CODE NINJAS®

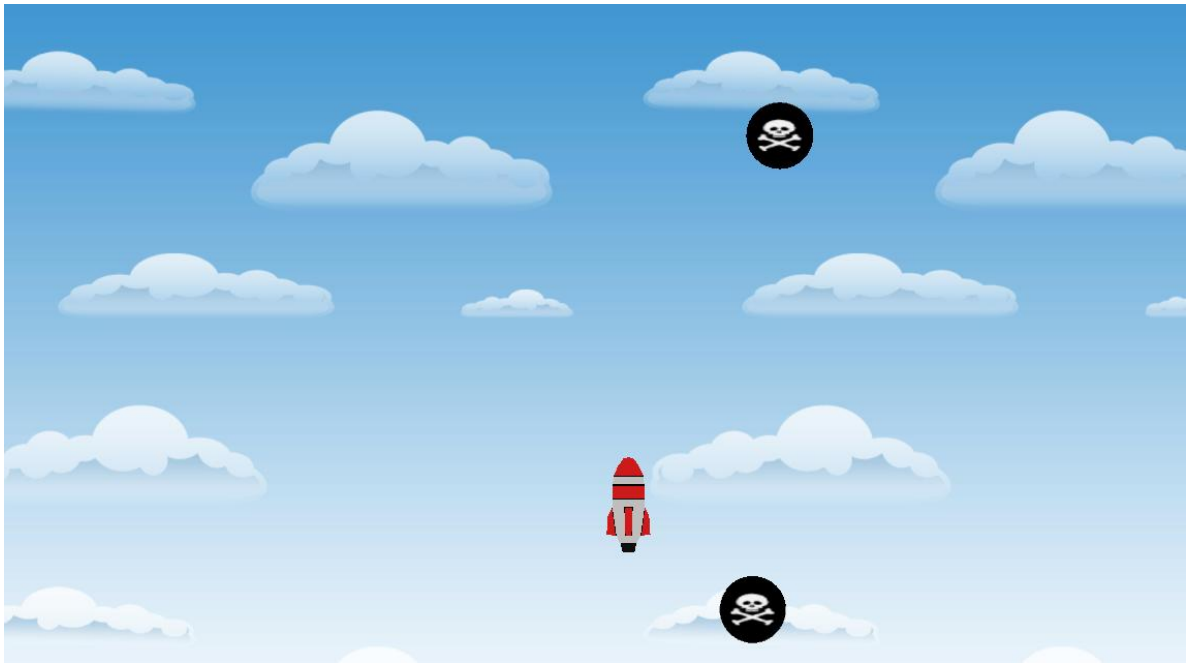
Bronze Belt Ninja Guide

Activity 09: Dropping Bombs Part 3

Activity 09: Dropping Bombs Part 3

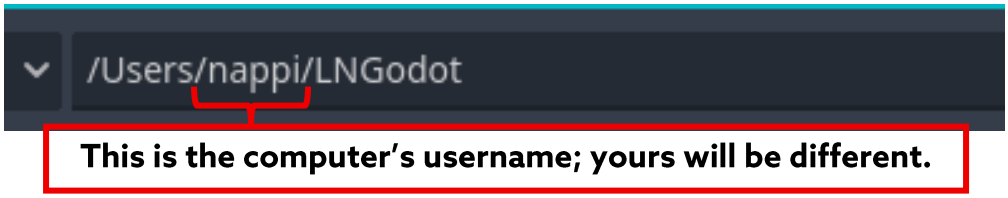
In this activity, you will continue refining the Dropping Bombs project by making the bomb spawning more robust with Godot's built-in features, cleaning up unused assets, and using the viewport to control the bounds that the bombs can spawn in.

By the end of this activity, you will have learned how to save resources in your code by converting nodes into packed scenes, how to add signals to make different parts of your code interact, and how to use **groups** to make something happen to many nodes at once.



1 All projects will be stored in a path like:
/Users/[MyComputerUsername]/[MyInitials]Godot

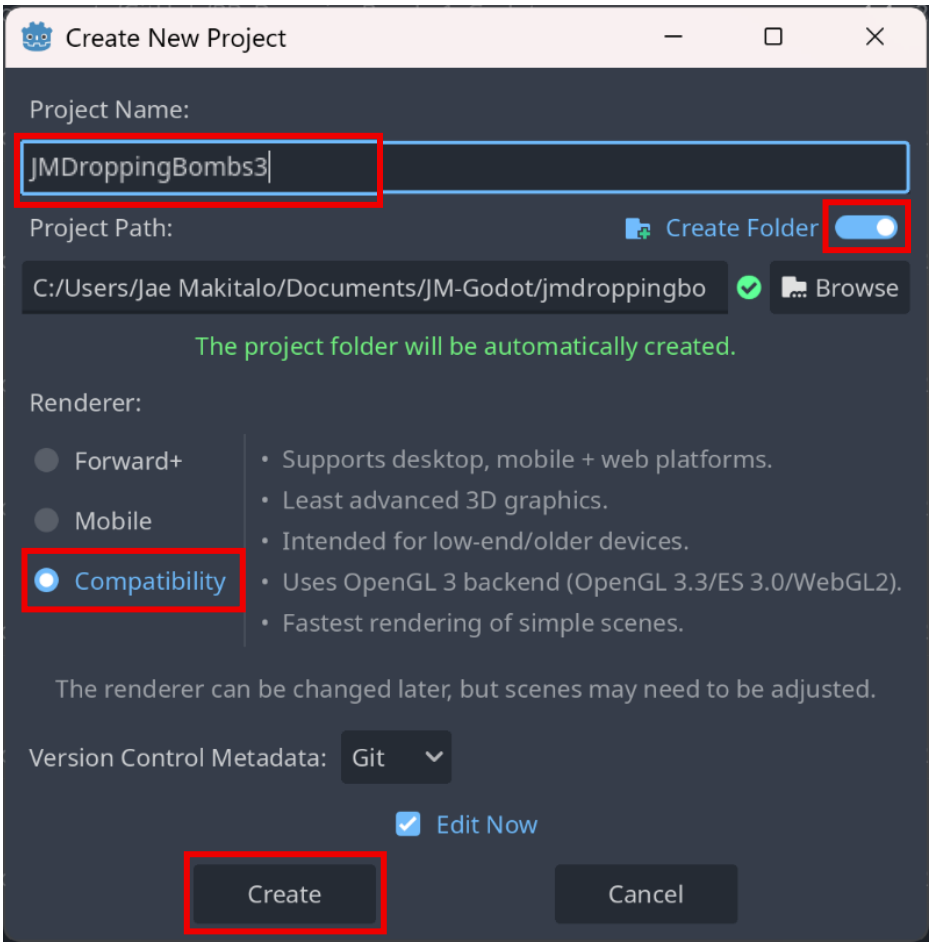
Don't worry if your path looks slightly different from the image shown! All computers have their own username.



2 After opening Godot, in the top left corner select **+ Create.**

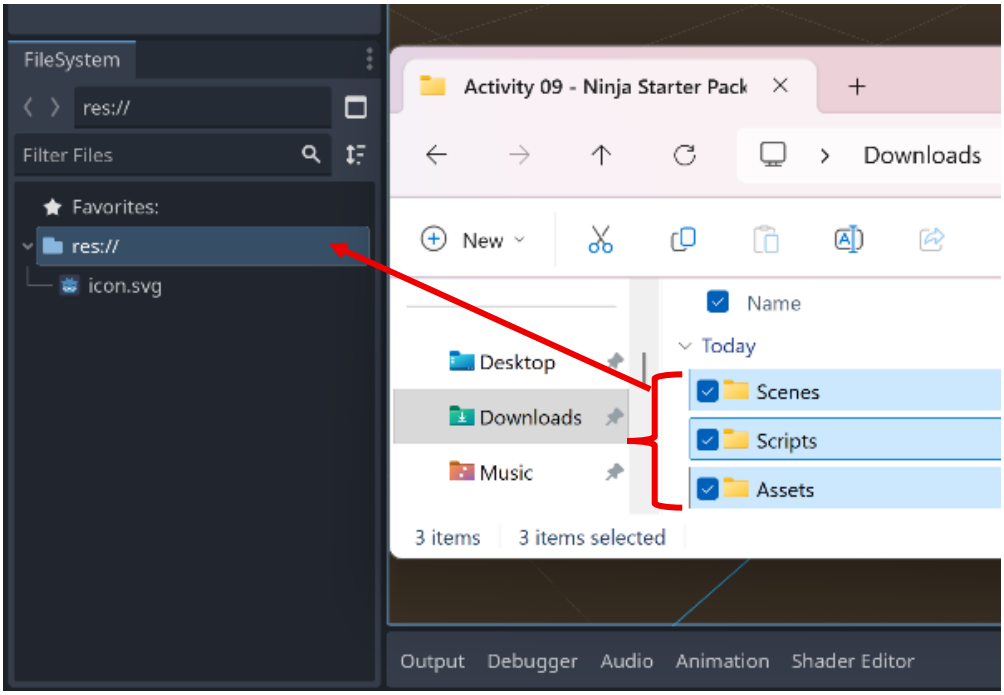
A **Create New Project** window will pop up. Name the project **[MyInitials]DroppingBombs3**.

Check that **Create Folder** is turned on, and that the **Compatibility** mode for the renderer is being used. Then click **Create**.

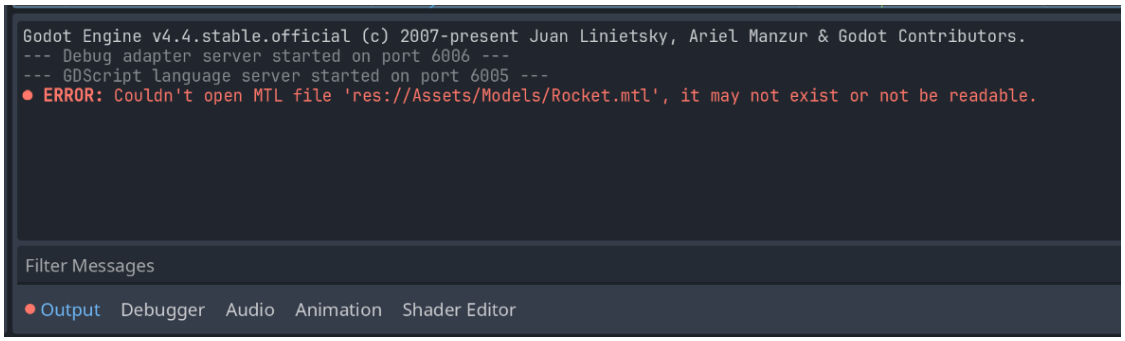


3 Don't create the **main scene** and **Main root** node just yet!

Extract **BB Activity 09 – Ninja Starter Pack.zip** and select all folders inside. Drag them into the **res://** folder in **FileSystem**.



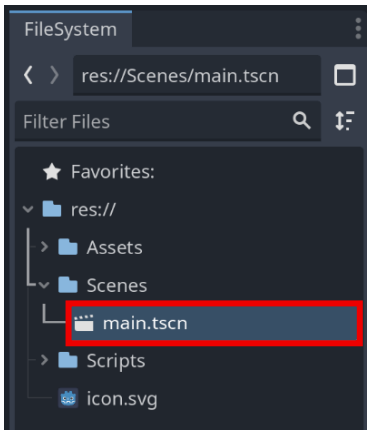
An error may appear in Output. This can be ignored for now and will resolve itself later in the project.



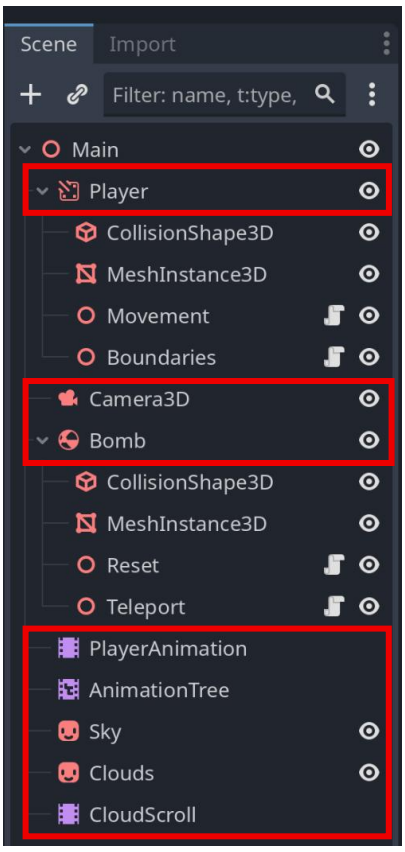
Reminder:

Double-click on the folder icon from **Downloads**. Then **right-click** on the zip file and select **Extract All.** Press **CTRL + J** on the keyboard to reopen the **File Explorer**.

4 In **FileSystem**, navigate to **main.tscn** and double click to open it.



Notice the main scene already has the **Player**, **Camera3D**, **Bomb**, animations, and sprite nodes that were created in Dropping Bombs Part 2.

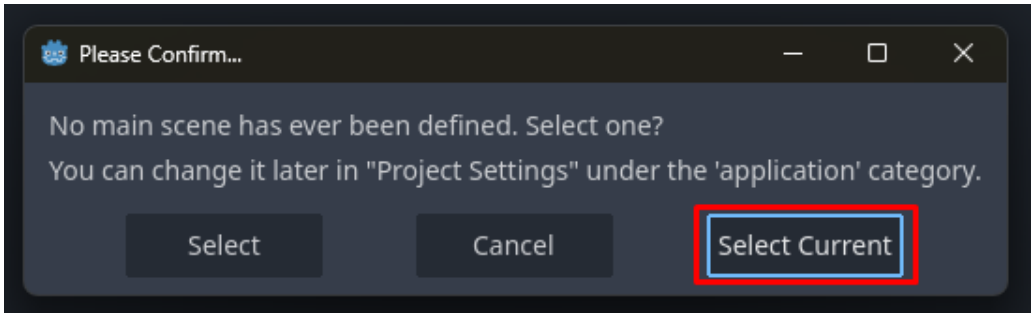


Reminder:

Click the arrows next to the folders to open or close them.

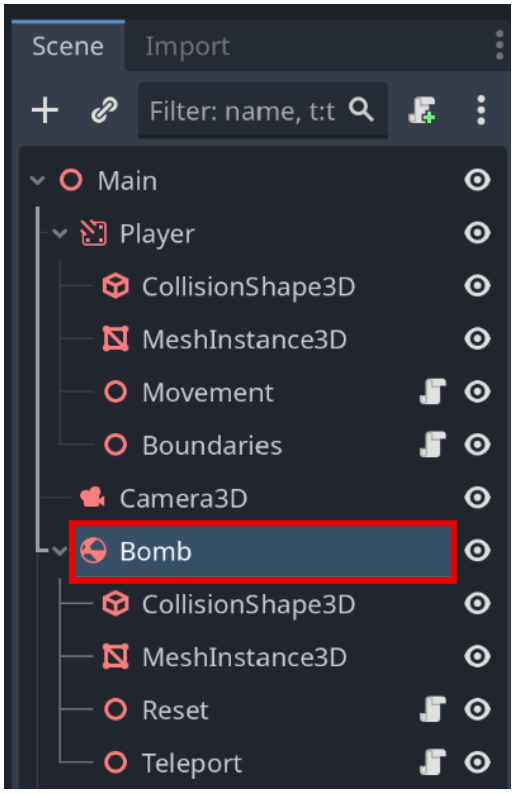
5 Playtest the project.

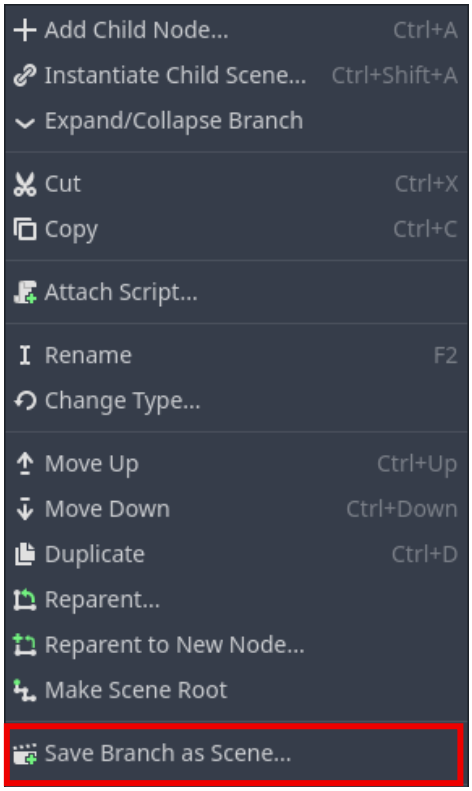
The main scene needs to be defined. Click **Select Current** to define the main scene.



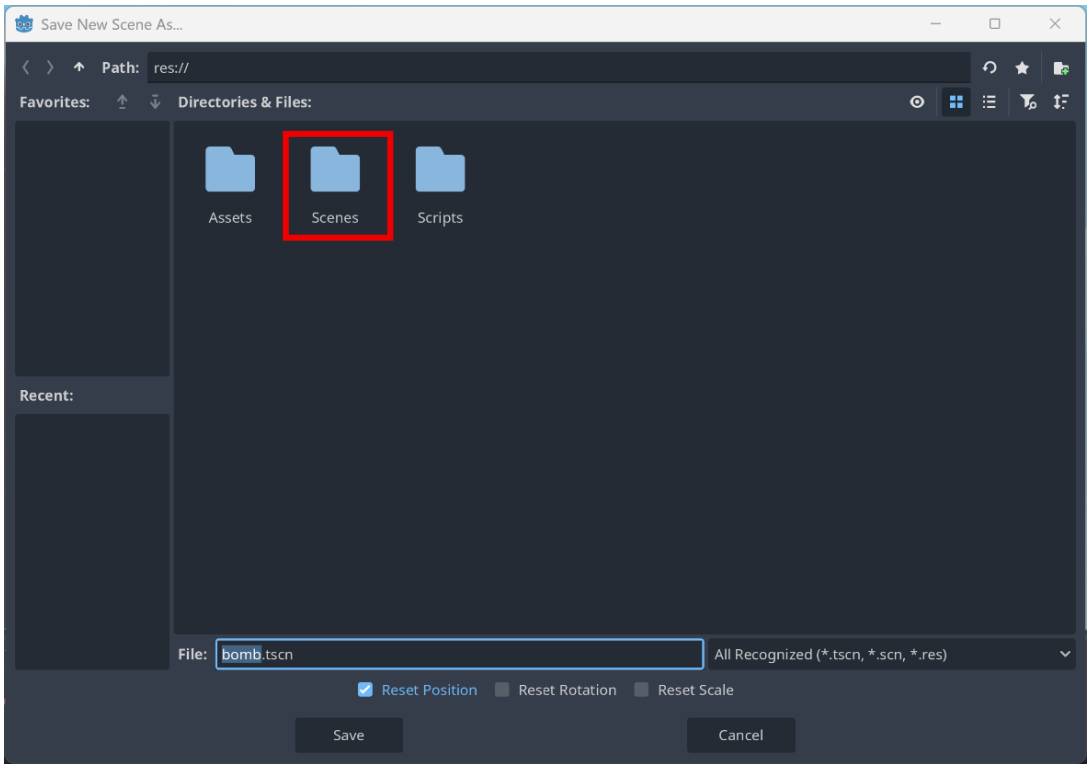
6 Create a scene from the Bomb node.

In **Inspector,** right click on the **Bomb** node and select **Save Branch as Scene** from the dropdown menu.

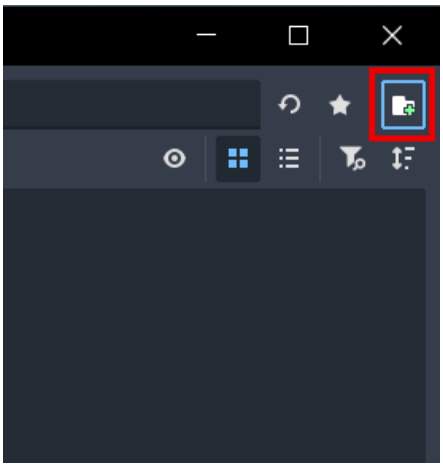


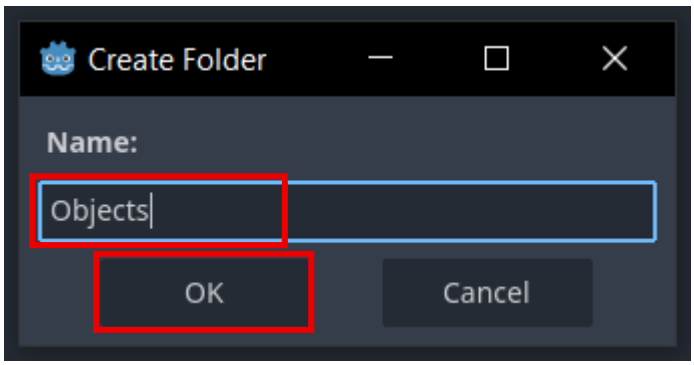


7 In the **Save New Scene As** window, double click the **Scenes** folder to open it.

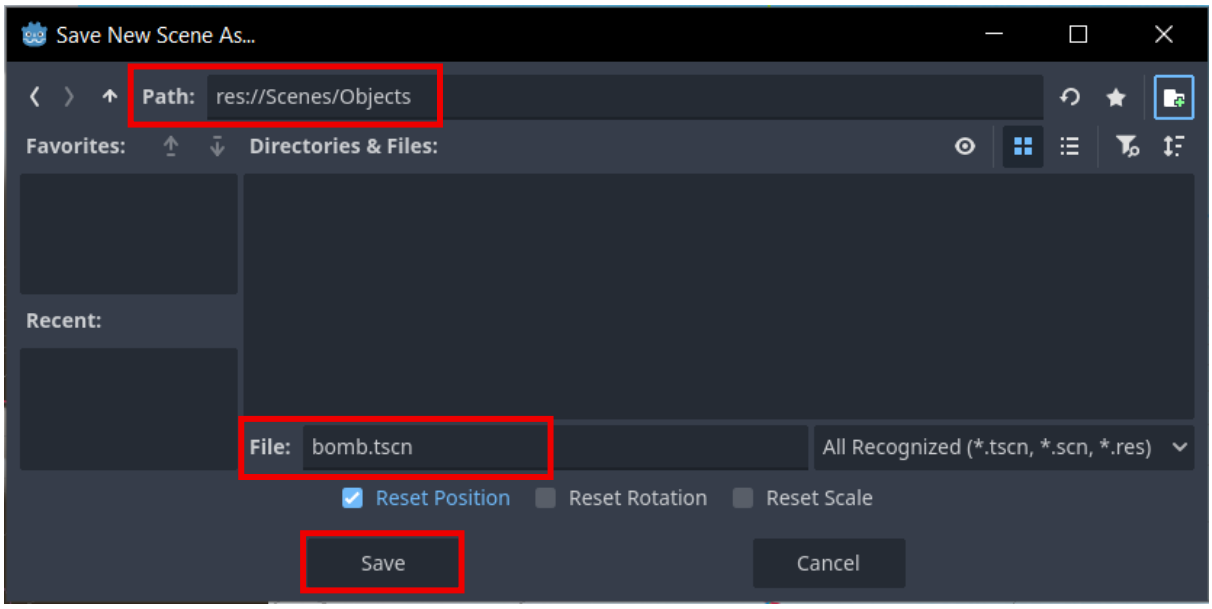


8 In the top right corner, click the **Create a New Folder** icon. Name the new folder **Objects** and click **OK**.

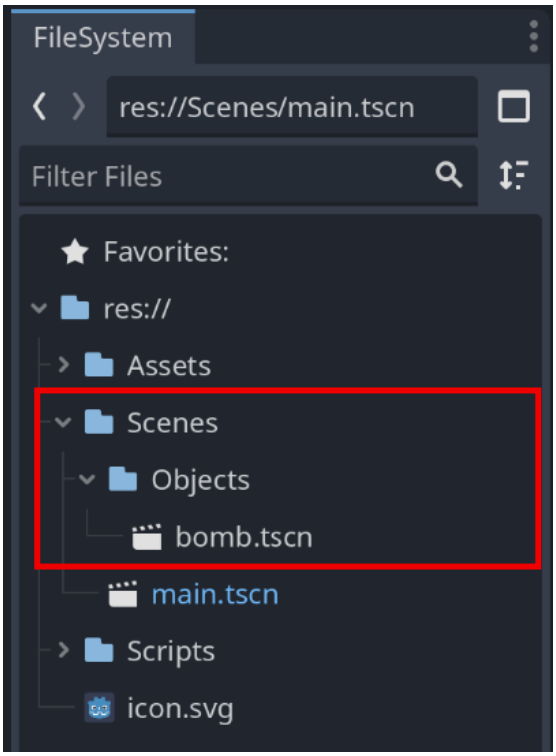




9 Check that the **Path** at the top of the window shows **res://Scenes/Objects** and that the **File** name is **bomb.tscn**. Click **Save**.

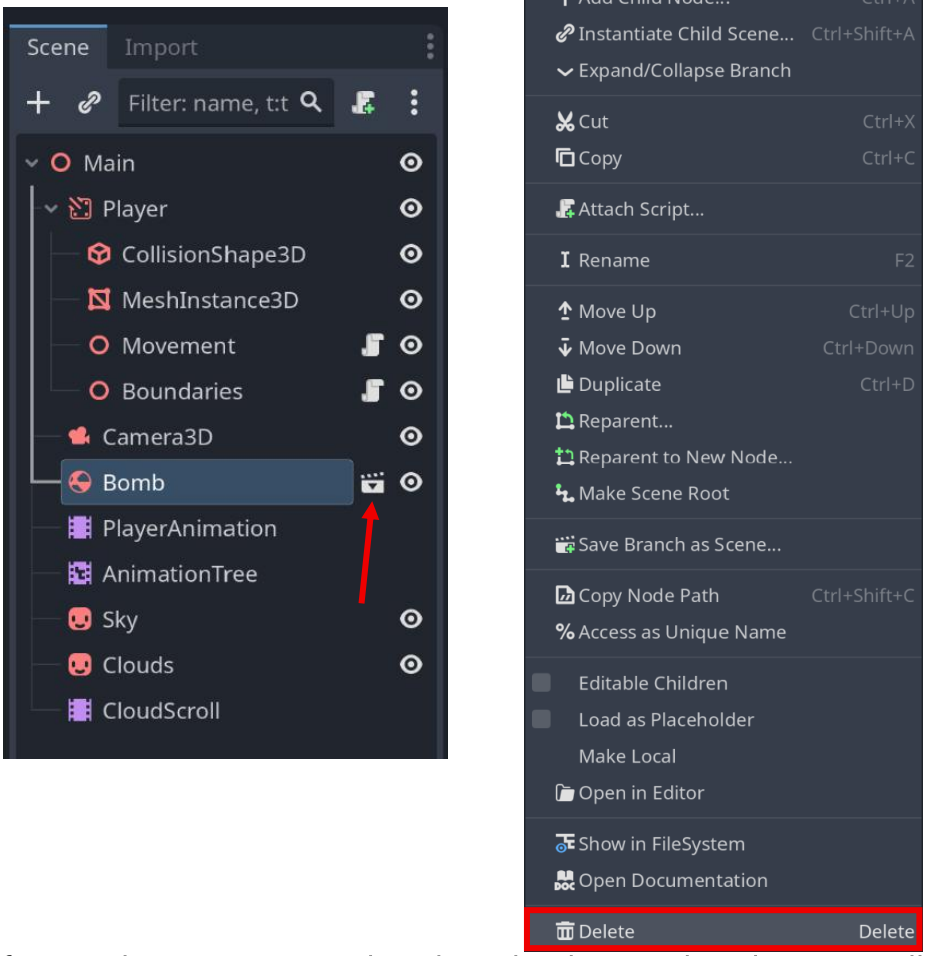


10 In **FileSystem** under **Scenes > Objects**, the **bomb.tscn** now appears. It has been saved as a scene.



11 In **Scene,** notice the **Scene** icon next to the Bomb node.

Since the bomb node has been saved as a **packed scene** in the new **Objects** folder, the Bomb node in the **main scene** can be removed. **Right-click** on **Bomb** and select **Delete**.



Instead of using the main scene's bomb node, the new bomb scene will be copied and instantiated into the game.

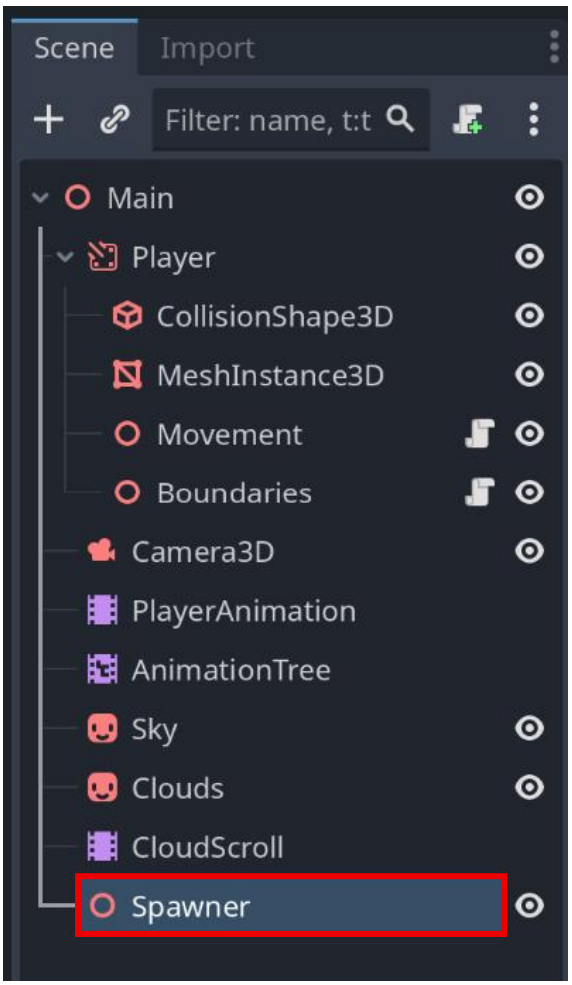
Pause for **Sensei Stop #1**!

Check in with a Code sensei before moving on. Make sure the **Ninja Starter Pack** has been imported correctly, and the bomb node has been saved as a scene and removed from the **Scene** hierarchy.

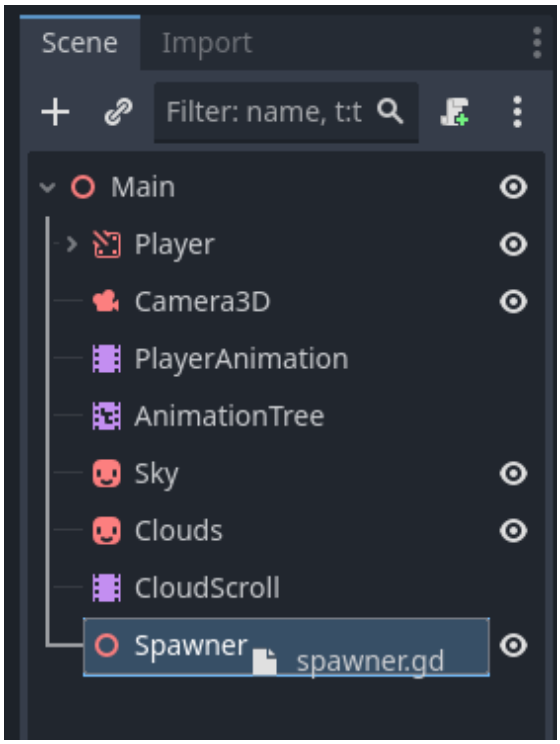
Reminder: Save your work!

12 In **Scene**, add a **Node3D** as a child to **Main** and rename it **Spawner.**

This will be used to repeatedly spawn new bombs during the game.



13 In **Scene**, attach the **spawner.gd** script to **Spawner** by dragging and dropping the script onto the node.



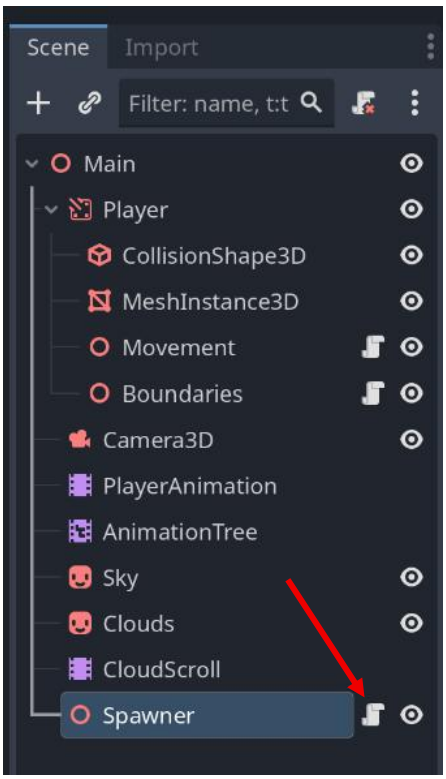
Reminder:

Another way to do this is in the **Inspector**. In the drop-down next to the **Script** property, select **Quick Load**, then click on the **spawner.gd** script.

14

Open the **spawner.gd** script.

Notice the **TODO** comments already in the script. What code might be added to program the spawner node?




```
1   extends Node
2
3   # --------------------
4   # TODO 1
5   # Create the variables
6   # --------------------
7
8
9   # --------------------
10  # TODO 2
11  # Get the viewport size
12  # --------------------
13
14
15  # --------------------
16  # TODO 3
17  # Instantiate a bomb object and give it a spawn position
18  # --------------------
19
20
21  # --------------------
22  # TODO 4
23  # Remove bomb objects when they've left the viewport
24  # --------------------
25
```

15 Add variables needed to store the different pieces of information this script will use.

Under **TODO 1**, use `@export` to declare a `bomb_scene` variable of type `PackedScene`. This variable will store an instance of the bomb scene.

```
3  # --------------------
4  # TODO 1
5  # Create the variables
6  # --------------------
7  @export var bomb_scene: PackedScene
8
```

Reminder:

@export is only used when a variable needs to be accessed or updated in the Inspector.

16 Declare 2 class-level variables to be used in this script.

- `spawn_x`, of type **`float`** – this will be used to determine the horizontal position of a spawning bomb.
- `spawn_y`, of type **`float`** – this will be used to determine the vertical position of a spawning bomb.

```
3  # --------------------
4  # TODO 1
5  # Create the variables
6  # --------------------
7  @export var bomb_scene: PackedScene
8  var spawn_x: float
9  var spawn_y: float
10
```

Pro Tip:

Class-level describes a variable that belongs to the entire script rather than a single function. At the top of the script, the variables that are outside of any function are called "class-level" variables.

17 Under **TODO 2**, use the code completion tool to add the script's `_ready()` method.

Inside `_ready()`, assign the `spawn_x` and `spawn_y` variables.

Use `get_viewport()` to access the viewport. Then, use the `size` property to access the viewport's `x` and `y` properties. Divide these values by 100.

```
12  # --------------------
13  # TODO 2
14  # Get the viewport size
15  # --------------------
16  func _ready() -> void:
17      # spawn_x ???
18      # spawn_y ???
19
```

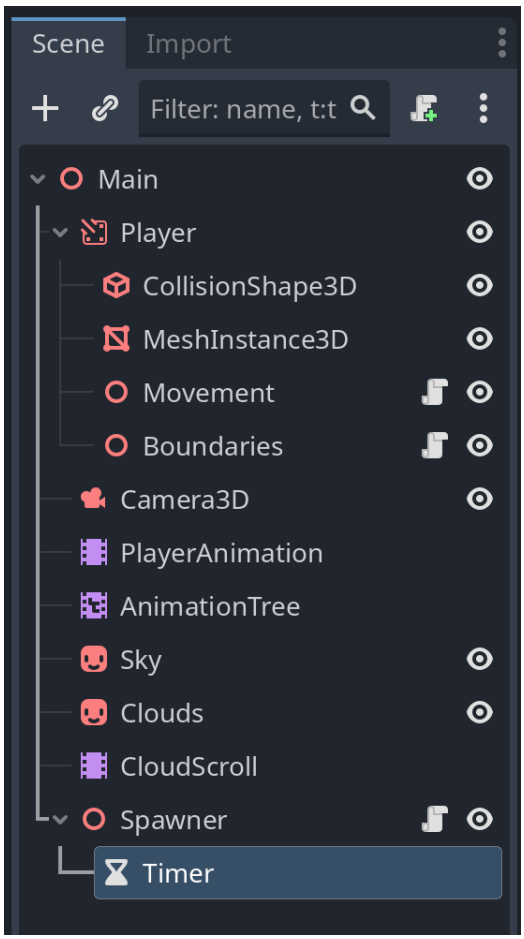
New Concept: Viewport

The **viewport** is a built-in feature of Godot. Think of it as the screen that Godot is drawing the game on, so the player can see it. The viewport will automatically associate itself with the highest Camera node in the **Scene** hierarchy.

18 Check the code! Update the script as needed.

```
12  # --------------------
13  # TODO 2
14  # Get the viewport size
15  # --------------------
16  func _ready() -> void:
17      spawn_x = get_viewport().size.x/100
18      spawn_y = get_viewport().size.y/100
19
```

19 In **Scene**, add a **Timer** node as a child to the **Spawner** node.

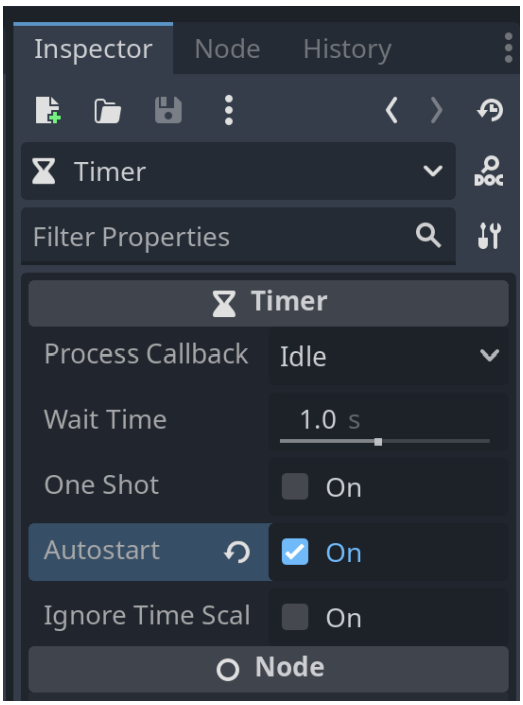


New Concept: Timer node

Remember how timers were used in Don't Touch the Cubes and Super Shapes? Well, those projects were really creating temporary Timer nodes! Example from Super Shapes:

```
await get_tree().create_timer(spawn_delay).timeout
```

20 In the **Inspector** for the **Timer** node, make sure the **Autostart** property is turned on.



21 In the **spawner.gd** script, declare the `_on_timer_timeout()` function under **TODO 3**. This will be used to make bombs each time the Timer node has waited for its entire wait time.

```
21  # --------------------
22  # TODO 3
23  # Instantiate a bomb object and give it a spawn position
24  # --------------------
25  func _on_timer_timeout() -> void:
26      
27
```

22 Inside the `_on_timer_timeout()` function definition, declare a `bomb` variable and set it to an instance of the `bomb_scene` PackedScene by using `.instantiate()`.

```
21  # --------------------
22  # TODO 3
23  # Instantiate a bomb object and give it a spawn position
24  # --------------------
25  func _on_timer_timeout() -> void:
26      var bomb = bomb_scene.instantiate()
27
```

23 Set the **position** property of the new **bomb** variable using a **Vector3**.

What parameters of a Vector3 can be used to set the new bomb's spawning position?

```
21  # --------------------
22  # TODO 3
23  # Instantiate a bomb object and give it a spawn position
24  # --------------------
25  func _on_timer_timeout() -> void:
26      var bomb = bomb_scene.instantiate()
27      bomb.position = Vector3()
28                  Vector3 Vector3(from: Vector3)
29  # ----------    Vector3 Vector3(from: Vector3i)
30  # TODO 4        Vector3 Vector3(x: float, y: float, z: float)
```

Reminder:

Hovering the mouse over certain terms will display more information about it.

24

Because this game is played in two dimensions, the bomb's position will be set using the **x** and **y** values.

`Vector3` takes **3** parameters. When adding them, place the values inside the parentheses of `Vector3()` and separate each one with a comma.

For example: `Vector3(parameter1, parameter2, parameter3)`

- **X**: Use `randf_range` to set the **x** value to a random value from the range between `-spawn_x` and `spawn_x`. This will make a new bomb appear at a random horizontal position.
- **Y**: Use `spawn_y` as the **y** value to make all bombs spawn at the same height.
- **Z**: Set the last parameter to `0`, so the bombs inherit the spawner's Z-value.

```
21 # --------------------
22 # TODO 3
23 # Instantiate a bomb object and give it a spawn position
24 # --------------------
25 func _on_timer_timeout() -> void:
26     var bomb = bomb_scene.instantiate()
27     bomb.position = Vector3(randf_range(-spawn_x, spawn_x), spawn_y, 0)
28
```

25

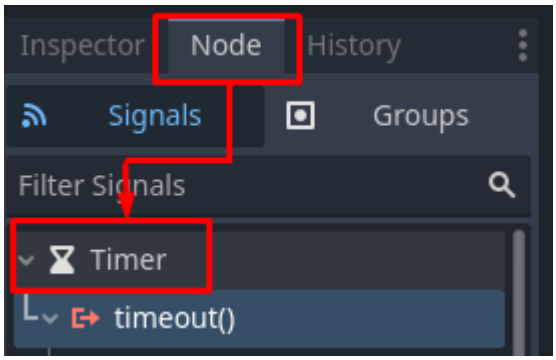
Add the new bombs as children to the **Spawner** node by using `add_child()`.

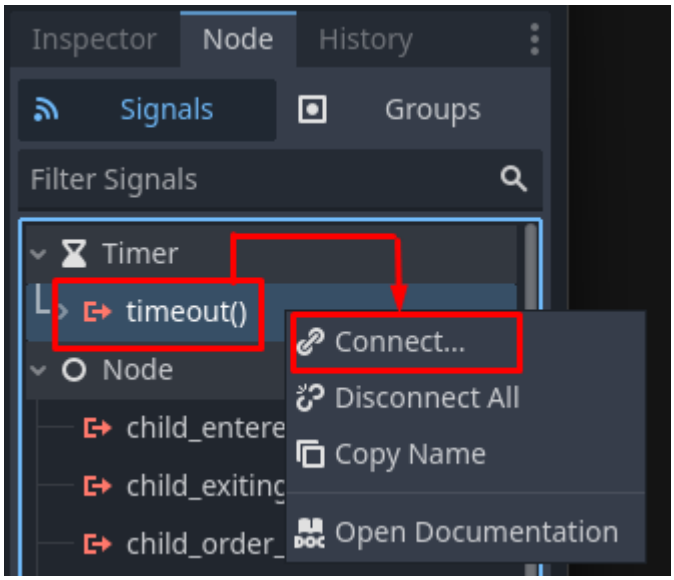
```
21 # --------------------
22 # TODO 3
23 # Instantiate a bomb object and give it a spawn position
24 # --------------------
25 func _on_timer_timeout() -> void:
26     var bomb = bomb_scene.instantiate()
27     bomb.position = Vector3(randf_range(-spawn_x, spawn_x), spawn_y, 0)
28     add_child(bomb)
29
```

26 To make a new bomb spawn when the timer runs out, `_on_timer_timeout()` needs to be called from the **Timer** node's `timeout()` signal.

In the **Inspector** for the **Timer** node, click on the **Node** tab at the top to open the **Signals** menu.

Right click on `timeout()` and choose **Connect...**

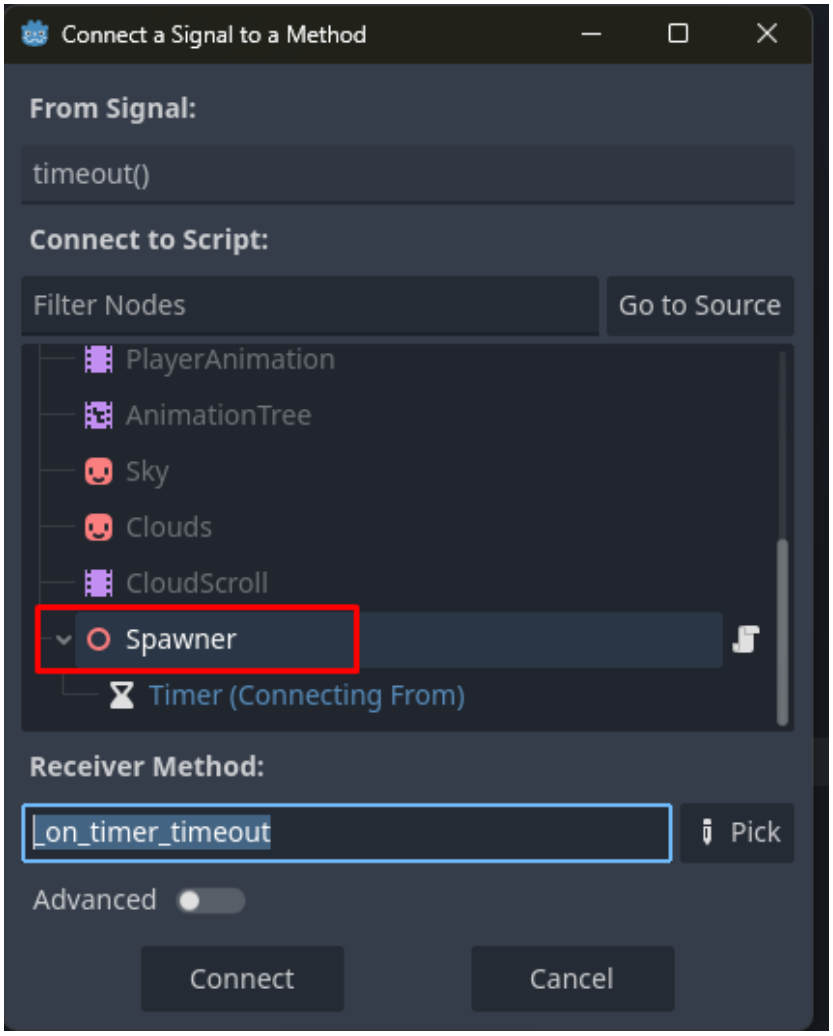




Reminder:

A signal sends a message from one piece of code in the program to another, triggering an event.

27 In the **Connect a Signal to a Method** window, select the **Spawner** node. Click **Connect**.

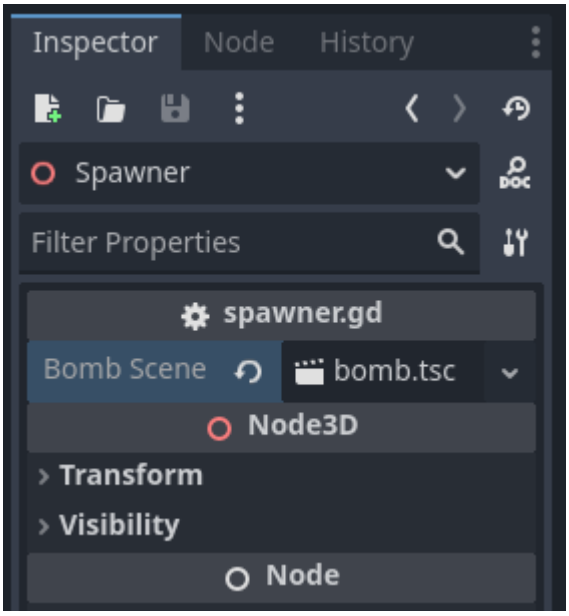


28 In the **spawner.gd** script, there will now be a green arrow icon next to the `_on_timer_timeout()` function, indicating that this function will be triggered by the **Timer** node.

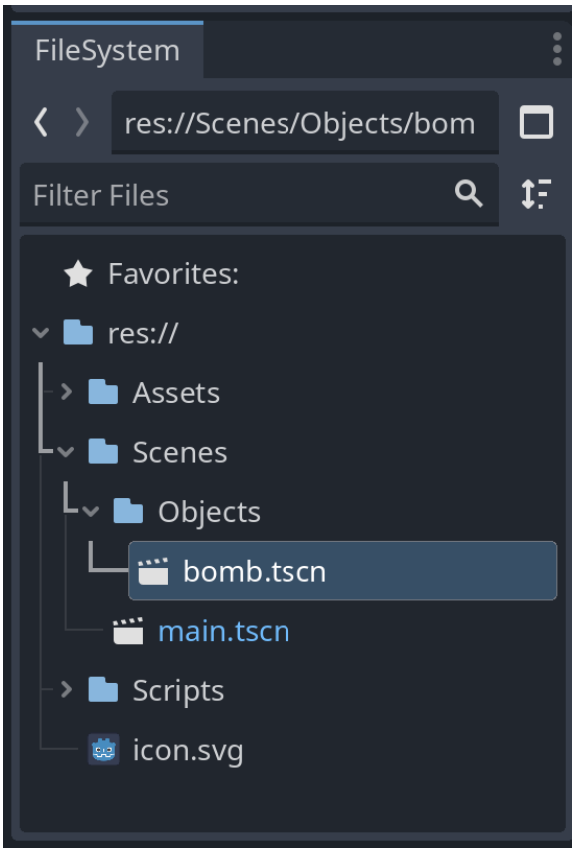
```
21  # --------------------
22  # TODO 3
23  # Instantiate a bomb object and give it a spawn position
24  # --------------------
25  func _on_timer_timeout() -> void:
26      var bomb = bomb_scene.instantiate()
27      bomb.position = Vector3(randf_range(-spawn_x, spawn_x), spawn_y, 0)
28      add_child(bomb)
29
```

29 In the **Inspector** for the **Spawner** node, assign the **Bomb Scene** variable.

Select **Quick Load...** then assign the **bomb** packed scene.

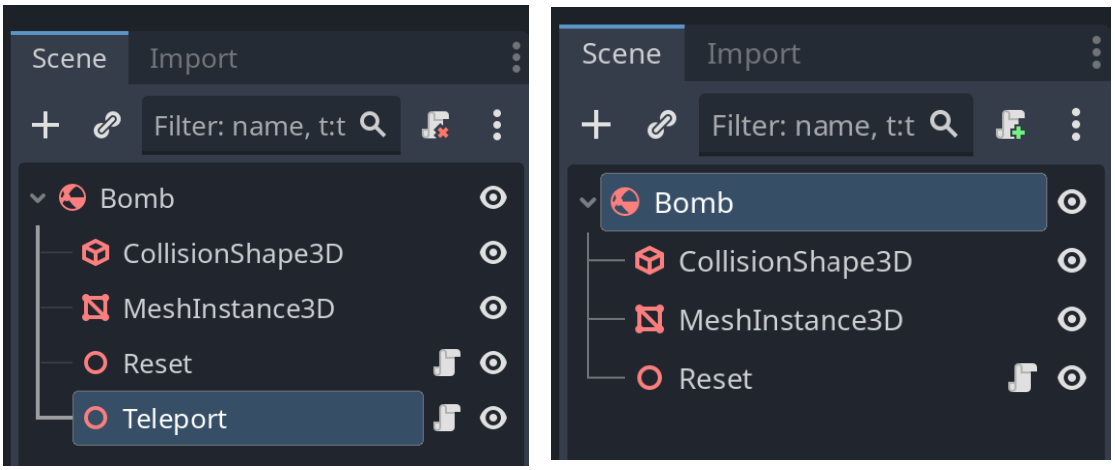


30 In **FileSystem**, open **bomb.tscn**.



31 Delete the **Teleport** child node, then save the scene.

The teleport node isn't needed anymore, because the **spawner** script will be handling creating and removing bombs from the game instead.

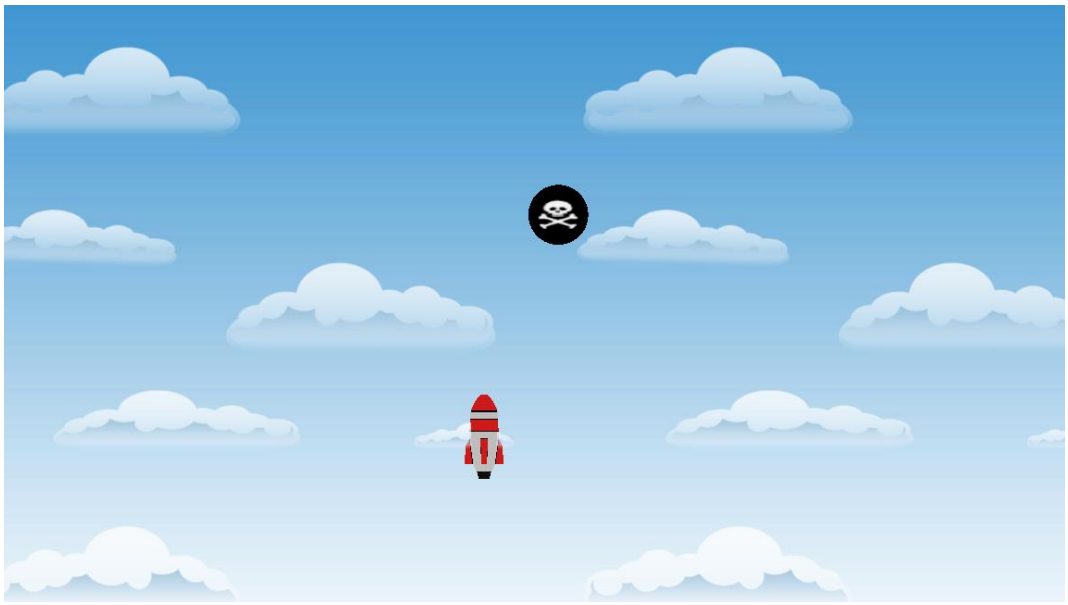


32 Playtest the game.

The bombs will now spawn on the timer, at a random horizontal position.

They aren't falling quite right, however. They appear at the top of the screen and fall very slowly at first, giving the player plenty of time to move and making the game too easy.

This can be fixed by changing where the bomb spawns into the game.



33 In the `spawner` script, find the `_on_timer_timeout()` function.

Locate the code that sets the bomb's position. Increase the `spawn_y` value by **5** in the second parameter of `Vector3()`.

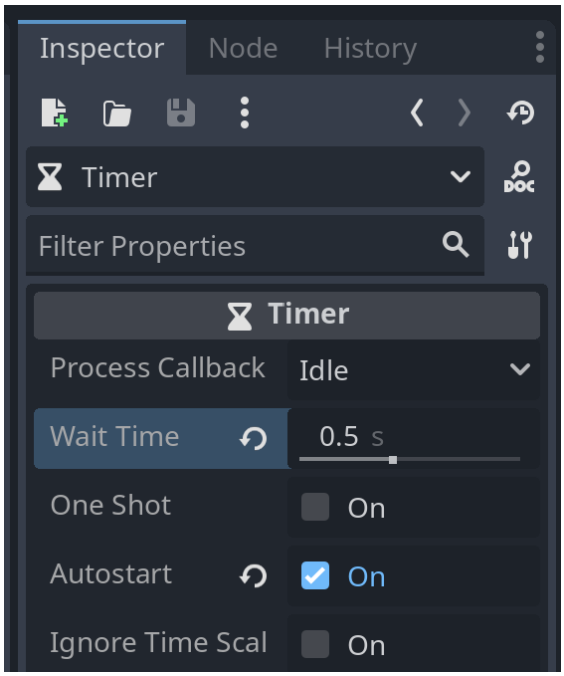
This will make the bomb spawn a bit higher, allowing it to pick up speed before appearing in the viewport.

```
21  # --------------------
22  # TODO 3
23  # Instantiate a bomb object and give it a spawn position
24  # --------------------
25  func _on_timer_timeout() -> void:
26      var bomb = bomb_scene.instantiate()
27      bomb.position = Vector3(randf_range(-spawn_x, spawn_x), spawn_y + 5, 0)
28      add_child(bomb)
29
```

34 Adjust the amount of time between each bomb spawning!

In the **Inspector** of the **Timer** node (in the **Main** scene), find the **Wait Time** property and adjust it.

Playtest the game while adjusting the **Wait Time** until it feels right.



Pro Tip:

For shorter wait times, try a decimal number like 0.1 - 0.9. For longer wait times, try numbers higher than 1.

Pause for **Sensei Stop #2**!

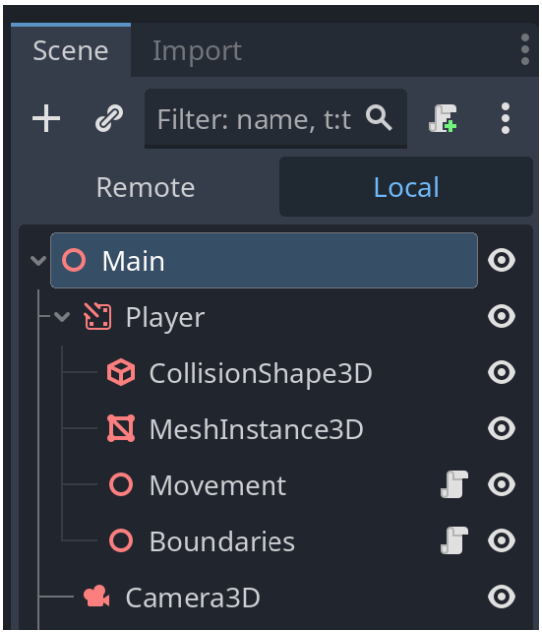
Check in with a Code Sensei before moving on; make sure the bombs are falling from the sky properly.

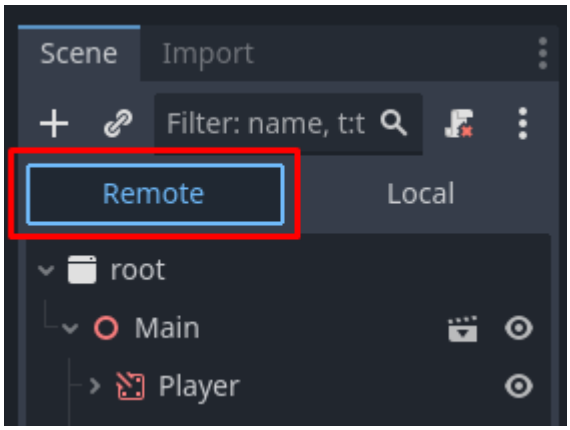
Reminder: Save your work!

35 Playtest the project.

When the game starts, take a second to pause it, then go back to the **Scene** menu. Notice the two options above the list of nodes: **Remote** and **Local**.

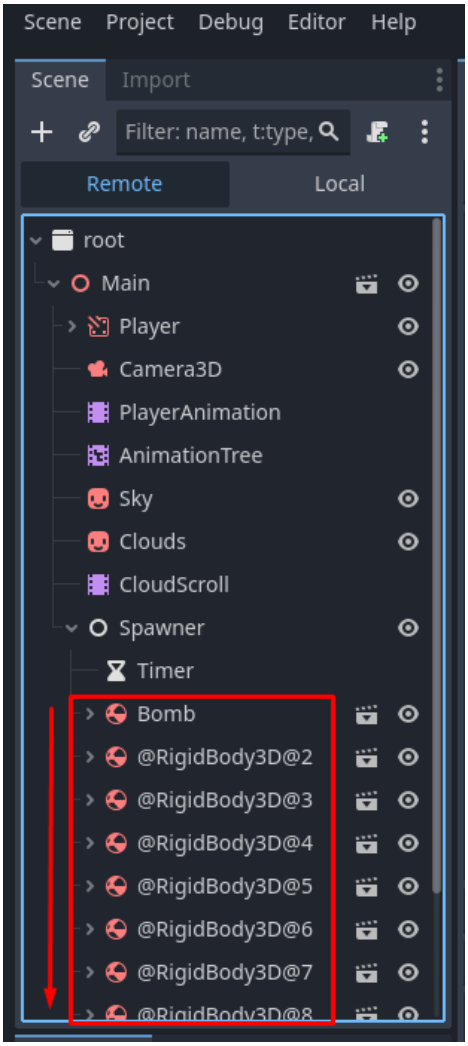
Click on **Remote**.





36 Expand the **Main** node and the **Timer** child node. Keep this visible and resume playing the game. As the game runs and more bombs are created, notice how the list of nodes increases as new nodes are added.

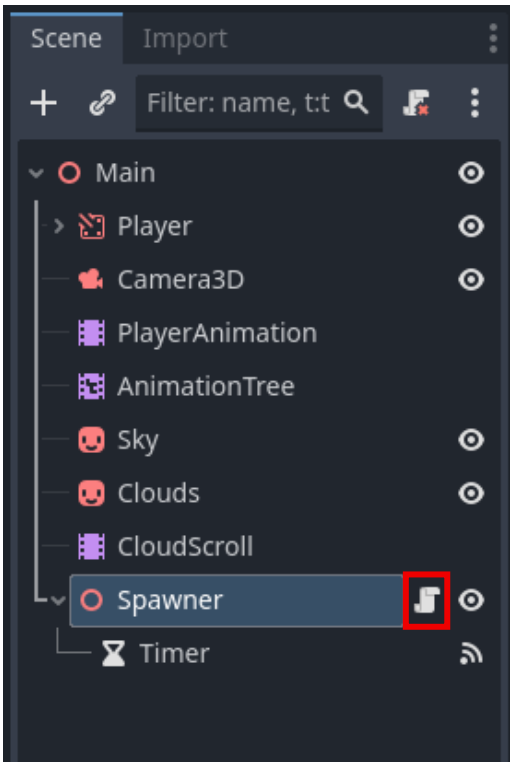
How might these new bombs be removed, once they're no longer needed? Why is it important for them to be removed?



Pro Tip:

The new nodes are called **@RigidBody3D** with a number added because Godot is auto-naming the newly created nodes since the code does not assign a name to the new nodes.

37 Close the playtesting window. Open the **spawner.gd** script.



38 Under **TODO 4,** use the code completion to define a `_process()` method.

This function will be used to remove bombs from the game once they fall past the area that the player can see. This is done to save resources and avoid crashing the game if it runs for too long.

```
30  # --------------------
31  # TODO 4
32  # Remove bomb objects when they've left the viewport
33  # --------------------
34  func _process(_delta: float) -> void:
```

Pro Tip:

The `delta` parameter won't be used, so prefix it with an underscore _.

39 Inside the `_process()` method, declare a local `all_bombs` variable.

Set the variable's value using the `get_tree().get_nodes_in_group()` method to make an array of all the **"Bomb"** nodes.

`all_bombs` is now an array of all the bomb nodes that the game is generating. How will this information be used to get rid of bombs that aren't needed anymore?

```
30  # --------------------
31  # TODO 4
32  # Remove bomb objects when they've left the viewport
33  # --------------------
34  func _process(_delta: float) -> void:
35      var all_bombs = get_tree().get_nodes_in_group("Bomb")
```

`get_nodes_in_group()`: Part of the SceneTree class, this method returns an array of all nodes inside the tree which have been added to the given **group.**

Parameters:

1. `group` ***(String):*** the group to be checked (`"Player"`).

Returns *(Array[Node]):* all nodes that are inside the tree with the given **group**.

40 After the `all_bombs` array declaration, use a `for` loop to check each item in that array for something.

```
30  # --------------------
31  # TODO 4
32  # Remove bomb objects when they've left the viewport
33  # --------------------
34  func _process(_delta: float) -> void:
35      var all_bombs = get_tree().get_nodes_in_group("Bomb")
36      for bomb in all_bombs:
37          
```

A **for loop** is a chunk of code that repeats the same action and only stops when a certain condition is met. When a **for loop** is used to check items in an array, that's called **iteration.**

Think back to how a `for element of` loop was used in MakeCode. What similarities do you notice between how a `for element of` loop is written in JavaScript and a **for loop** in GDscript? What syntax differences do you notice?

```
55  forever(function () {
56      for (let bomb of allbombs) {
57
58      }
```

41 On the first line inside the `for`-loop, write an `if` statement that checks if the value of the **bomb's y position** is less than `-spawn_y`.

```
30  # --------------------
31  # TODO 4
32  # Remove bomb objects when they've left the viewport
33  # --------------------
34  func _process(_delta: float) -> void:
35      var all_bombs = get_tree().get_nodes_in_group("Bomb")
36      for bomb in all_bombs:
37          if bomb.position.y < -spawn_y:
38
```

```
55  forever(function () {
56      for (let bomb of allbombs) {
57          if (bomb.y < -1 * player2.y) {
58
59          }
60      }
```

42 Inside the `if`-statement, use `queue_free()` to remove the currently iterated `bomb`.

Now, the `_process()` method will make an array of all the bomb nodes currently in the game. If the y-position of any bomb on that array is less than `-spawn_y`, it will be deleted.

```
30  # --------------------
31  # TODO 4
32  # Remove bomb objects when they've left the viewport
33  # --------------------
34  func _process(_delta: float) -> void:
35      var all_bombs = get_tree().get_nodes_in_group("Bomb")
36      for bomb in all_bombs:
37          if bomb.position.y < -spawn_y:
38              bomb.queue_free()
39
```

```
55  forever(function () {
56      for (let bomb of allbombs) {
57          if (bomb.y < -1 * player2.y) {
58              sprites.destroy(bomb)
59          }
60      }
61  })
```

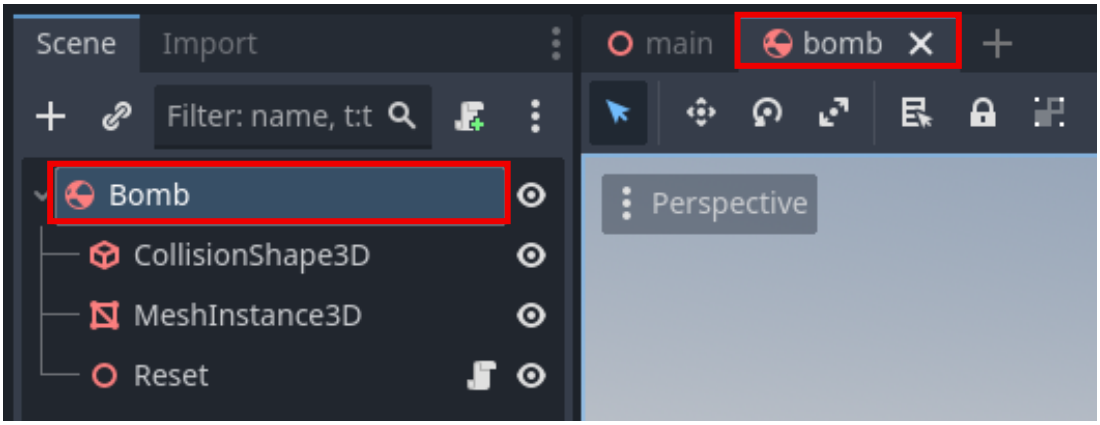
Reminder:

Check that the code is properly indented inside the `if` statement!

43 To use the function that was created in the previous steps, the **Bomb** scene will be added to a new **group** called "**Bomb**."

By using the `get_nodes_in_group()` method and passing the "**Bomb**" group label to it, the **spawner** script will use the **Bomb group** to access all the bombs that will be created.

Open the **bomb scene** and select the **Bomb** node.

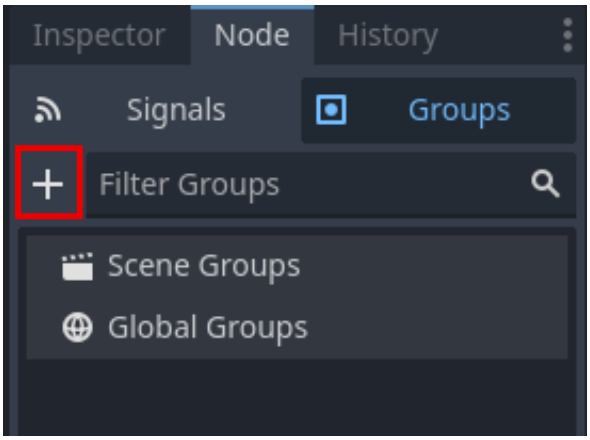


New Concept: Groups

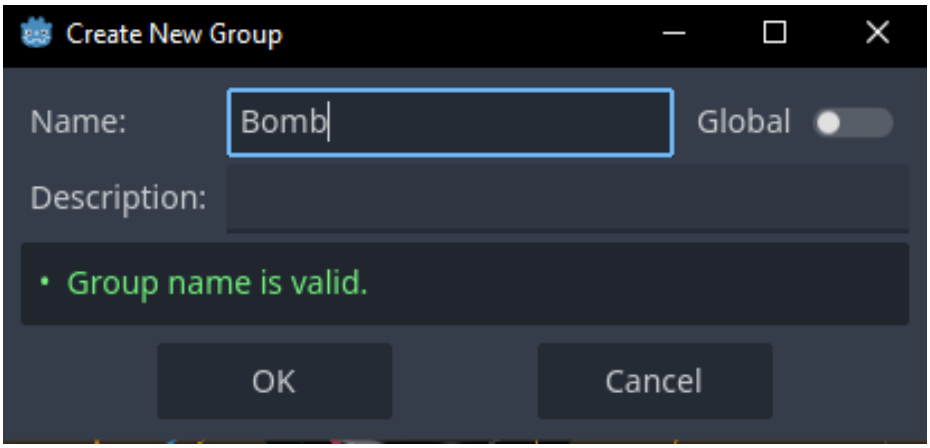
Groups act as Godot's tagging system, where the developer can assign multiple **groups** to each node. **Groups** allow the developer to perform actions (like calling methods) on all nodes in a **group** at once.

44 In the **Inspector** for the **Bomb** node, click on the **Node** tab. Underneath, open the **Groups** menu.

Click **+** to create a new group.



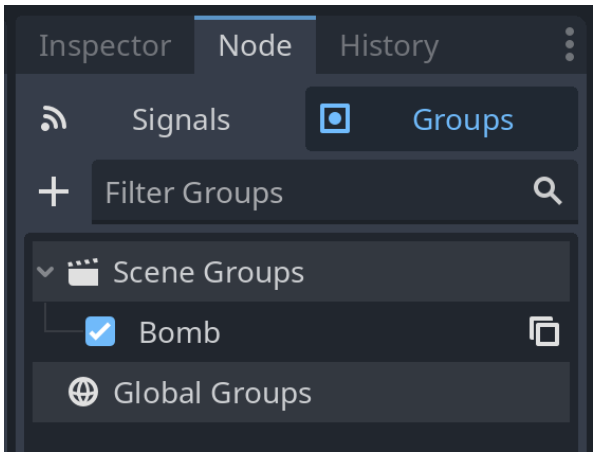
45 In the **Create New Group** window, type "**Bomb"** for the group name.



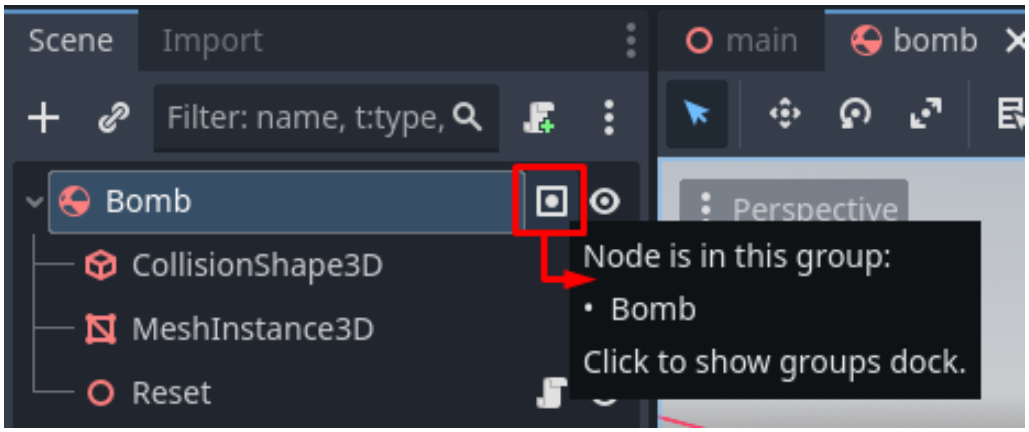
Pro Tip:

Make sure the group name is capitalized! If it doesn't match the name used in the spawner script's `_process()` method, the code won't be able to see it.

46 Now that the **Bomb** scene has been used in a group, it will appear on the list of scene groups.

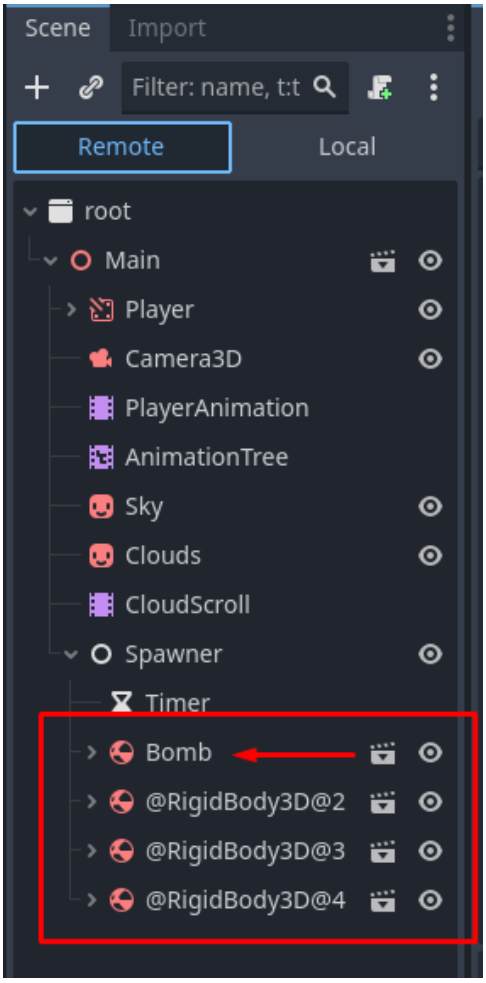


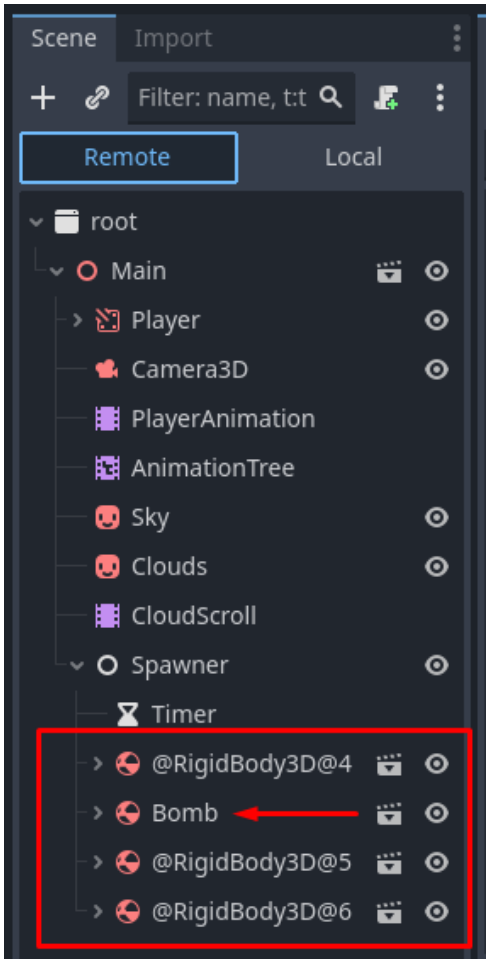
In **Scene**, notice the square icon that appears next to the **Bomb** node that shows this.



47 Playtest the game again, keeping an eye on the **Remote** tab in **Scene**.

Under the **Spawner** node, notice how **Bomb** nodes that fall out of the screen area are deleted.





This saves memory and computing resources, allowing the program to run for long periods without a problem.

Reminder:

This will be seen as the node named “bomb” gets recreated periodically – the program can only add a node with the name “bomb” after the old node with the same name has been destroyed.

Pause for **Sensei Stop #3**!

Congratulations on exploring using packed scenes, the viewport, groups, and deleting nodes that fall too far!

Before submitting, check in with a Code Sensei to make sure the project works as intended then reflect on the following:

- Why would a game developer want to do these tasks in the first place?
- What did you enjoy most when creating this project?
- What was something you found difficult and why?

Reminder: Save your work!

Congratulations on completing **BB Activity 09: Dropping Bombs Part 3** and in Godot – **You Rock**! You are now ready to save this project and submit it.

Continue your exploration with Godot by opening the **BB Activity 10: Dropping Bombs Part 4** Ninja Guide.