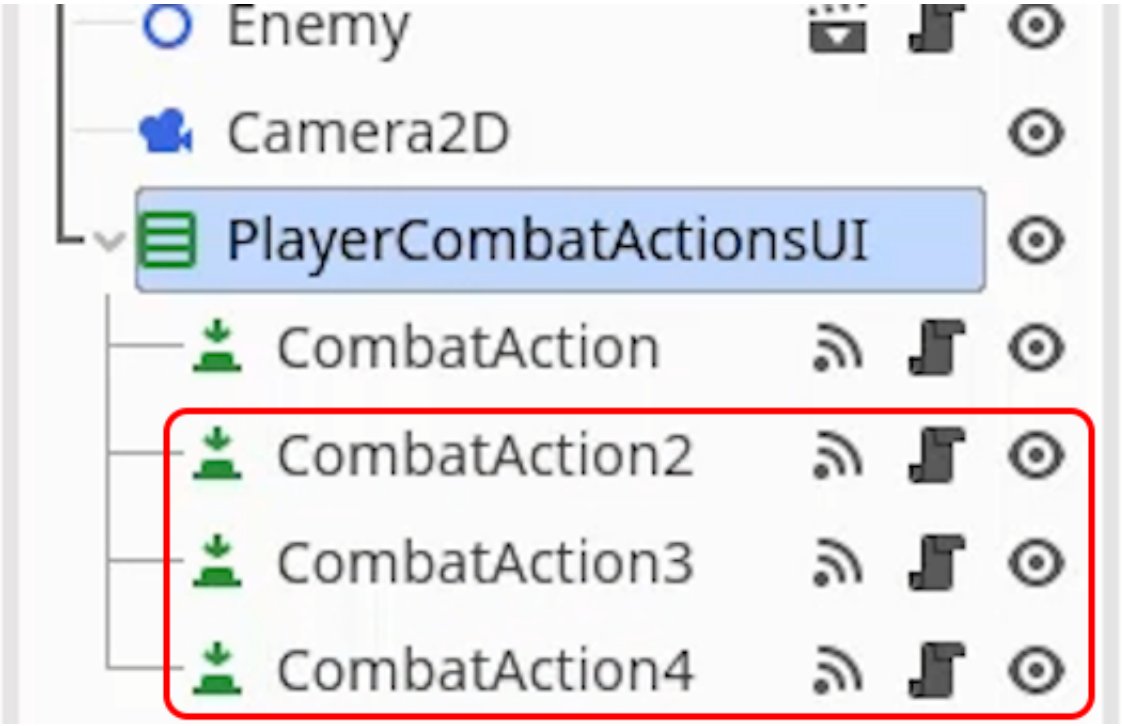
This is a relatively long line, however, it can easily be broken down into multiple steps:

- Firstly, we access the *BattleScene* root node using the **get_node** method.
- Then, we get the **cur_char** variable from the *TurnManager* script attached to the *BattleScene* node. Because the player's buttons are visible, we know this is our *Player* character instance.
- Finally, we call the **cast_combat_action** function on the current character, with a parameter of *combat_action.*

We can now **rename** the *Button* node to **CombatAction** so that it can be identified easily.

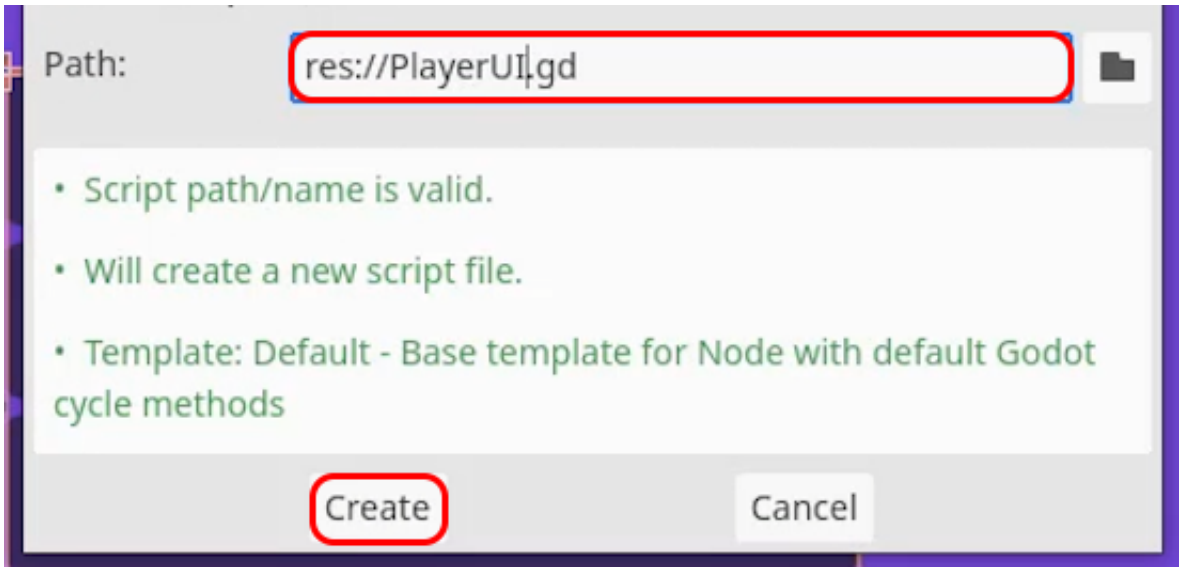


From here, **duplicate** (**CTRL+D**) the node a few times to create four buttons in total. These will automatically resize and position into a list, because of the *VBoxContainer* parent node.



PlayerUI Script

We now need to create a new script to handle connecting the *Player* character instance to the individual buttons. This script can be attached to the **PlayerCombatActionsUI** node and will be called **PlayerUI**.



In this script, we won't need the *_process* function, so this can be removed. After this, we can create a new variable called **buttons**, with a type of **Array**. This can be **exported** so that we can assign the buttons in the *Inspector* window.

```
@export var buttons : Array
```

This script will make use of our *character_begin_turn* and *character_end_turn* signals that we created previously. To do this, we will need to create two functions that can listen to these signals. The first will be called **_on_character_begin_turn**, with a parameter of **character**.

```
func _on_character_begin_turn(character):
    pass
```

We can then copy this function format for the **on_character_end_turn** function.

```
func _on_character_end_turn(character):
    pass
```

Next, we can connect these functions to the relevant *TurnManager* signals. This can be done by accessing the *TurnManager* script from our *BattleScene* root node and assigning the individual signals, inside the **_ready** function.

```
func _ready():
    get_node("/root/BattleScene/").character_begin_turn.connect(_on_character_begin_t
urn)
    get_node("/root/BattleScene/").character_end_turn.connect(_on_character_end_turn)
```

The next step is to toggle the *PlayerCombatActionUI's* visibility based on whether it is the player's turn or not. This can be done in the **_on_character_begin_turn** function.

```
func _on_character_begin_turn(character):
    visible = character.is_player
```

Here we assign the **visible** variable, supplied by Godot on the *VBoxContainer* node, to the **is_player** value of the currently active *Character*. This means that if it is our player's turn, the UI will be visible, and if it is not, the UI will be hidden. To make sure the player can't run multiple actions, we will also disable the visibility in the **_on_character_end_turn** function, no matter which character it is.

```
func _on_character_end_turn(character):
    visible = false
```

In the next lesson, we will look at populating the buttons and giving them the correct information.

In this lesson, we are going to continue creating our player's combat UI. Previously, we set up our buttons and the scripts required to make them work. The next stage is to implement the systems to pass the combat action information from the *Player* character to the buttons in the UI. We will begin by creating a new **_display_combat_actions** function in the **PlayerUI** script, with a parameter of **combat_actions**, which will be the list of combat actions to apply to the buttons.

```
func _display_combat_actions(combat_actions):
```

In this function, we can loop through each button in our *buttons* list using a *for* loop.

```
func _display_combat_actions(combat_actions):
    for i in len(buttons):
```

From here, we can grab the button node instance using the **get_node** function.

```
func _display_combat_actions(combat_actions):
    for i in len(buttons):
        var button = get_node(buttons[i])
```

We can now check to see whether we need to give this button a combat action. We can do this by checking whether **i** is larger than the *combat_actions* array's length. If the button is within the *combat_actions* length, we can make the button visible.

```
func _display_combat_actions(combat_actions):
    for i in len(buttons):
        ...
        if i < len(combat_actions):
            button.visible = true
```

Then, we can update the **text** property of the button based on the **display_name** value of the *CombatAction* at this index (*i*), along with assigning the **combat_action** property of the button.

```
func _display_combat_actions(combat_actions):
    for i in len(buttons):
        ...
        if i < len(combat_actions):
            ...
            button.text = combat_actions[i].display_name
            button.combat_action = combat_actions[i]
```

If the button is outside the bounds of the *combat_actions* array, we want to hide it.

```
func _display_combat_actions(combat_actions):
    for i in len(buttons):
```

```
        ...
        if i < len(combat_actions):
            ...
        else:
            button.visible = false
```

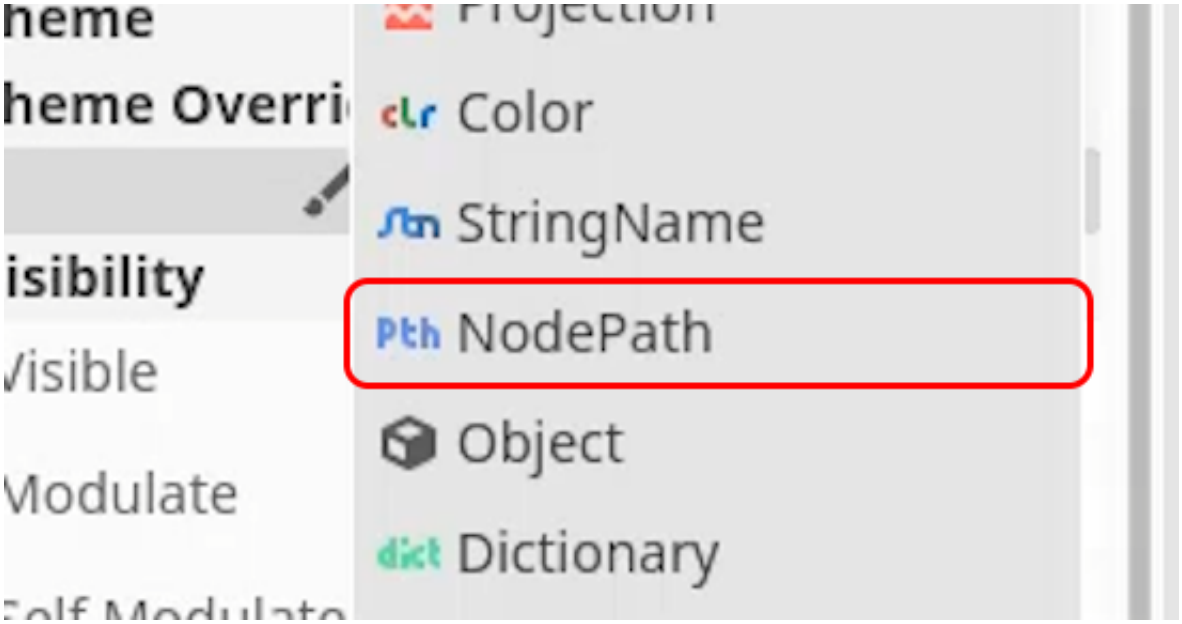
We want to call the **_display_combat_actions** function from the **_on_character_begin_turn**. However, we only want to do this when it is the player's turn, as we don't want the player to be able to choose the enemy's actions.

```
func _on_character_begin_turn(character):
    ...
    if character.is_player:
        _display_combat_actions(character.combat_actions)
```

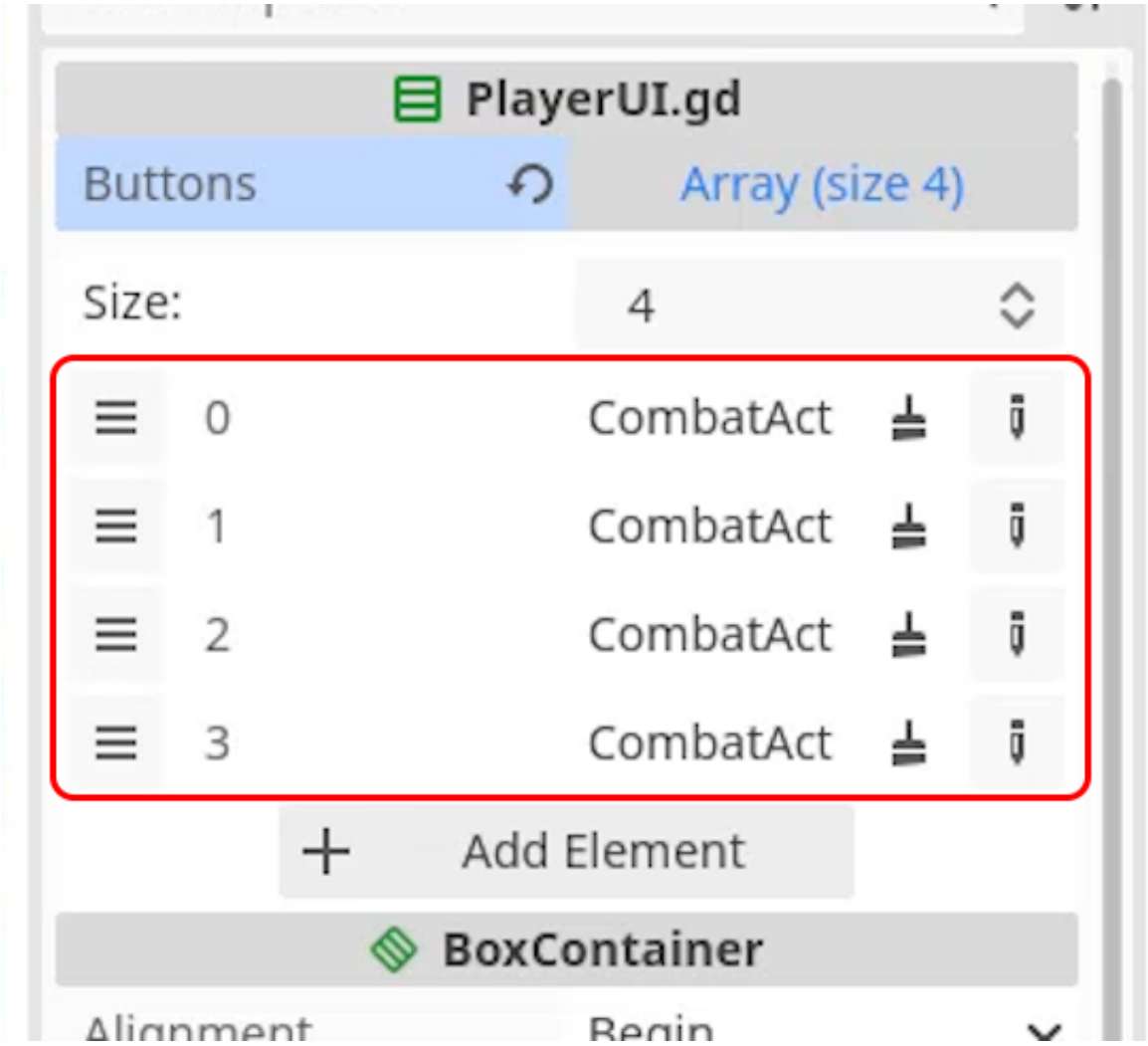
Finally, we also need to fix a small error in the previous lesson. In the **_on_pressed** function of the **CombatActionButton** script, we use a variable called *cur_character*. This should be **cur_char** to allow our buttons to work properly, well done if you caught this!

```
func _on_pressed():
    get_node("/root/BattleScene").cur_char.cast_combat_action(combat_action)
```

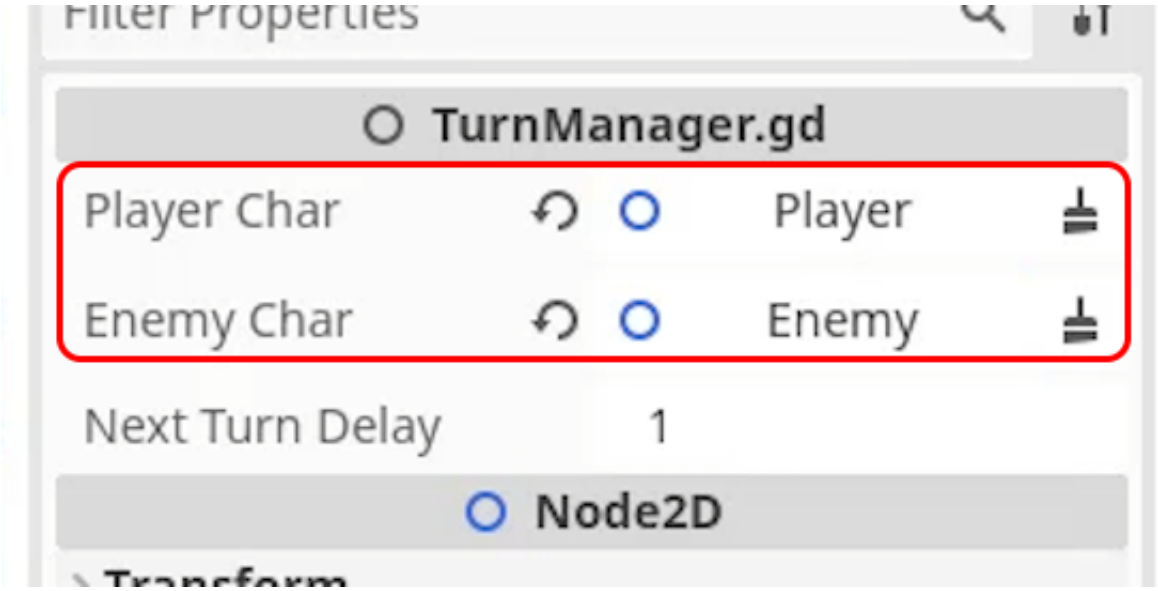
We now need to fill the **buttons** property of the **PlayerCombatActionsUI** in the *Inspector* window. For this *Array*, the element type (selected using the pencil icon) will be a **Node Path**.



We can drag each *CombatAction* button node into this *Array*.

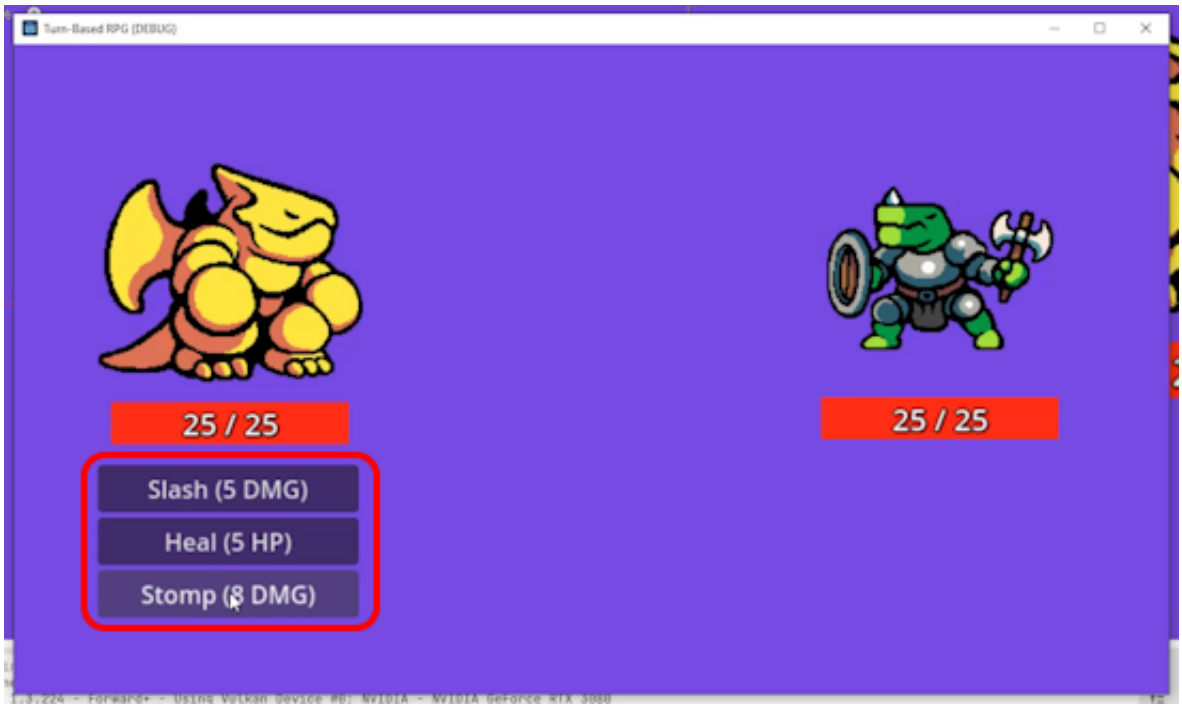


We also need to assign the *Player Char* and *Enemy Char* properties of the **BattleScene** node, inside the *Inspector* window.



We can now press the **play** button in the editor to try the game. With everything set up, you will see

the buttons appear as expected.



You can also try clicking the buttons. However, nothing will happen as we haven't yet set up the ability to cast the combat actions, so we will cover this in the next lesson.

The final step to finishing the player's UI is to set up the systems to handle our *Characters* casting combat actions. In this lesson, we will expand the **cast_combat_action** function, which is called both by our enemy AI (that we will create in the next lesson), and the player's onscreen buttons. The first step is to find out whether the **action** property is a combat action that causes damage or a combat action that heals the character. This can be done by checking if the **damage** property is more than zero.

```
func cast_combat_action(action):
    if action.damage > 0:
```

If the *damage* property is more than zero, this means the *action* property is aimed at causing damage to our opponent. Luckily, we have already set up the **take_damage** function on our *Character* script, so we can simply call this function on the **opponent** variable.

```
func cast_combat_action(action):
    if action.damage > 0:
        opponent.take_damage(action.damage)
```

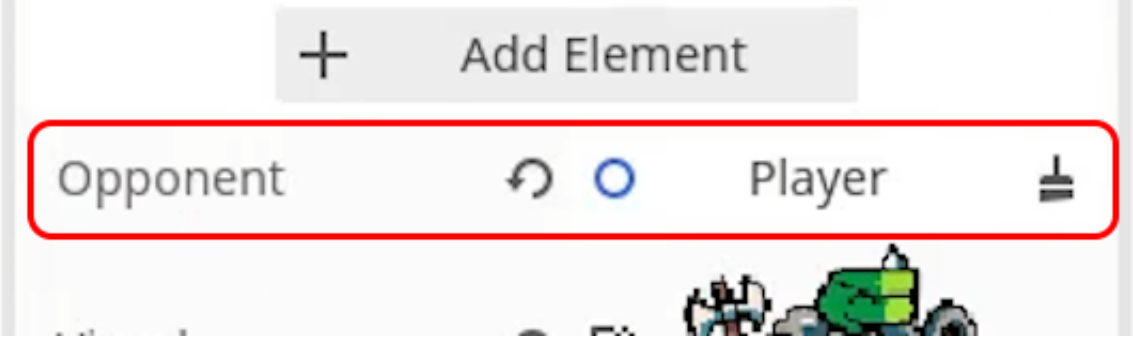
If the *damage* property is zero, but the **heal** property is more than zero, this means the *action* property is aimed at healing the current character. Because we want to heal the current character instance, we can simply call the **heal** function.

```
func cast_combat_action(action):
    if action.damage > 0:
        ...
    else if action.heal > 0:
        heal(action.heal)
```

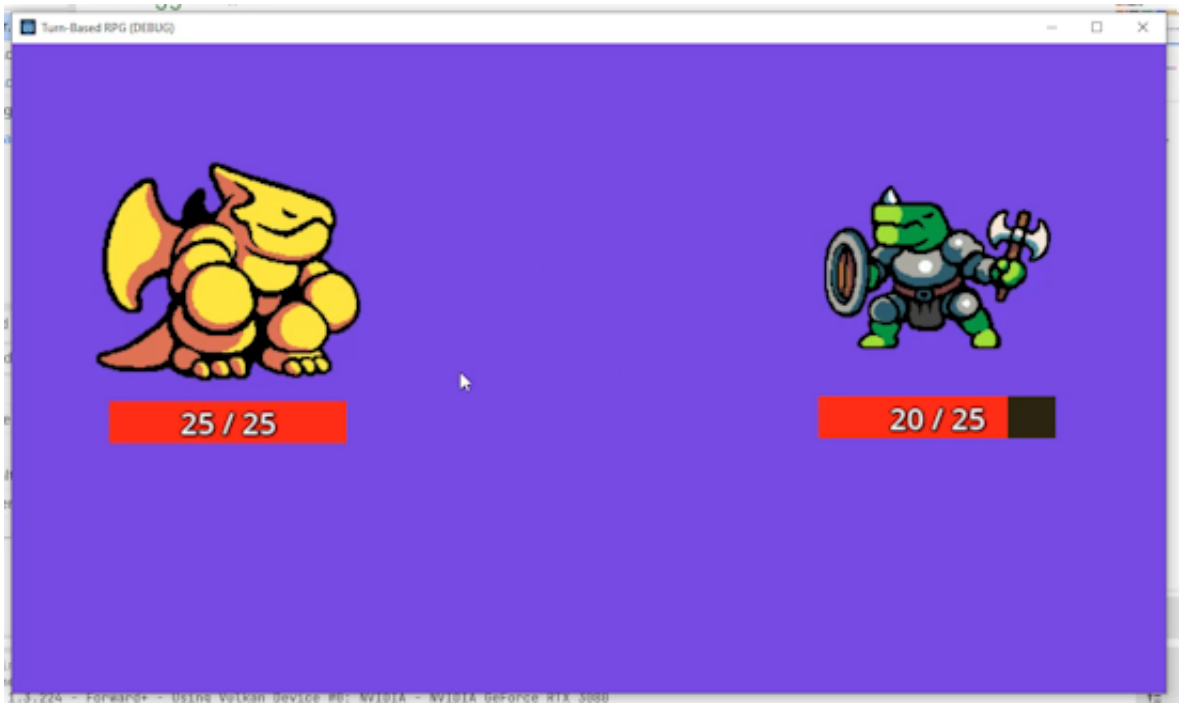
With the combat action processed, we then want to end the character's turn. To do this we want to call the **end_current_turn** function on the **BattleScene** root node.

```
func cast_combat_action(action):
    ...
    get_node("/root/BattleScene").end_current_turn()
```

Make sure to assign the **Opponent** variable for both the *Player* and the *Enemy* characters, if you haven't previously.



We should now be able to press **play** and test the combat actions. If you try pressing the *Slash* or *Stomp* buttons, you will see the damage is applied to the enemy and the UI disappears as the turn changes, just as expected.



Currently, our *Enemy* character can't fight back, so we will be setting up an enemy AI in the next lesson.

For the final lesson in this course, we will be focussing on creating a small enemy AI that allows our *Enemy* character to decide what combat action they want to take. This AI is going to go over its available combat actions and select one based on its current health.

AI Scripting

As our *Enemy* character already has the **Character** script attached to it, we will be adding our AI logic to this script. We will start by creating a new function called **_decide_combat_action**.

```
func _decide_combat_action ():
    pass
```

We will be calling this from the **_on_character_begin_turn** function. However, we want to make sure not to call this if the **character** parameter is the player, as we don't want to automatically decide the player's action.

```
func _on_character_begin_turn (character):
    if character == self and is_player == false:
        _decide_combat_action()
```

As the enemy AI is going to make decisions based on the health property, we want to convert the current health into a percentage of the maximum health. This will be represented as a value between 0 and 1.

```
func _decide_combat_action():
    var health_percent = float(cur_hp) / float(max_hp)
```

You will notice we cast these values to **floats**. This is because dividing an integer by an integer in GDScript will result in an integer number. For this use case, we want a decimal number (a float), so we have to convert the values we are dividing to floats. The next step will be to loop through each combat option with a **for** loop.

```
func _decide_combat_action():
    for i in combat_actions:
        var action = i as CombatAction
```

As you can see, we created a new variable called **action** here, which is simply our loop iterator (**i**) represented as a *CombatAction* type. We can use this *action* variable to check if it is a healing action and if it is, we want to check whether the *Enemy* character should choose to heal themselves. To do this, we are going to give the chance of healing a random chance.

```
func _decide_combat_action():
    for i in combat_actions:
        ...
        if action.heal > 0:
            if randf() > health_percent + 0.2:
                cast_combat_action(action)
```

```
                return
```

This code is best broken down into separate lines:

1. We begin by checking if the action is a healing *CombatAction*. We do this by checking if the *heal* property is more than zero on the *action*.
2. Next, we check if a random number between 0 and 1 is more than the current health percentage. We generate this random number using the **randf** function. If it is more than the current health percentage, we can call the action. To explain this further, if the health of the *Enemy* was 10/20, this means the *health_percent* variable would be 0.5. This means there is a 50% chance of *randf* being more than 0.5, and therefore a 50% chance of the *Enemy* character selecting heal. We will add **0.2** onto this, to lower the chances of healing by 20%.
3. Finally, we call to cast the action by calling the **cast_combat_action** function, and we use the **return** keyword to break out of the function to avoid more than one *CombatAction* being run.

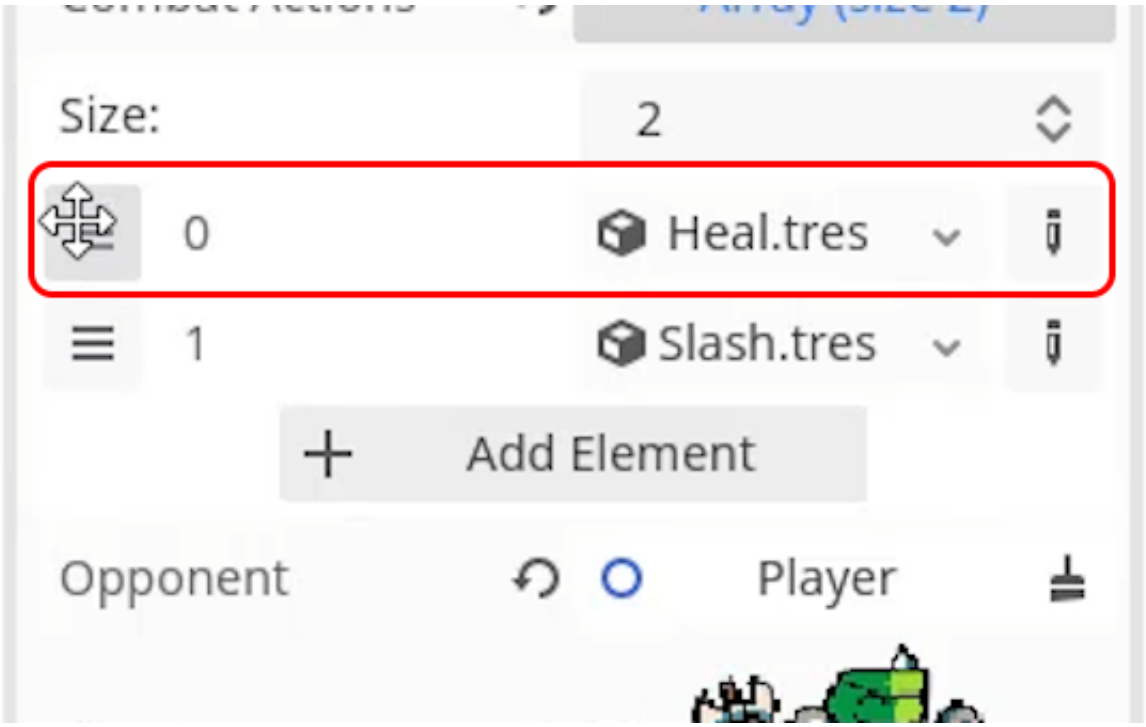
We can then add a call to **continue** if the action isn't cast. This will make the *for* loop start the next iteration.

```
func _decide_combat_action():
    for i in combat_actions:
        ...
        if heal > 0:
            if randf() > health_percent + 0.2:
                ...
            else:
                continue
```

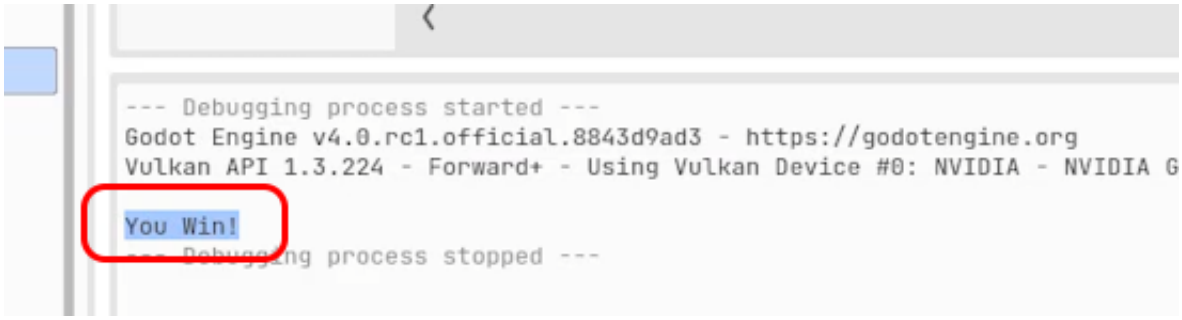
If the *action's* health property is zero, we will simply cast this *action* using the **cast_combat_action** function, and then **return** to avoid more than one *CombatAction* being run.

```
func _decide_combat_action():
    for i in combat_actions:
        ...
        if action.heal > 0:
            ...
        else:
            cast_combat_action(action)
            return
```

This is relatively simple because if there was no healing action, it would always just use the first attack. However, if you do want to expand this AI on your own, feel free to add as much functionality as you like. You should now be able to press **play** in the Godot editor to try fighting your enemy AI. If you find the enemy is only attacking and not healing, make sure the heal *CombatAction* is at the top of the *Enemy* character's *combat_actions* array.



You will also notice that if you look inside the Godot console after the game ends, you will see a win or loss message.



This AI is quite easy to defeat, so feel free to try adding different attacks, changing the chances of them healing, or adding your own functionality to the AI to make the game more challenging! Unfortunately, this topic could take up an entire course on its own, so we will have to leave it there for this lesson.

Congratulations! You have just completed our course on creating a turn-based RPG in Godot. The systems that we have created here can act as a pretty good foundation for creating the game you want to create. You can take the lessons learned in this project to expand into new projects or to continue to create more features for this game. For example, you could expand the combat action system to add spells that require a mana stat or different types of attacks. The options are limitless. With this project, we looked at a range of topics, so we will quickly cover them here.

Characters

We began by working on our characters. In this game, our *Character* script and scene handle both our player and our enemy. Our characters are shown on the screen as a sprite and a health bar with a text representation of the health. The script allows our characters to take and deal damage, along with healing, and update the health bar to represent these actions.

Turn Manager

The second stage was to create the *TurnManager* script. This handles keeping track of whether it is the player's or the enemy's turn and communicating this to the characters. The turn manager also handles game-over functions.

Combat Actions

The next step was to create the systems to handle *CombatActions*. When it is a specific character's turn they get to cast a specific combat action. This allows them to either damage the opponent or heal themselves. To create our *CombatActions* we used Godot's *Resource* system, allowing us to create custom assets that can then be supplied to the player and enemy characters. We also created a UI for our *Player* character, allowing them to select the combat action when it is their turn.

Enemy AI

The final element of this game was to create an enemy AI. This allows the *Enemy* character to choose which combat action they want to use. Depending on the enemy's health, they are more or less likely to select the heal action over the attack action.

Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the included course files.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

Character.gd

Found in the Project root folder

This Godot script controls a Character object in a turn-based combat game. Each character subjects to actions like taking damage, healing, and updating the health bar. If it's not the player, the character decides on an action automatically. When an action is taken, either by the player or the AI, the action affects either the character itself or its opponent, and the turn ends.

```
extends Node2D
class_name Character

@export var is_player : bool
@export var cur_hp : int = 25
@export var max_hp : int = 25

@export var combat_actions : Array
@export var opponent : Node

@onready var health_bar : ProgressBar = get_node("HealthBar")
@onready var health_text : Label = get_node("HealthBar/HealthText")

@export var visual : Texture2D
@export var flip_visual : bool

# Called when the node enters the scene tree for the first time.
func _ready():
  $Sprite.texture = visual
  $Sprite.flip_h = flip_visual

  get_node("/root/BattleScene").character_begin_turn.connect(_on_character_begin_turn
)
  health_bar.max_value = max_hp

func take_damage (damage):
  cur_hp -= damage
  _update_health_bar()

  if cur_hp <= 0:
    get_node("/root/BattleScene").character_died(self)
    queue_free()

func heal (amount):
  cur_hp += amount

  if cur_hp > max_hp:
    cur_hp = max_hp
```

```
  _update_health_bar()

func _update_health_bar ():
  health_bar.value = cur_hp
  health_text.text = str(cur_hp, " / ", max_hp)

# called when a character's turn begins
func _on_character_begin_turn (character):
  if character == self and is_player == false:
    _decide_combat_action()

# called when the player selects a CA, or the enemy decides on one
func cast_combat_action (action):
  if action.damage > 0:
    opponent.take_damage(action.damage)
  elif action.heal > 0:
    heal(action.heal)

  get_node("/root/BattleScene").end_current_turn()

# enemy ai
func _decide_combat_action ():
  var health_percent = float(cur_hp) / float(max_hp)

  for i in combat_actions:
    var action = i as CombatAction

    if action.heal > 0:
      if randf() > health_percent + 0.2:
        cast_combat_action(action)
        return
      else:
        continue
    else:
      cast_combat_action(action)
      return
```

CombatActionButton.gd

Found in the Project root folder

This Godot script extends a Button node and associates it with a specific ‘CombatAction’. When the button is pressed during gameplay, it triggers the current character in the ‘BattleScene’ to execute the stored combat action.

```
extends Button

var combat_action : CombatAction

# called when we press the button
# cast the stored combat action
func _on_pressed():
```



```
  get_node("/root/BattleScene").cur_char.cast_combat_action(combat_action)
```

PlayerUI.gd

Found in the Project root folder

This Godot script manages a vertical box container in a turn-based combat game. When a character's turn begins, the script makes combat action buttons visible for the player and hides them once the turn ends. It dynamically populates the buttons with the current character's available combat actions.

```
extends VBoxContainer

@export var buttons : Array

# Called when the node enters the scene tree for the first time.
func _ready():
  get_node("/root/BattleScene").character_begin_turn.connect(_on_character_begin_turn
)
  get_node("/root/BattleScene").character_end_turn.connect(_on_character_end_turn)

# called when a character begins their turn
func _on_character_begin_turn (character):
  visible = character.is_player

  if character.is_player:
    _display_combat_actions(character.combat_actions)

# called when a character ends their turn
func _on_character_end_turn (character):
  visible = false

# display the buttons for the player's combat actions
func _display_combat_actions (combat_actions):
  for i in len(buttons):
    var button = get_node(buttons[i])

    if i < len(combat_actions):
      button.visible = true
      button.text = combat_actions[i].display_name
      button.combat_action = combat_actions[i]
    else:
      button.visible = false
```

TurnManager.gd

Found in the Project root folder

This Godot script manages the turn-based combat mechanism between a player character and an enemy character. It signals the beginning and end of each turn. When the game ends, caused by either character's death, it will print "Game Over!" if the player character dies or "You Win!" if the

enemy character dies.

```
extends Node

@export var player_char : Node
@export var enemy_char : Node
var cur_char : Character

@export var next_turn_delay : float = 1.0

var game_over : bool = false

signal character_begin_turn(character)
signal character_end_turn(character)

func _ready():
  await get_tree().create_timer(0.5).timeout
  begin_next_turn()

func begin_next_turn():
  if cur_char == player_char:
    cur_char = enemy_char
  elif cur_char == enemy_char:
    cur_char = player_char
  else:
    cur_char = player_char

  emit_signal("character_begin_turn", cur_char)

# called when a character has cast their combat action
func end_current_turn():
  emit_signal("character_end_turn", cur_char)

  await get_tree().create_timer(next_turn_delay).timeout

  if game_over == false:
    begin_next_turn()

# called when a character has died
# sends over the now dead character
func character_died (character):
  game_over = true

  if character.is_player == true:
    print("Game Over!")
  else:
    print("You Win!")
```

CombatAction.gd

Found in the Project root folder

This Godot script defines a CombatAction resource class, which can be used to describe an action in a combat scenario in the game. Each instance of this class specifies an action's display name, the damage it inflicts, and the healing it performs.

```
extends Resource
class_name CombatAction

@export var display_name = "Action (x DMG)"
@export var damage : int = 0
@export var heal : int = 0
```

