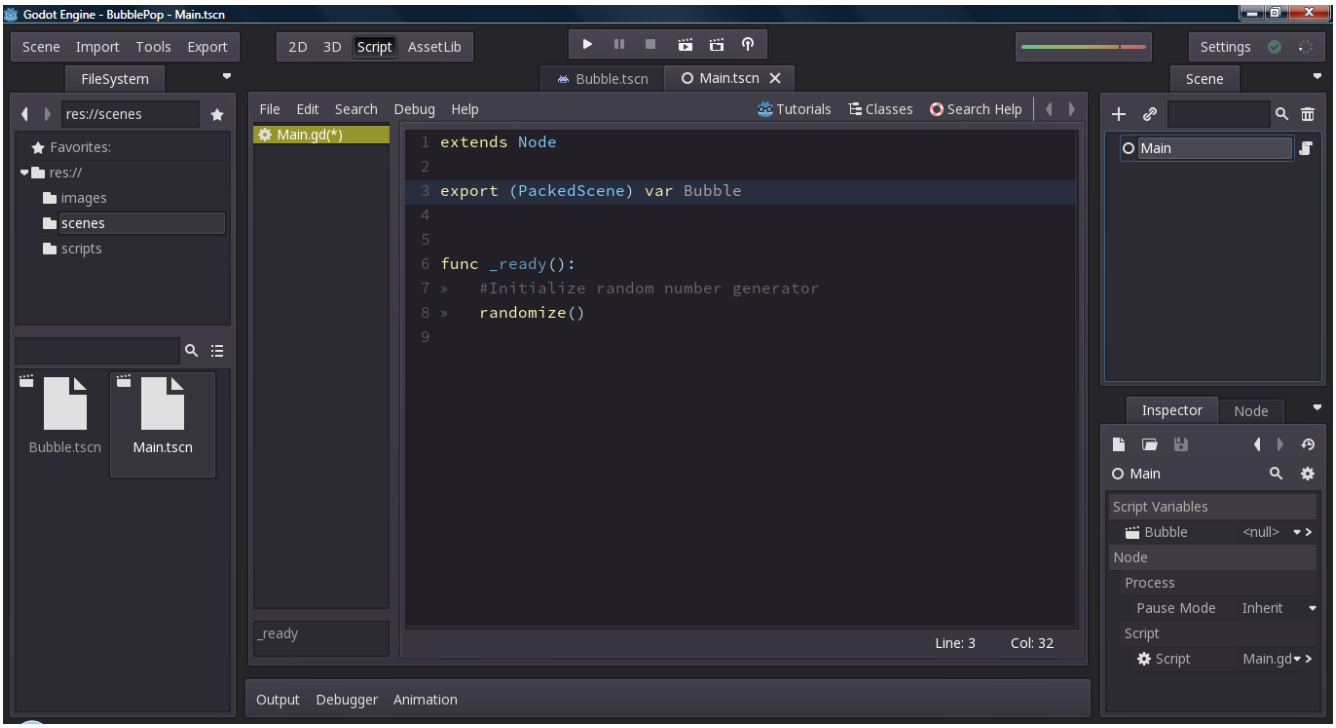
Godot 2D Game
Lesson 5: Timers and Instancing

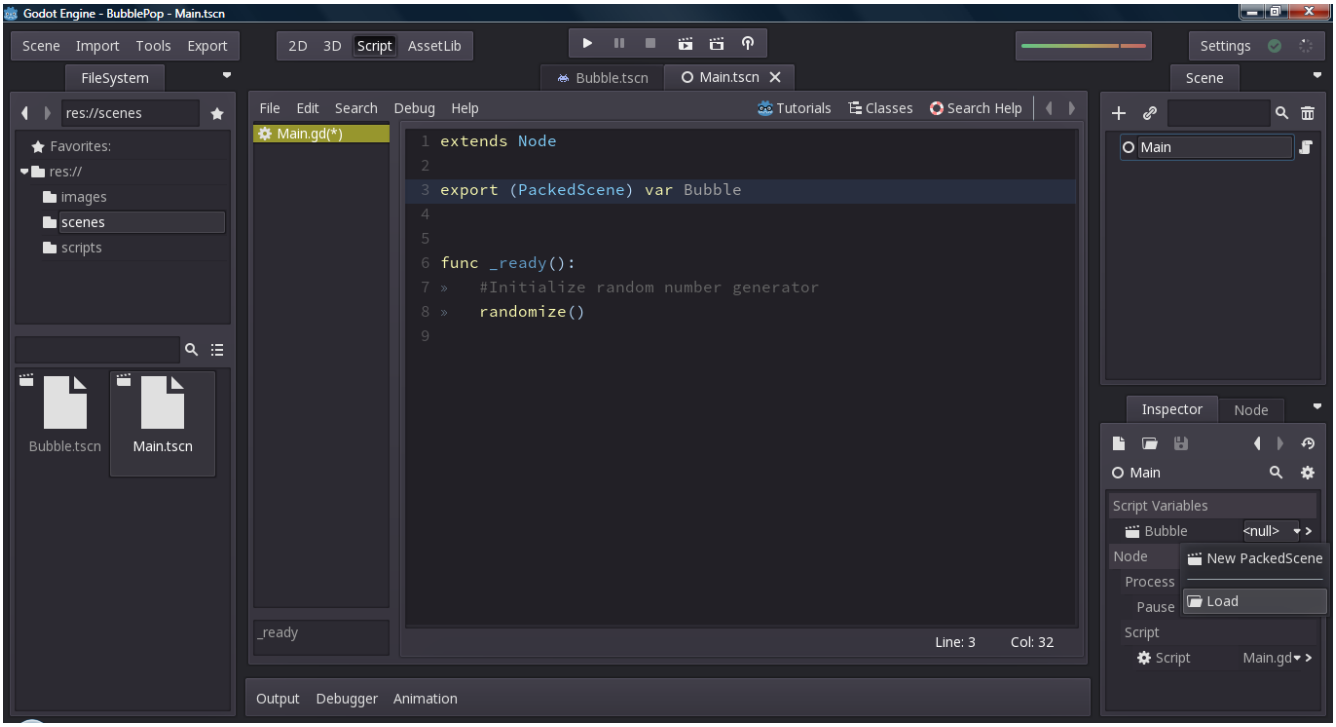
In this lesson, we are going to write some code to start spawning bubbles. First, we will need to obtain a reference to our bubble scene so we can use it in our main scene. We will do so by placing this code after the first line of our script:

```
export (PackedScene) var Bubble
```

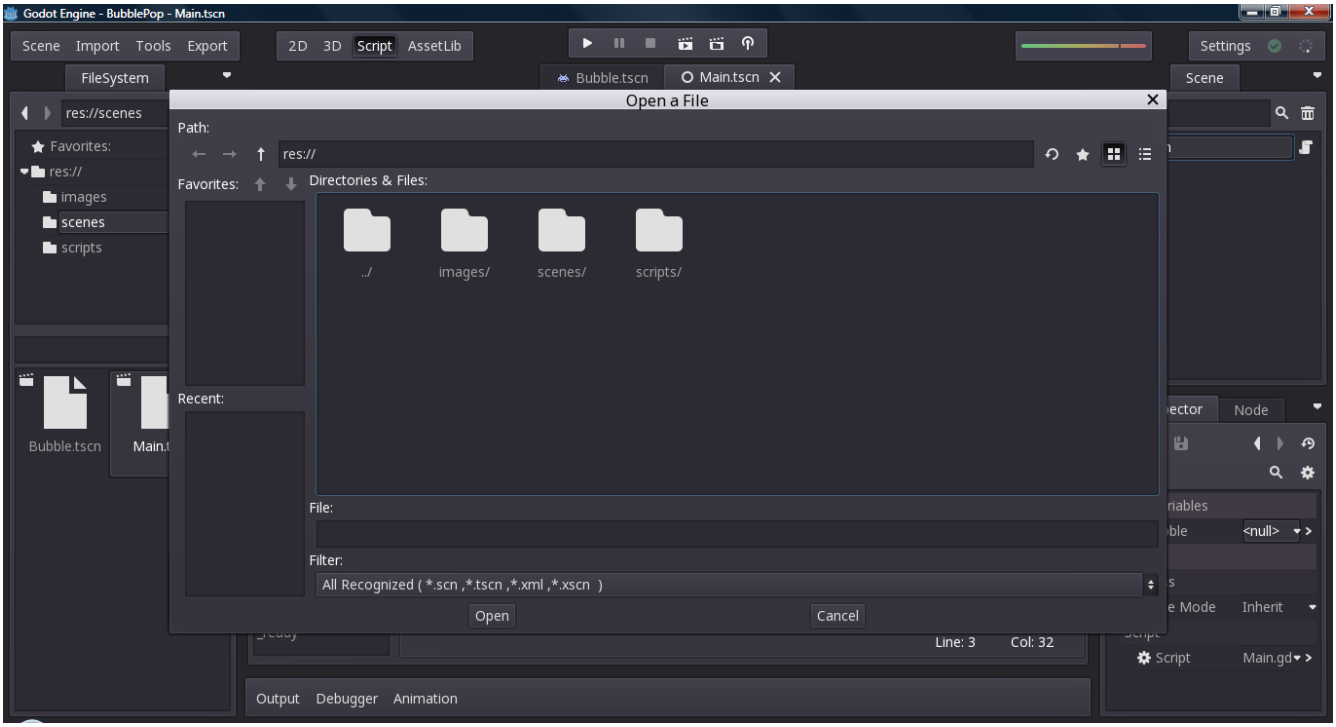
The "var" keyword is used to declare a new variable. Adding the "export" keyword followed by the variable type in parenthesis before the variable declaration will make this variable visible as a property in the lower right pane:



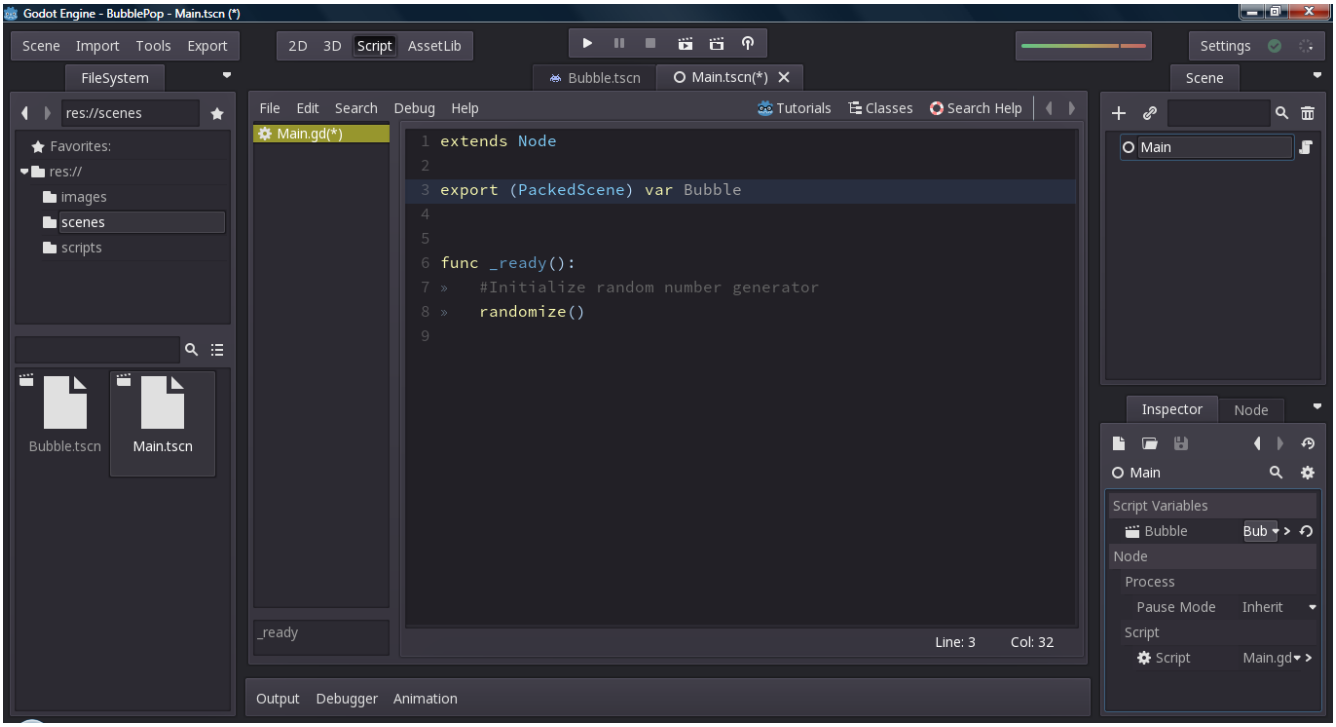
Now if we click the box beside the "Bubble" property we will get this menu:



Choose load and this dialog will open:



Next, we need to double-click the “scenes” folder and select our “Bubble” scene. Then the editor window will look like this:

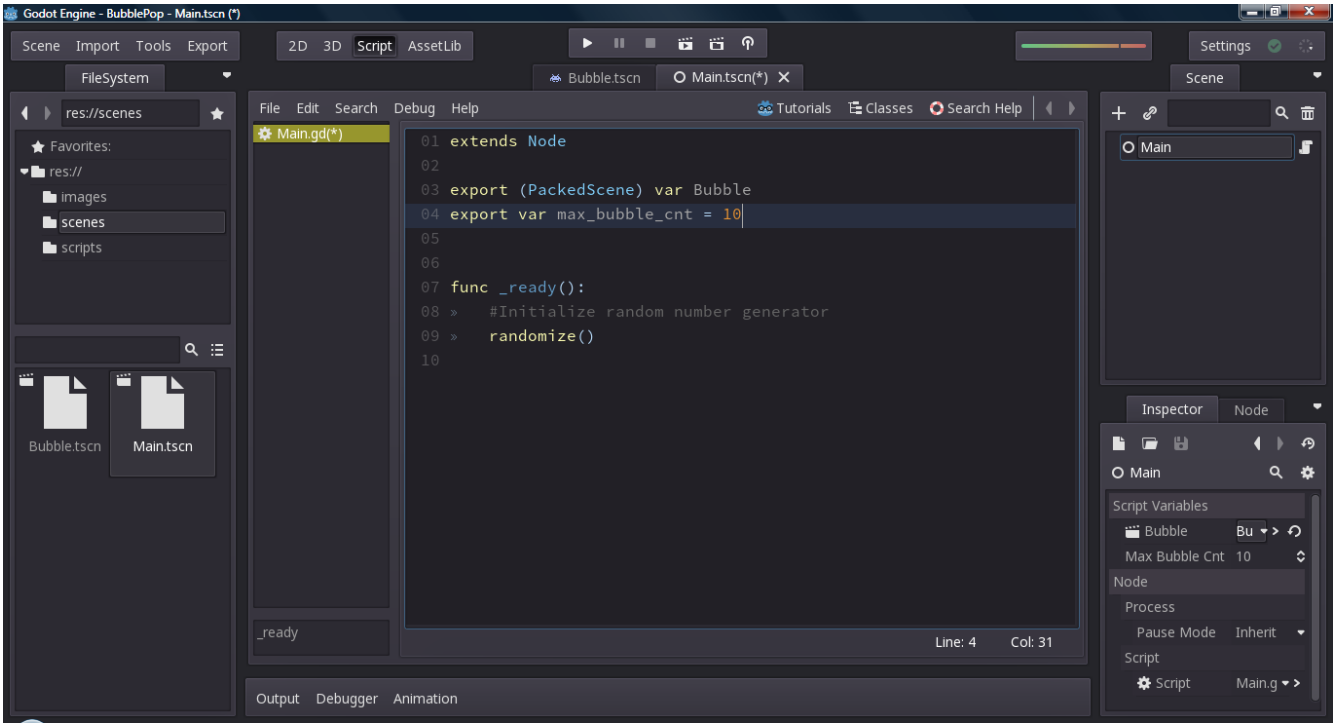


As you can see, the property now contains a reference to our "Bubble" scene. When we run the game, the "Bubble" variable will contain that reference which we can use to create bubble instances. An instance is simply a copy of an object that has its own unique properties. For example, if we have 3 bubbles, each bubble is an instance and has its own unique position and velocity.

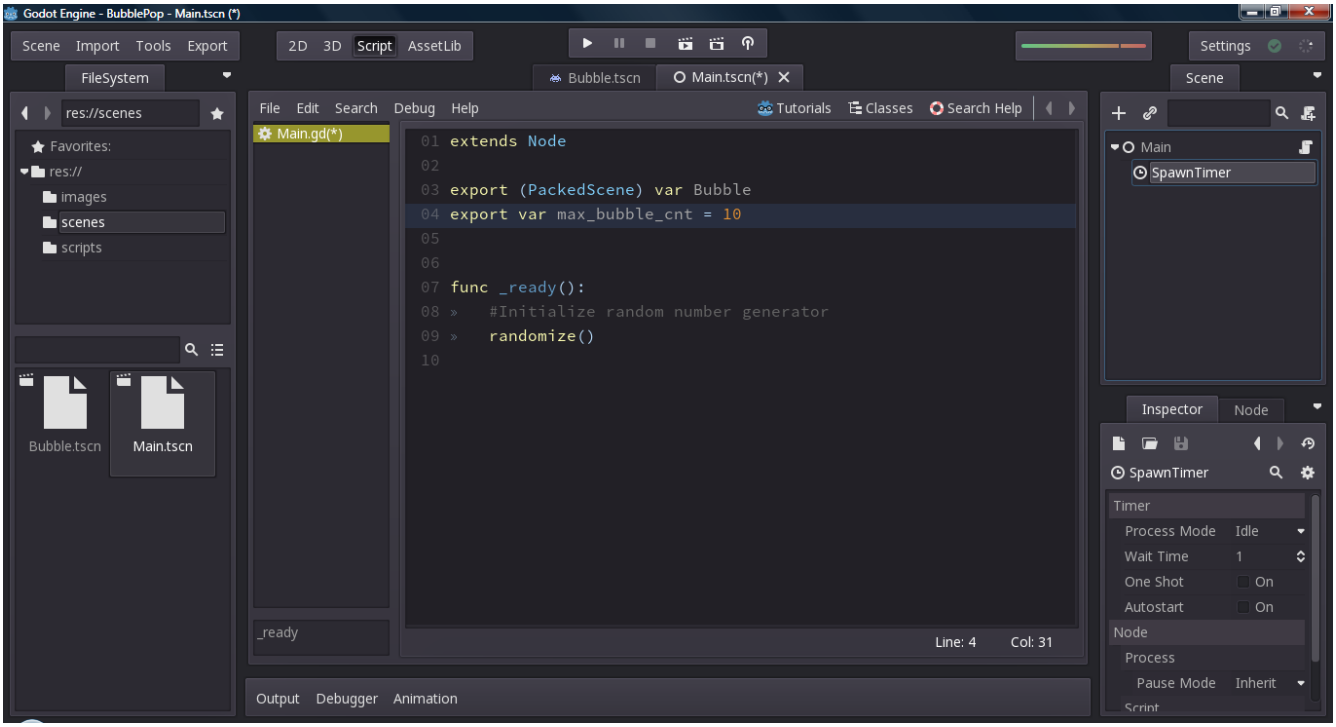
Next, let's add another property that controls the number of bubbles we can have at a time:

```
export (PackedScene) var Bubble
export var max_bubble_cnt = 10
```

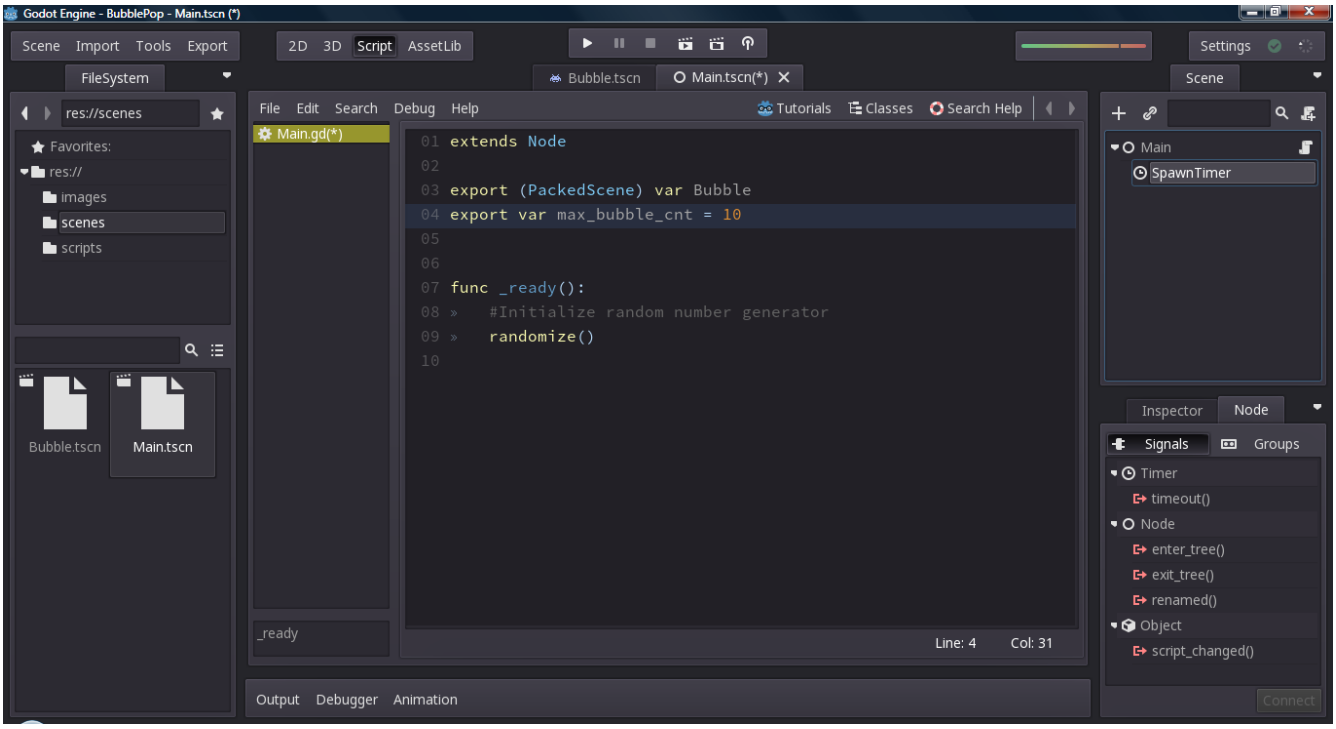
"max_bubble_cnt" will be the maximum number of bubbles that can be present at the same time. Notice how this time we gave no type and set an initial value. If we give an initial value for a variable and export it, the type will be inferred based on the variable's initial value. We can still adjust the initial value via the lower right pane too:



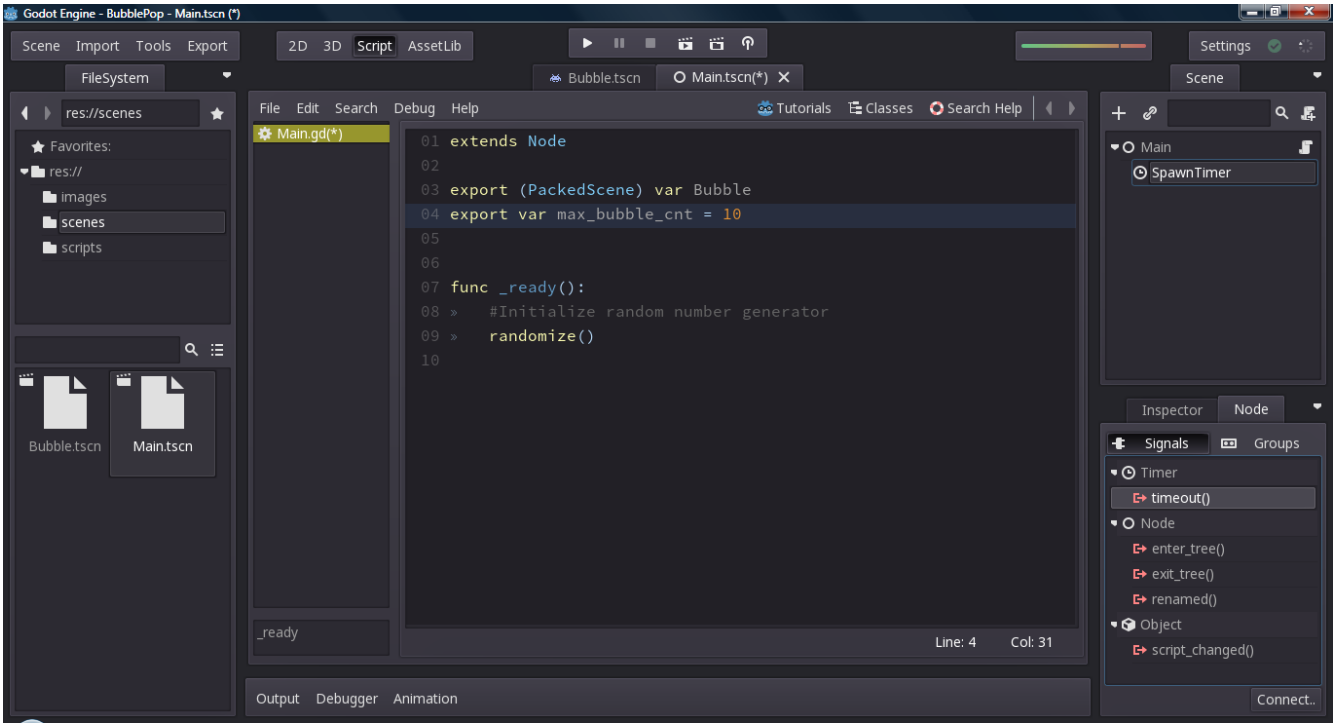
Now that we have that setup, let's setup the bubble spawning code. We want bubbles to spawn at regular intervals if there are less than 10 bubbles already in play. But how can we write code that runs at regular intervals? There is actually a type of node called a Timer that we can use. Right-click the "Main" node and choose "Add Child Node". Then create a new Timer node like this:



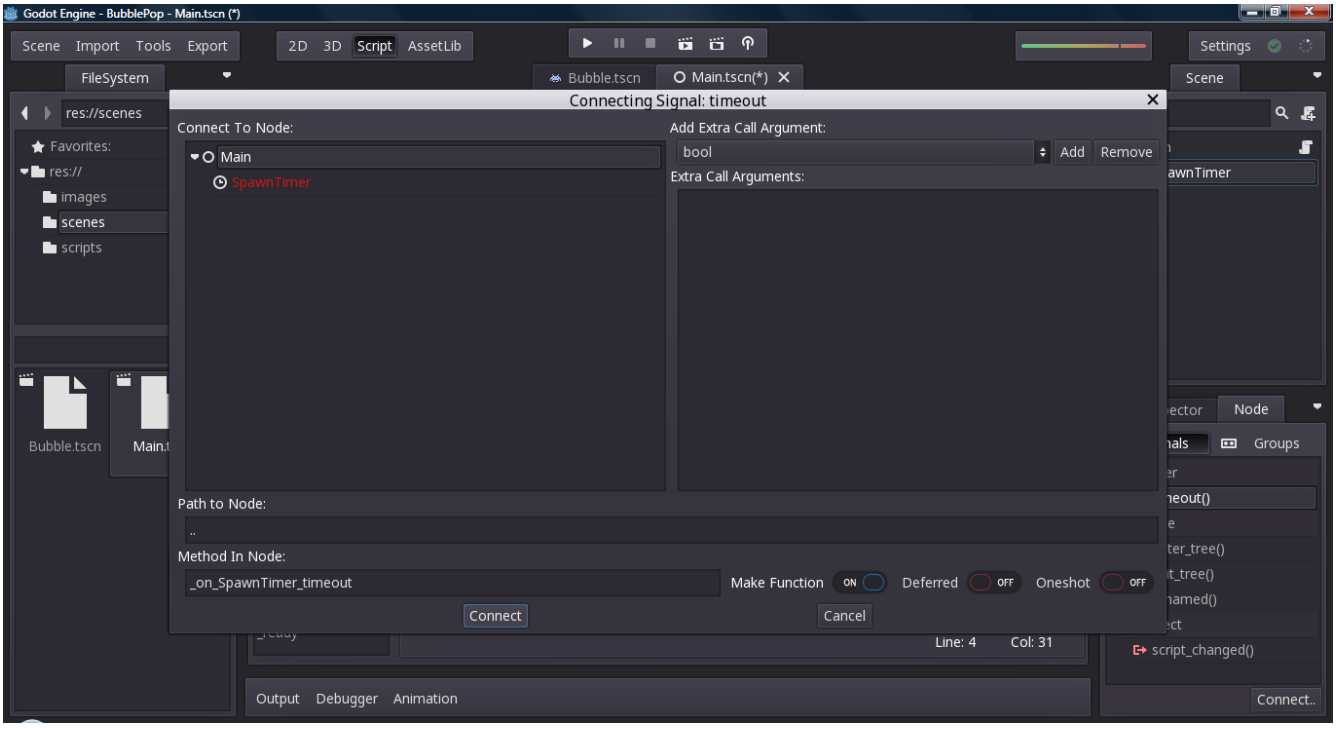
We will call this our “SpawnTimer”. The “Wait Time” property of a Timer determines how often the timer fires. But how do we know when the timer fires? Click the “Node” tab in the lower right pane and you will see this:



The list you see now is the signals that the selected node can emit. Every timer emits the “timeout” signal whenever it fires. We can attach one or more functions to each signal. Whenever a node emits a signal, it will automatically call all the functions that are attached to that particular signal. Click on the “timeout” signal:



Now click the “Connect...” button to open the connect signal dialog:



For this part, the default settings are sufficient. Simply click the “Connect” button and a new function will be automatically generated:

```
func _on_SpawnTimer_timeout():
	pass # replace with function body
```

The default name of a signal handler function always has the format "_on_NodeName_signal", which is quite descriptive and usually sufficient. The default body of a new signal handler just has a single line that does nothing. Notice that there is a comment at the end of the line. Godot allows us to have comments at the end of a line as well as on a line by themself.

Now we can start writing our bubble spawning code right? Not quite yet. We first need to know what rectangular area we want to spawn our bubbles in. Let's start by creating another new variable at the top of our script:

```
export (PackedScene) var Bubble
export var max_bubble_cnt = 10

var screen_rect
```

Notice that our new variable is not an exported property and has no default value. Instead, it will be used to store the rectangular area of the screen. Now we need to modify our "_ready" function:

```
func _ready():
	#Initialize random number generator and get screen rect
	randomize()
	screen_rect = get_viewport().get_rect()
```

To get the rectangular area of the screen, we first need to get the current viewport. The viewport represents the visible area of the screen where we can see whatever content is drawn. Then we call its "get_rect" method to retrieve the rectangular area of the screen and store it into our "screen_rect" variable. A method is a function that belongs to an object. We can access any function or variable that belongs to an object by putting a period after the object and then the name of the function or variable.

We will also need a variable to keep track of the total number of bubbles in play:

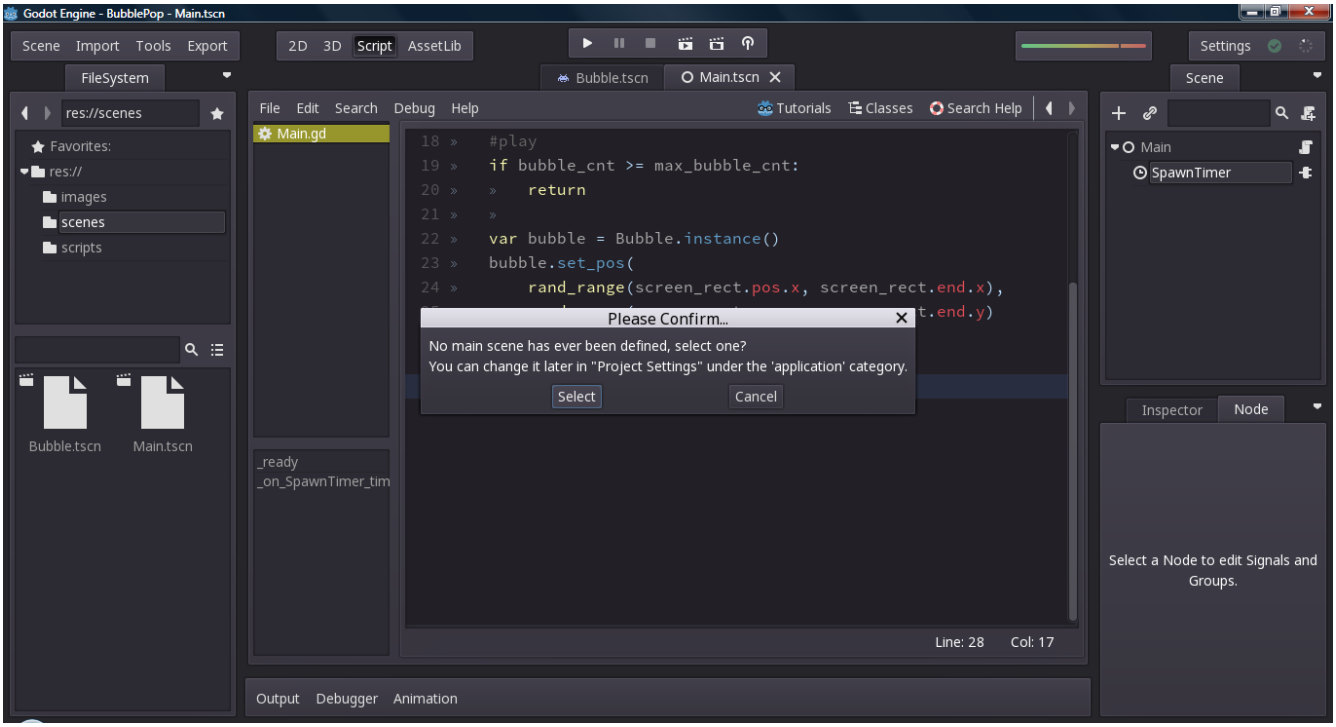
```
export (PackedScene) var Bubble
export var max_bubble_cnt = 10

var screen_rect
var bubble_cnt = 0
```

This time we will set the initial value of the variable to 0. And now we can write our bubble spawning code:

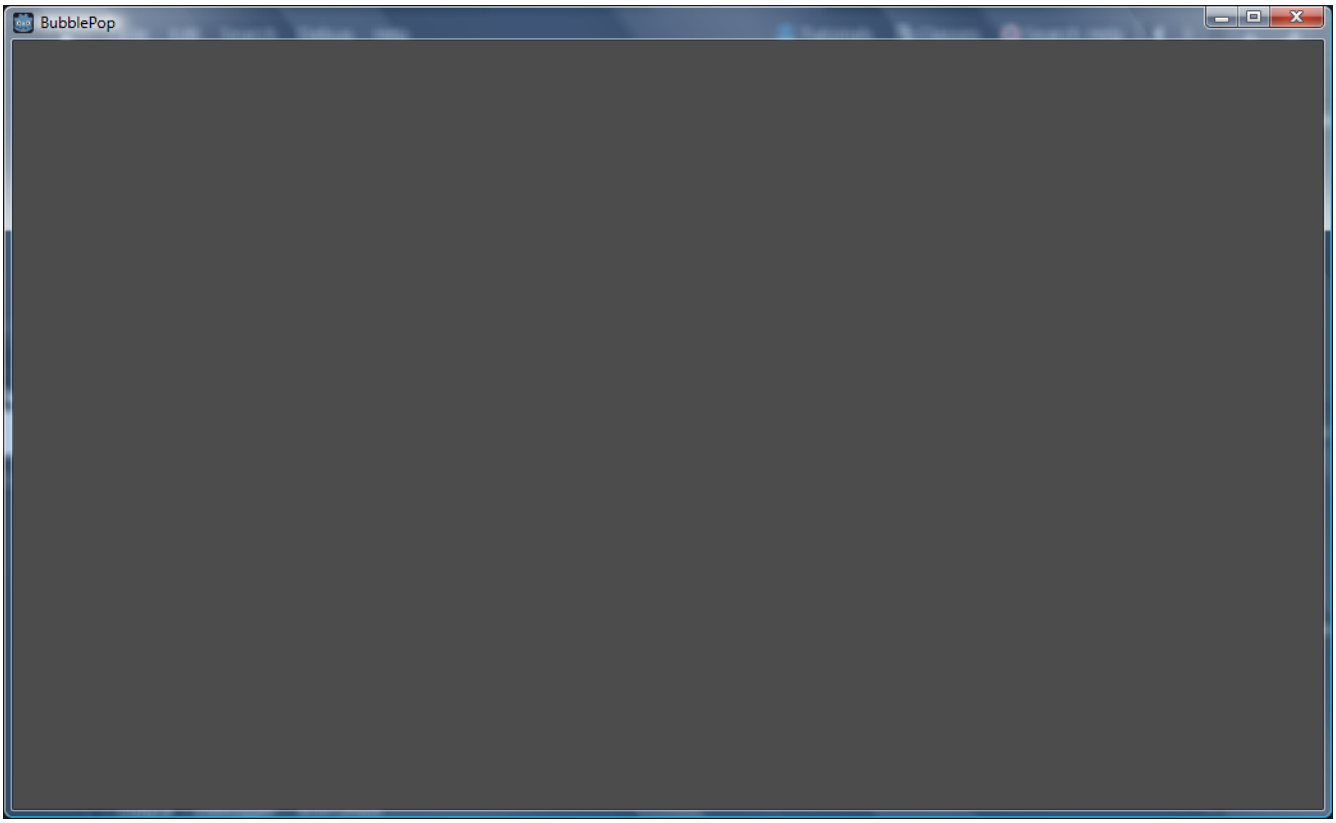
```
func _on_SpawnTimer_timeout():
	#Spawn a new bubble if there are less than 10 bubbles in
	#play
	if bubble_cnt >= max_bubble_cnt:
	    return

	var bubble = Bubble.instance()
	bubble.set_pos(
	    rand_range(screen_rect.pos.x, screen_rect.end.x),
	    rand_range(screen_rect.pos.y, screen_rect.end.y)
	)
	add_child(bubble)
	bubble_cnt += 1
```

We will start by checking if the current bubble count is greater than or equal to our maximum bubble count. If the condition is true, we will simply return because we don't want to spawn more bubbles yet. Notice that the contents of an if statement need to be indented just like the contents of a function do. Next, we will create a new instance of our bubble object by calling its "instance" method and store it into a variable called "bubble". Any variable declared within a function is called a local variable and only exists while we are inside the function. Whereas any variable created outside a function exists as long as the node the script is attached to exists. But don't worry, our bubble will still exist after we return from the function. I will explain how in a bit. Next, we will randomize the bubble's initial position. The "rand_range" function returns a random number between the 2 numbers we pass as parameters. In this case, we will use "screen_rect.pos.x" and "screen_rect.end.x" which refer to the left and right edges of the screen. In the second call to "rand_range" we will use "screen_rect.pos.y" and "screen_rect.end.y" which refer to the top and bottom edges of the screen. We will pass the 2 random numbers to the "set_pos" method of our new bubble object to set its position. Then we will pass our new bubble as a parameter to the "add_child" function. This will attach our new bubble to our "Main" node as a child node. Doing this will create a new reference to our bubble and prevent it from being lost when the function returns. Finally, we will add one to our bubble count.

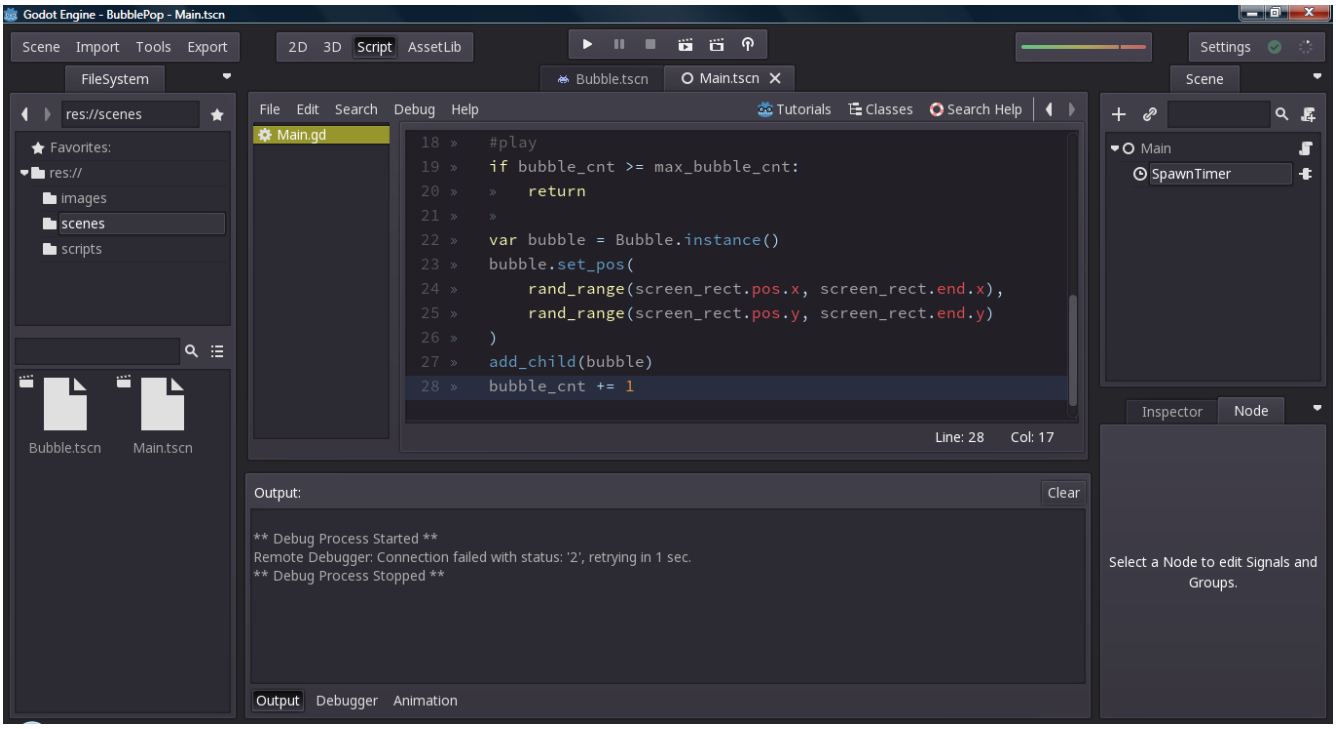
Now that we finished our bubble spawning code, let's test what we have made so far. Click the play button above the middle workspace and you will see this dialog:



Huh? What could this mean? In Godot, only one scene can be considered the main scene and the first time we click the play button, we need to select our main scene. Click “Select” and select your main scene. Afterward, a new window will open:



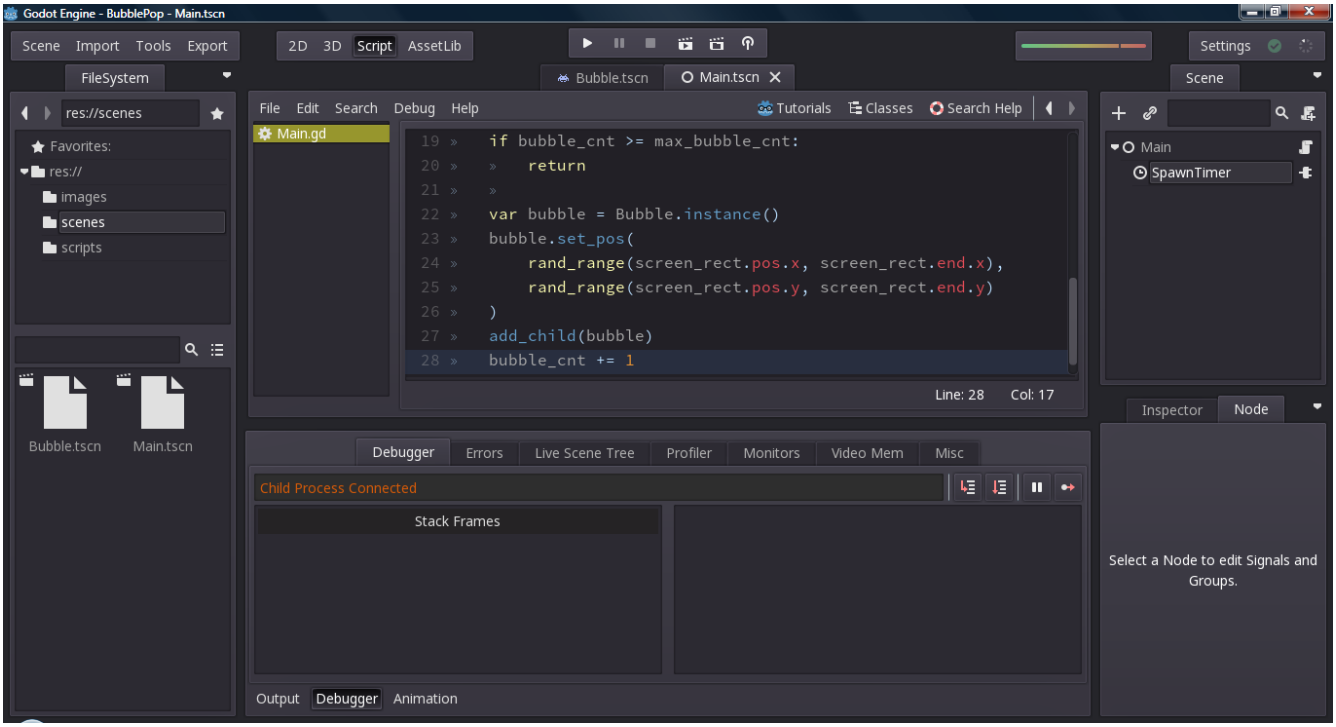
An empty window huh? What could be going on? The problem is that we never told Godot to start our spawn timer. Every timer has started and stopped states. A timer only fires if it has been started. We need to tell Godot to start our spawn timer. Close the window and I will show you something else useful:



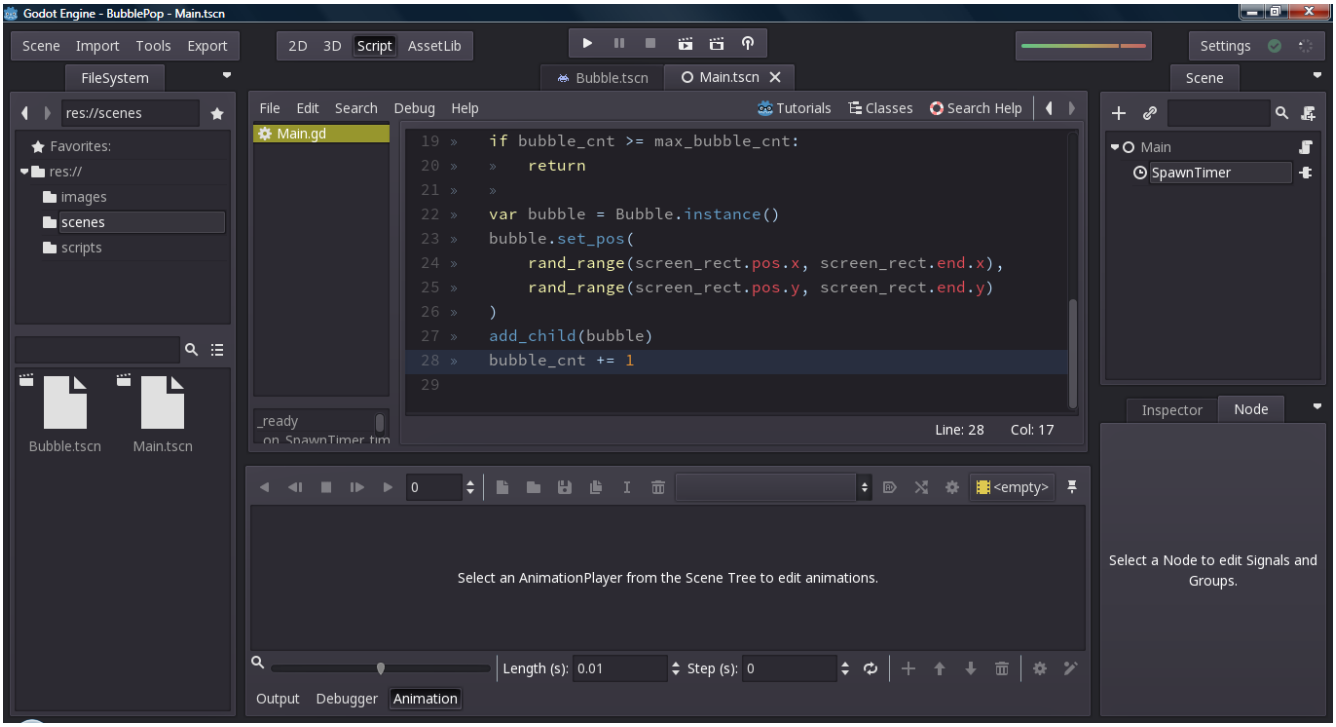
As you can see, a new pane has opened. This pane has 3 tabs called “Output”, “Debugger”, and “Animation”. The “Output” tab shows the output from your program. If you want your program to emit output, simply pass some text or objects to the “print” function as parameters such as:

```
print(“Hello World!”)
```

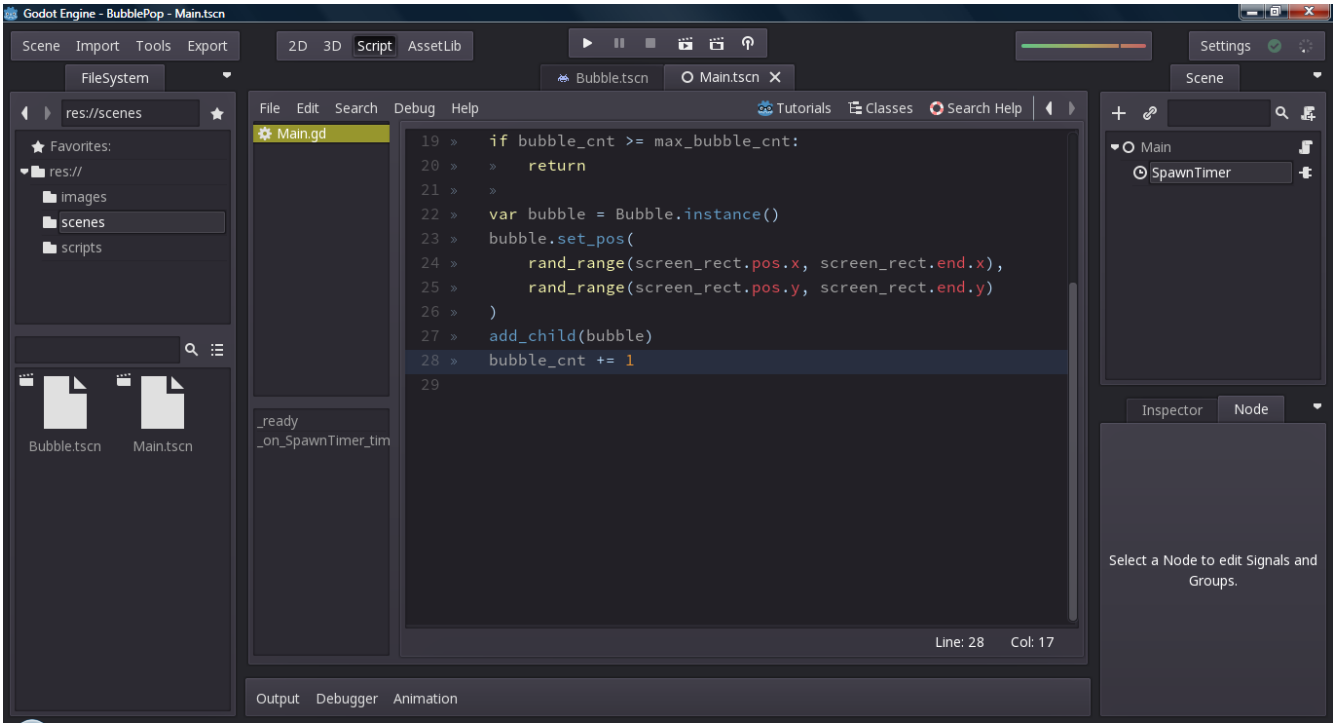
This is a great way to debug your games. The “Debugger” tab allows you to a wide variety of useful information about your game in realtime during a testing session:



The “Animation” tab allows you to edit animations for a node:



If you want to hide the bottom pane again. Simply click the tab that is active:



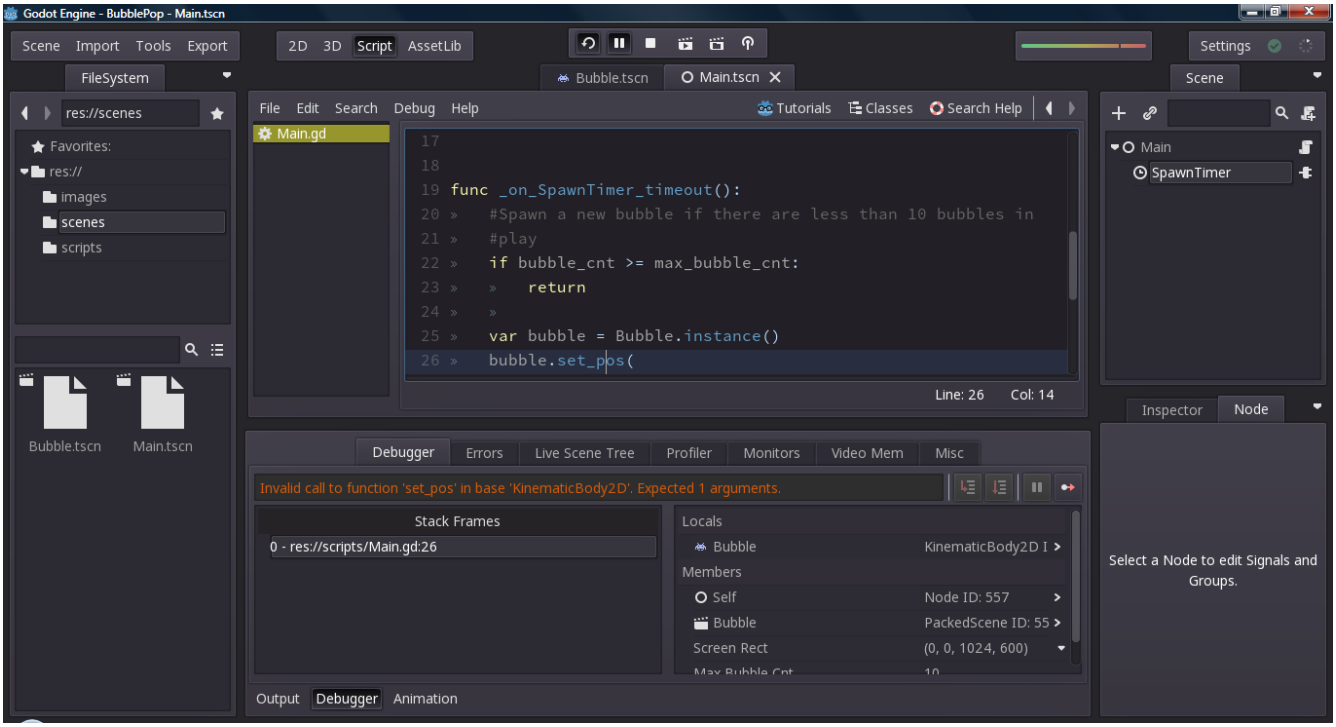
Now let's fix our game. Let's start by creating a new function called "start_game". This function will be called to start the game and it will start the spawn timer for us:

```
func start_game():
	#Reset the bubble count and start the spawn timer
	bubble_cnt = 0
	get_node("SpawnTimer").start()
```

Resetting the bubble count is simple, but what is this "get_node" function for? The "get_node" function is used to a child of the current node. We simply need to give a node path as a parameter. A node path can be just the name of a child node such as "SpawnTimer" or it can refer to a child of a child node like "Child/Grandchild". Once we have a reference to our spawn timer, we can simply call its "start" method to start it. Now we need to also update our "_ready" function so it will call our new "start_game" function:

```
func _ready():
	#Initialize random number generator and get screen rect
	randomize()
	screen_rect = get_viewport().get_rect()

	#Start the game
	start_game()
```

Now we are ready to try testing our game again. Click the play button and see what happens this time:



Shortly after the game starts, the editor window will come to the foreground and display the above error message. Whenever you make a mistake that causes your game to crash, this will happen. But don't worry, the error message can be used to determine what went wrong. In this case, the error message is "Invalid call to function 'set_pos' in base 'KinematicBody2D'. Expected 1 arguments." This particular message is telling me that the "set_pos" method of my bubble object requires one parameter. However, I gave it 2 in my code. Which is why the game crashed. Mistakes are inevitable when designing a game, but the important thing is to learn from them. And in the next lesson, I will teach you a valuable skill. How to debug your game.