CODE NINJAS®

Silver Belt Ninja Guide
Activity 11: Amazing Ninja Worlds Part 1

Activity 11

Amazing Ninja Worlds – Part 1

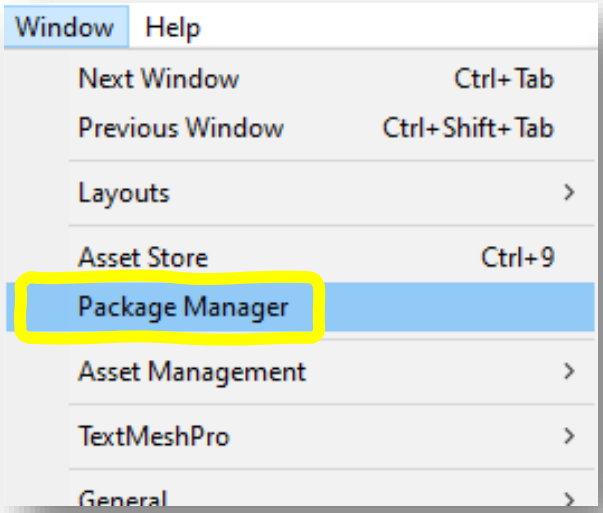
Super Ninja World is a side-scrolling adventure where the player must activate and find a teleportation pad to advance to the next level. There are a few obstacles that get in the way.

In this activity, you will start building the game by setting up a health system for the player.

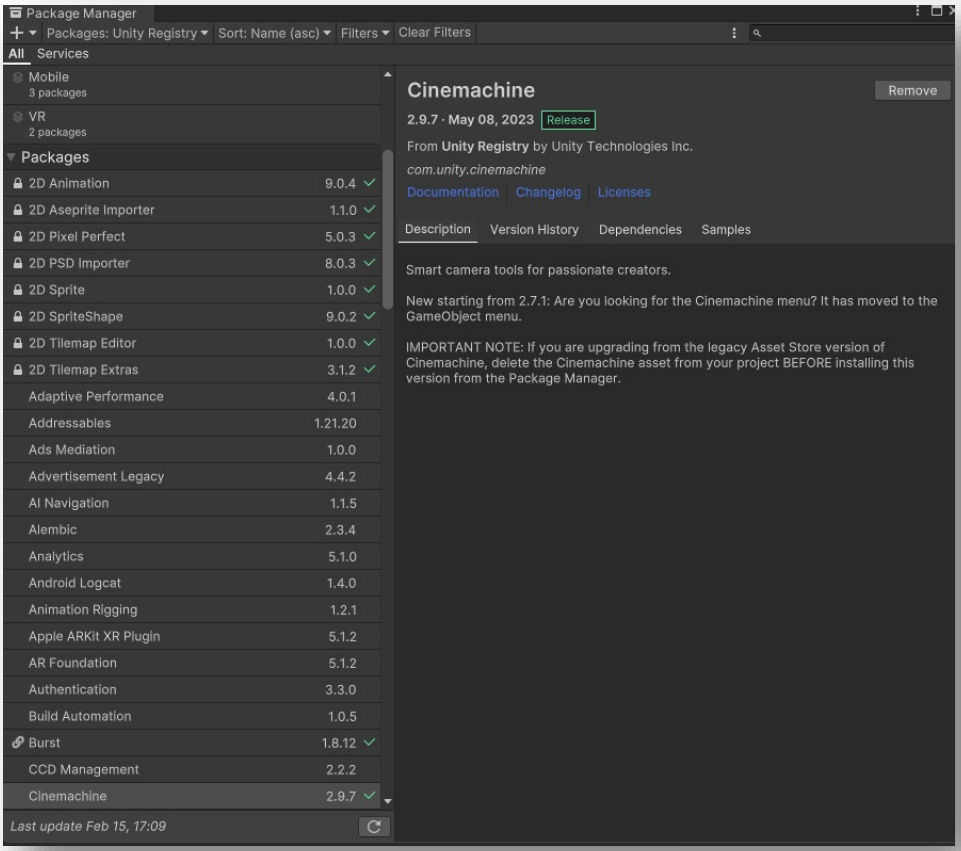
1 Start a new Unity Project and name it *YOUR INITIALS –Amazing Ninja Worlds*. Select **3D core**.

2 We will be using Cinemachine to control our camera. This will help us create awesome camera shots and angles for our game.
Go ahead and open **Window > Package Manager**.

Be patient if it doesn't open right away. It might take a minute to load.



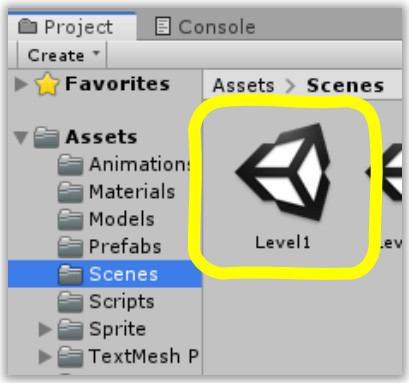
3 Find **Cinemachine** and click **Install** in the bottom right of the window.



Remember to switch to Unity Registry in the top left.

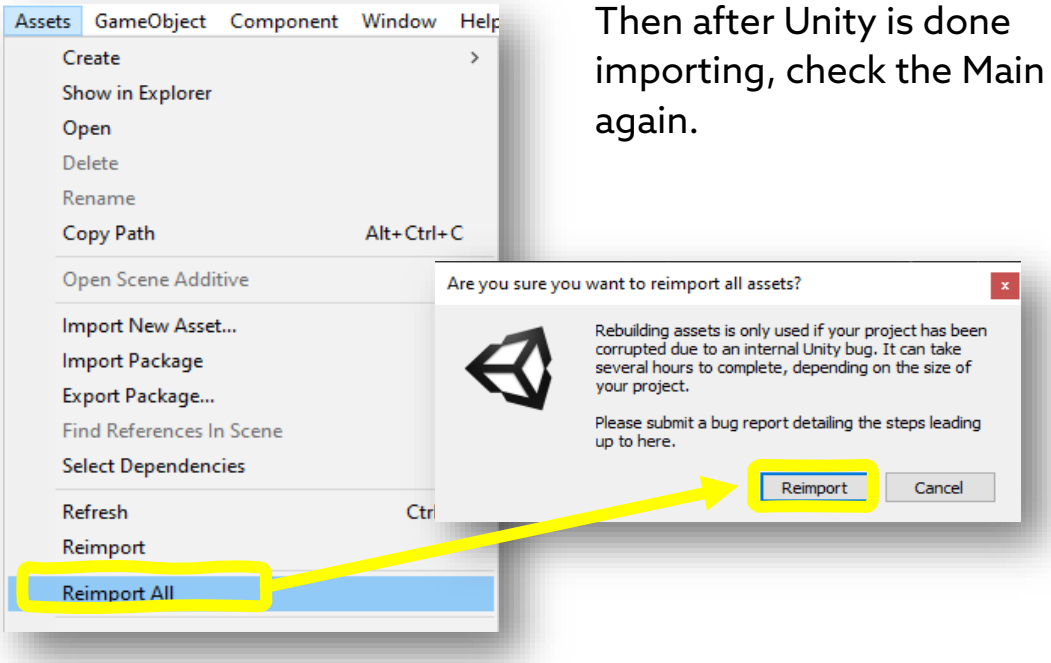
4 We've created a starter pack to give you a head start! To use it, import the **Activity 11 – ANWP1.unitypackage** by going to **Assets > Import Package > Custom Package> All > Import**.

5 To open the starter package, double-click on the **Level1** scene.
You can find this in the **Project** tab under **Assets > Scenes > Level1**.



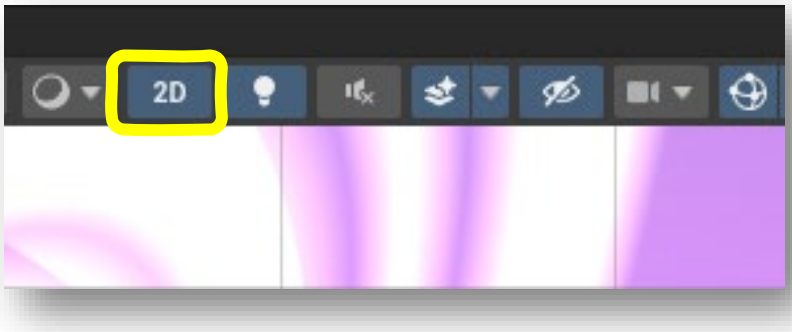
6 We should check that Cinemachine is working like it is meant to.
In the **Hierarchy** tab, you should see a list of all the game objects in the scene. If you see a little grey and red icon attached to the Main Camera, Cinemachine is working!

If you do not see it, go into the **Assets** tab and press **Reimport All**.



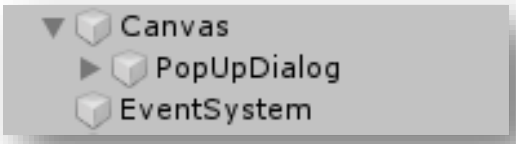
Then after Unity is done importing, check the Main Camera again.

7 Make sure you click the 2D button on the bar on top of the Scene tab. This will align your camera properly.

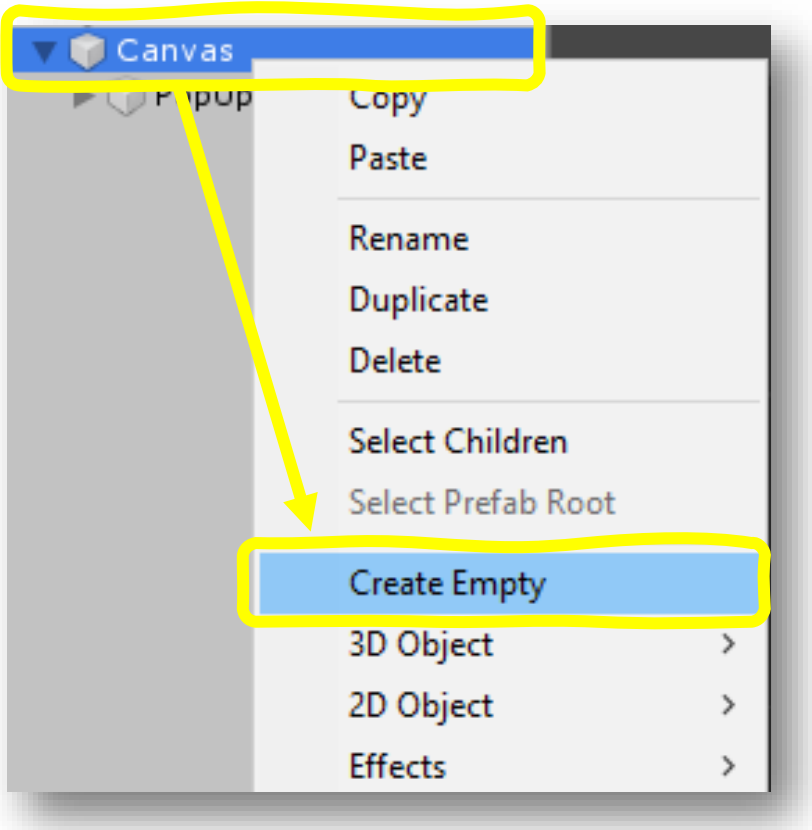


8 Play the game. Did you get hurt and reset? If the player touches the vines, they get sent back to the beginning. We want to add a health system in which the player has only three lives before they get a game over.

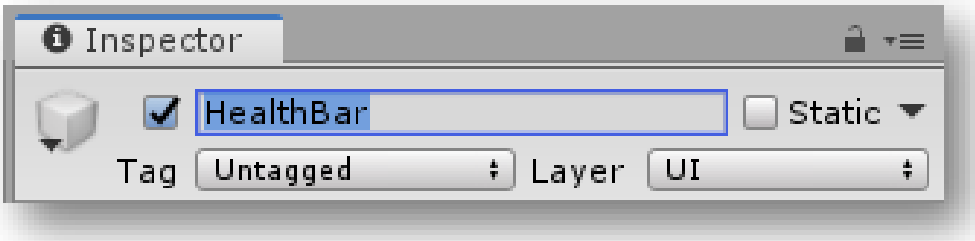
9 In the Level 1 Scene, find the game object named "Canvas".



10 Right click the "Canvas" game object and click "Create Empty".



11 Rename the empty object you just created from "GameObject" to "HealthBar". To do this, left click "GameObject" to select it, then change the name at the top of the inspector tab.

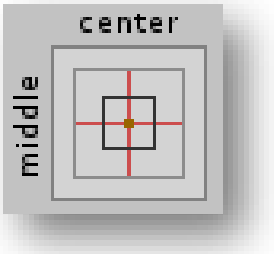


12 Double click on the Canvas Component to Zoom out to see the whole UI canvas.

13 With your new "HealthBar" object selected, look at the "Rect Transform" component in the Inspector. We need to move the health bar to the top left of its parent "Canvas" element.

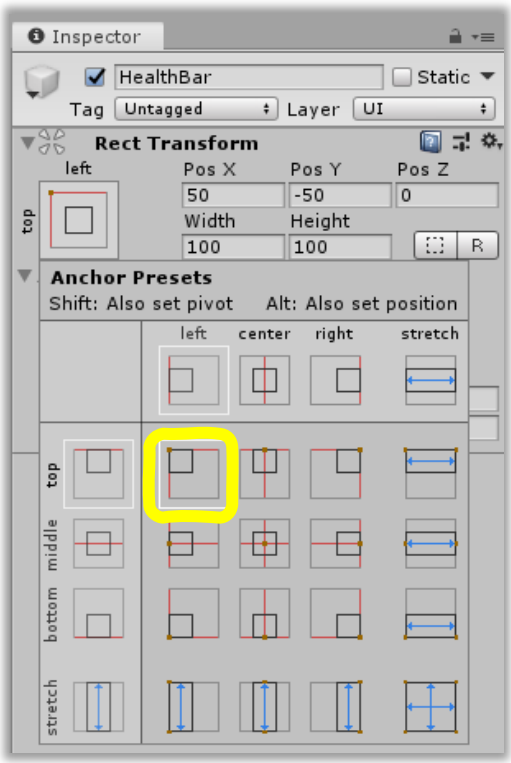
We could manually type in Pos X, Pos Y, and Pos Z values, but Unity can do it for us using Anchoring! Anchoring allows UI elements to general stay in the same spot at difference screen sizes. When we set an anchor to the top left for example, the UI element will try to keep the same distance from the top left corner, even on different screen sizes.

14 Click the box with the red and gray lines to open a new dialogue called "Anchor Presets".



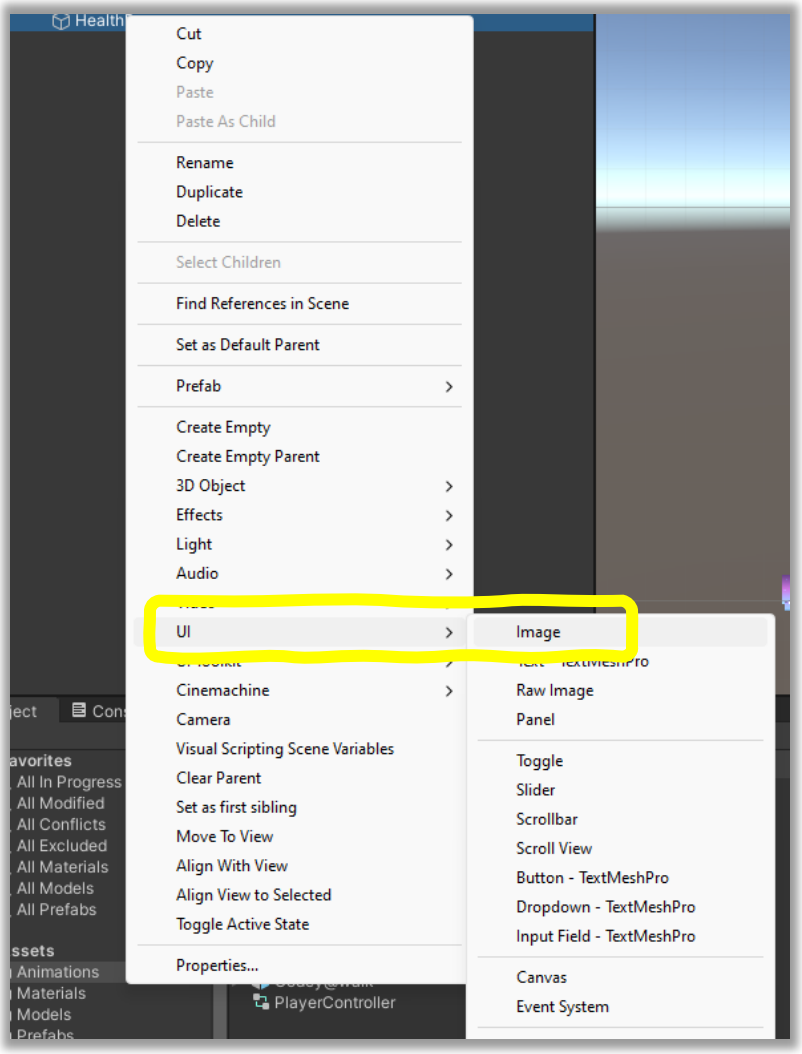
This will let you easily anchor or pin the HealthBar object to the top corner of the Canvas.

Since we want to set the position, hold the **Alt key** while you click the square circled in the image.



15 We will use the HealthBar object to hold the images we need to make in our game UI.

Right click on the HealthBar object in the Scene Hierarchy, go to UI and then click on Image.



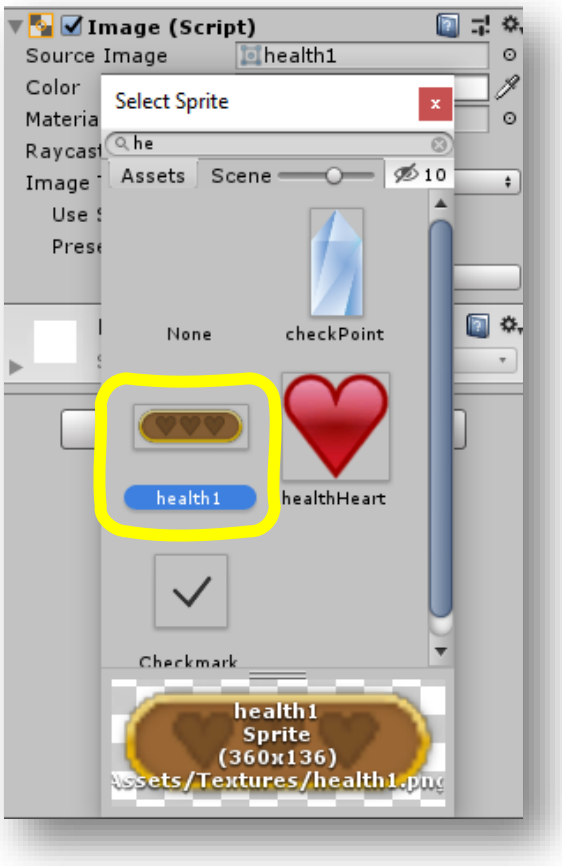
16 Rename this image to "BG" to identify it as the background.

To do this, left click "Image" to select it, then change the name at the top of the inspector tab.

17 In the Inspector tab, we need to set the Source Image property to our health bar background asset.

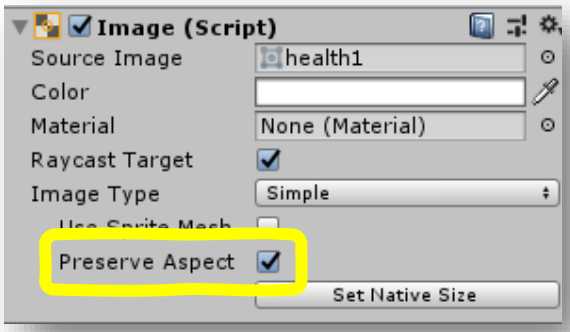
Click the tiny circle ⊙ to open up the sprite selector.

18 Search for “health1”, click the “health1” image, and close out the sprite selector menu.



19 Does your image look a little squished? If so, it’s easy to fix!

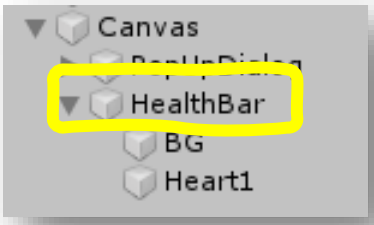
20 Select the BG game object. On the Image (Script) component, enable "Preserve Aspect". This will let us drag the corner of the image to change the size without worrying about squishing or stretching it.



21 A Width of 180 and a Height of 60 look pretty good, but you can make it as big or as small as you want!

22 We also need move it to the top left corner of its parent. We did that earlier with the HealthBar game object! Click the box with the red and gray lines to open a new dialogue called "Anchor Presets". Since we want to set the position, hold the Alt key while you click the square circled in the image.

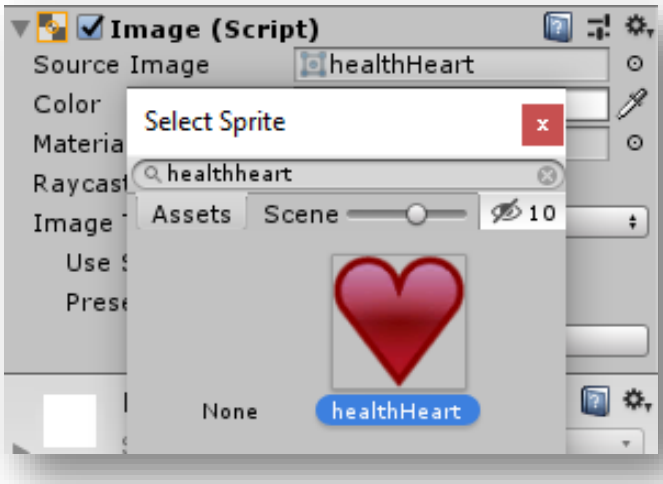
23 Now that we have the background, we need to place the three hearts. Right click the HealthBar object in the Hierarchy and add an Image by finding it in the UI menu.



24 Click this new game object named "Image" and rename it from "Image" to "Heart1" by using the box at the top of the Inspector tab.

25 With the Heart1 game object open it in the inspector, find the Image (Script) component. Open the Source Image's Sprite Selector by clicking on the little circle ⊙.

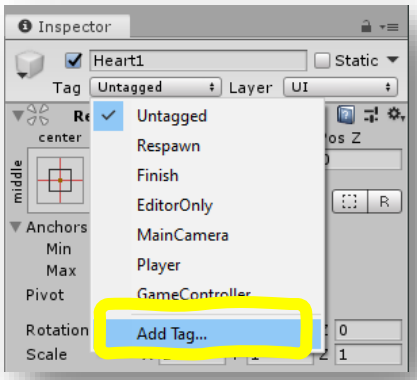
26 Search for and select the image named "healthHeart".



27 Close the Select Sprite dialog.

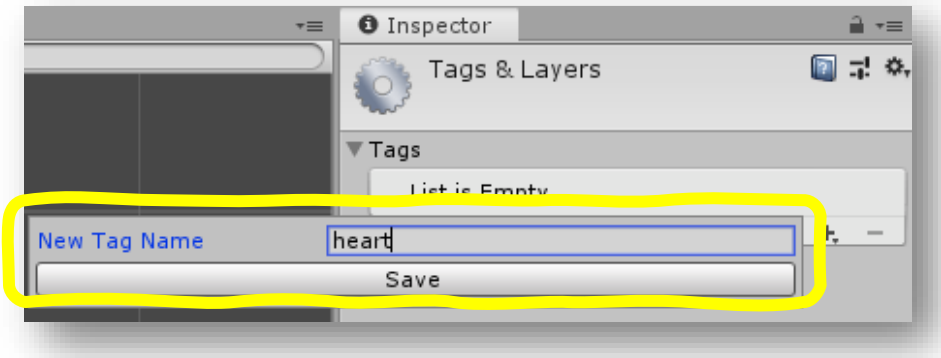
28 Since we are using these images to track our player's health, we want to be able to easily find them. Unity's tagging system is the perfect solution.

29 In the inspector, under the name of the game object is a dropdown to change the object's Tag. We want to create our own, so click on the "Add Tag..." option.

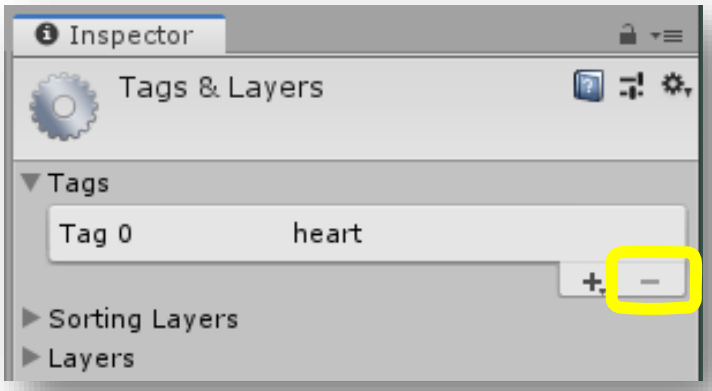


30 This changes the Inspector to the Tags & Layers options.

31 Click the plus sign and type "heart" into the box that pops up. Click the Save button.

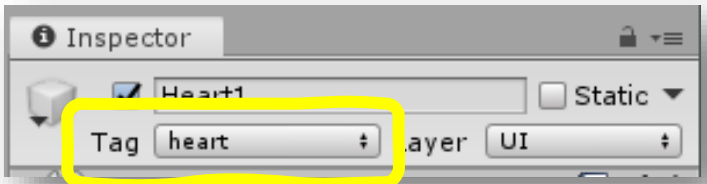


32 You should now see your new tag in the Tags table.
If you did something wrong, you could click on the tag name and then the "-" button to delete it and try again.

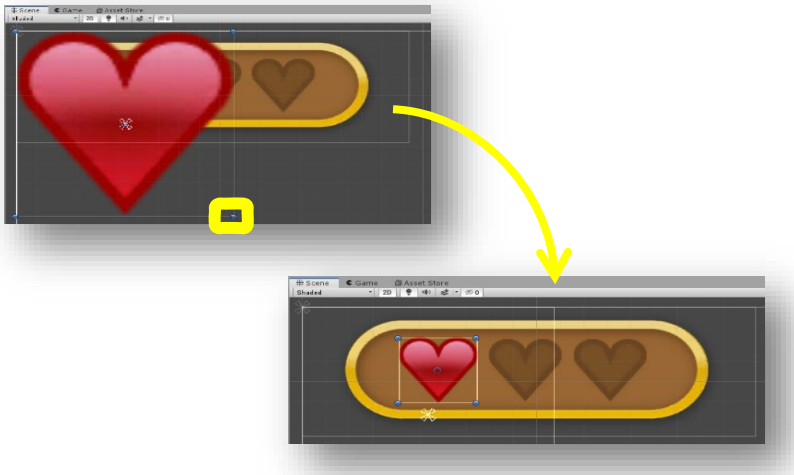


33 Click on the Heart1 game object in the Hierarchy. We now need to set Heart1's tag to the new "heart" tag we just created.

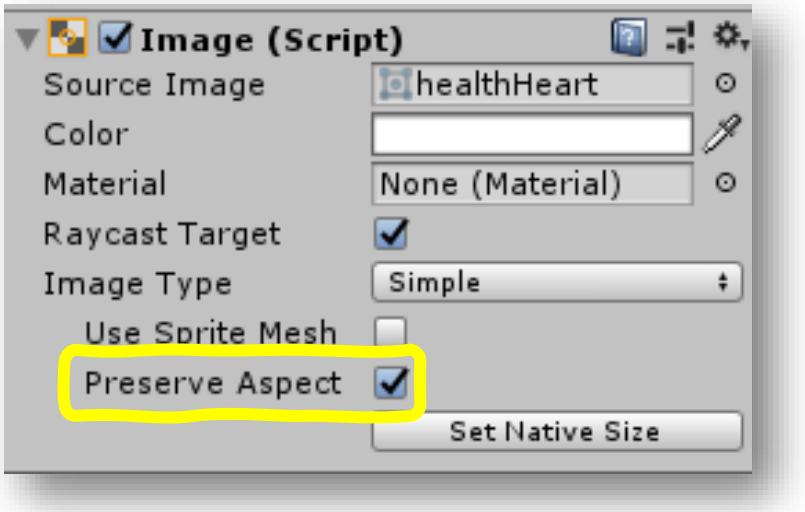
Click on the Tag dropdown and select "heart" from the list.



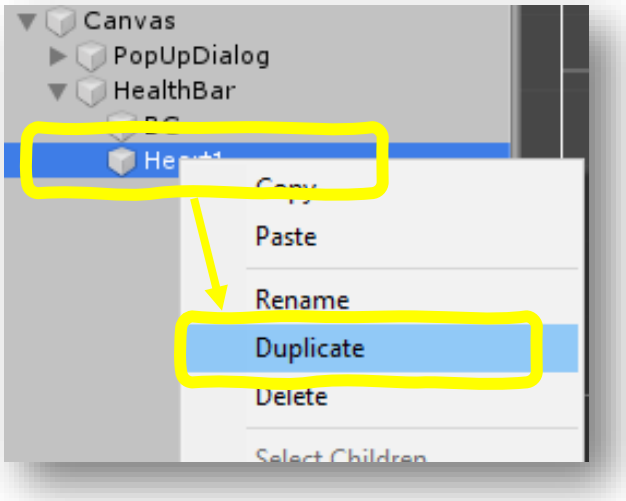
34 In the scene, click on the red heart. Using the blue circles in the corners, change the size and align it with the first dark heart slot.



35 You can enable Preserve Aspect in the Heart1's Image (Script) component to help you get the proportions correct.



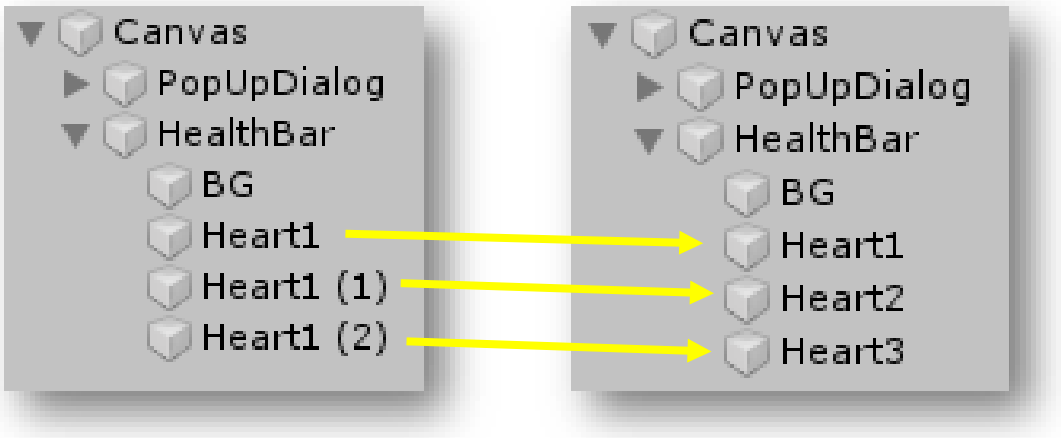
36 We need to place two more hearts. Instead of repeating the process of adding an Image, setting a source image, tagging, and resizing, we can just duplicate Heart1!



37 In the Hierarchy, right click on the Heart1 game object and click Duplicate.

38 Right click Heart1 a second time and click Duplicate again.

39 You should now have Heart1, Heart1 (1), and Heart1 (2).



Rename these new game objects from "Heart1 (1)" and "Heart1 (2)" to "Heart2" and "Heart3".

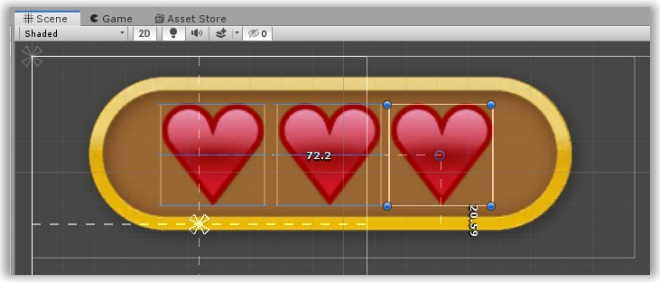
40 If you look in the scene, it doesn't look like anything changed! There are three hearts in the scene, but it looks like just one because we duplicated Heart1. This means that all three hearts have the same exact x and y positions.

41 Click on Heart2 in the Hierarchy. Click inside the highlighted box in the scene and drag it to the right and place it in the second slot of the background.

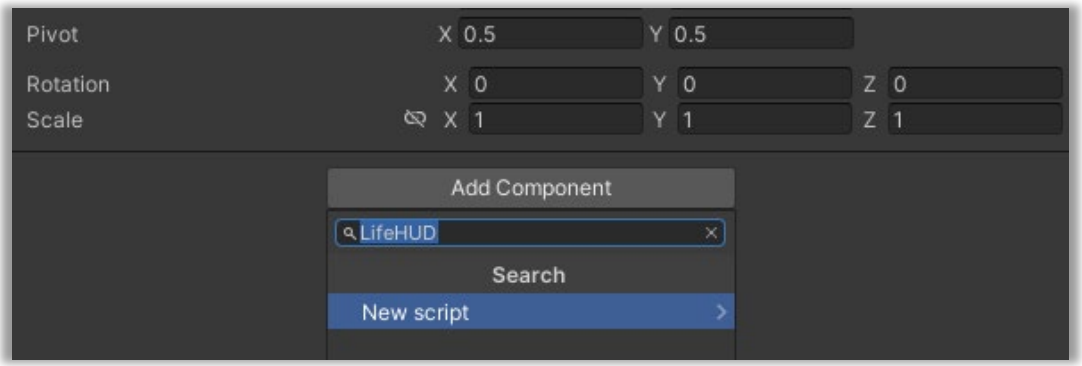
42 Click on Heart3 in the Hierarchy. Click inside the highlighted box in the scene and drag it to the right and place it in the third slot of the background.

43 You might notice that Unity helped you place the hearts in the right place! Unity was able to understand that you wanted to keep the images in a straight line.

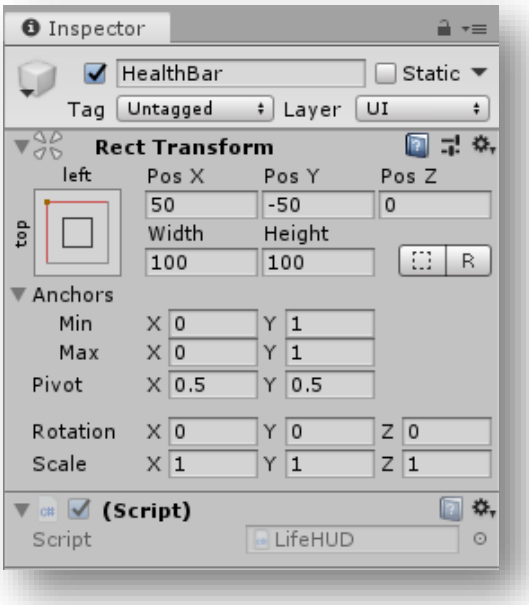
We now have the UI for our health system all set up! Next, we need to write a script to program it!



44 Select the HealthBar object in the Hierarchy to open it in the Inspector. Below the Rect Transform, there should be an Add Component button. Search for "LifeHUD" and click the "New script" button.



45 Double click on LifeHUD in the (Script) component. This will open up the code file in Visual Studio.



46 We need to set up 3 variables at the start of the class: one that points to the hearts in our HealthBar, one that points to the GameManager, and one that states how many lives the player has.

These three variables will go inbetween public class `LifeHUD : MonoBehaviour {` and `void Start()`.

```
public class LifeHUD : MonoBehaviour
{

    // Start is called before the first fr
    0 references
    void Start()
```

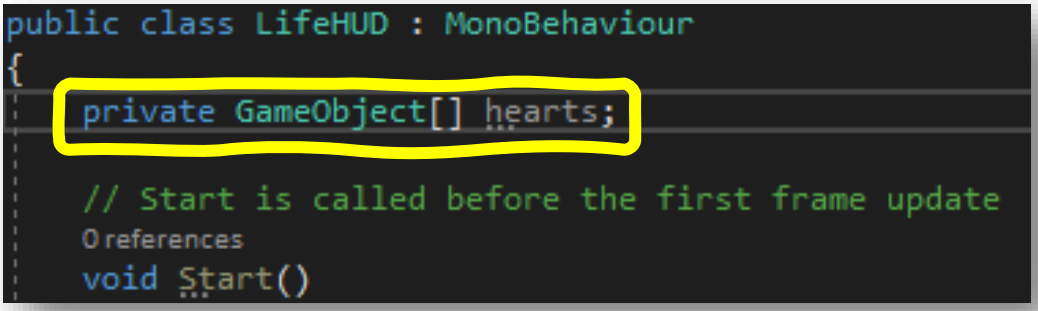
47 The first line to add is

```
private GameObject[] hearts;
```

This will create a private variable named hearts.

This means that no script or game object other than LifeHUD will even know that hearts exist!

The GameObject[] in the middle tells Unity that we want hearts to be an array of GameObjects.

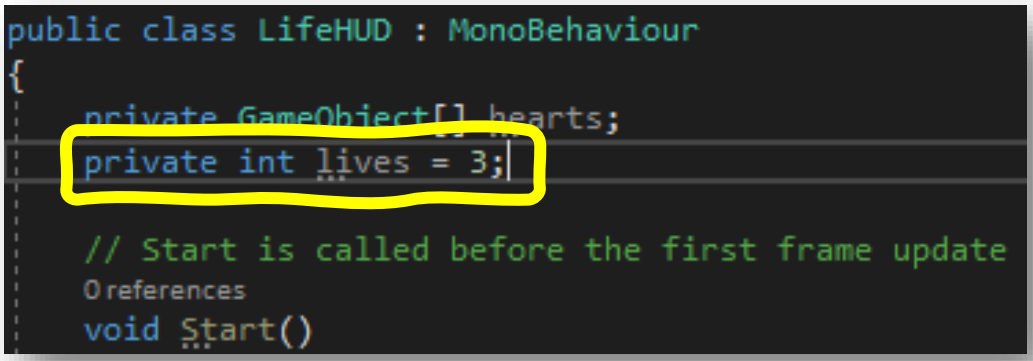


Remember that an array is just a way to store a group of similar objects together.

48 The second line to add is

```
private int lives = 3;
```

This will create a private variable named lives.



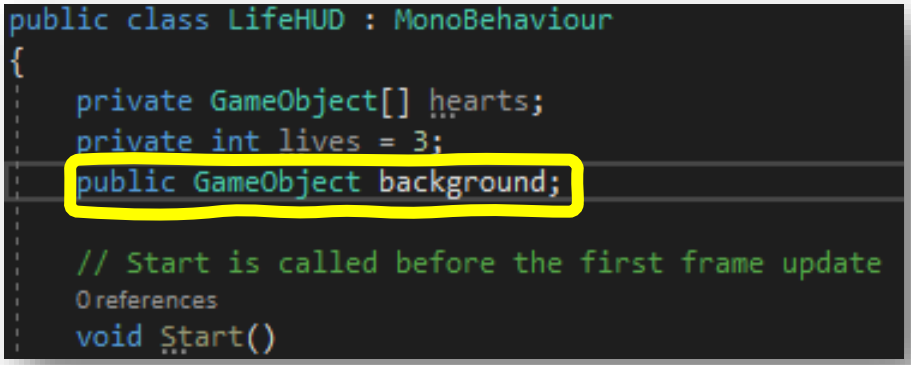
Remember that this means that no script or game object other than LifeHUD will know that the lives variable exists!

The int is telling Unity that we want lives to be an integer (a fancy way of saying a whole number). Finally, we set the value of lives to equal 3. This means that every time we start the game, Codey will have exactly 3 lives.

49 The final line to add is

```
public GameObject background;
```

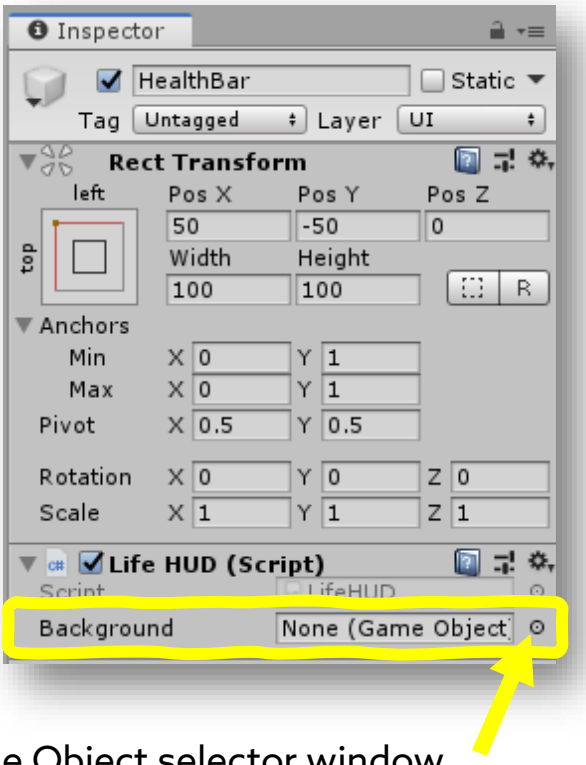
This will create a public variable that is a GameObject named background. By making this variable public, we can use the Inspector to connect the variable to an existing game object in our scene.



50 The game object that we need to connect to our LifeHUD script is simply called “background”.

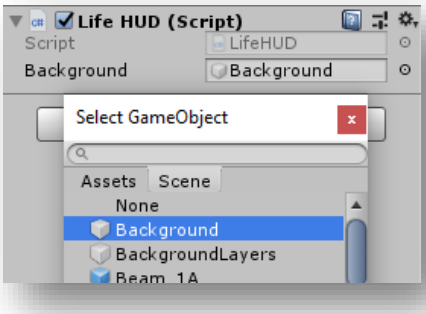
51 Go back to Unity and click on the HealthBar game object in the Hierarchy.

52 Now, in the Inspector look at the Life HUD (Script) component. There should be a new property called Background.



Click the little circle ⊙ to open the Game Object selector window.

53 This window will display all the GameObjects in the scene. We want to find the one named “Background”. Click on it and close the selector window.



Our LifeHUD script is now connected to our Background game object. We will use this to tell the game to show the game over screen.

54 We need to add a new function that will run our code when the entire game starts. The Start function will run when the object first loads into the scene.

55 Once the everything in the game is loaded and the HUD is ready to start, we want to find all of the hearts we set up in an earlier step with the line private GameObject[] hearts;.

We do this by asking the HealthBar GameObject to find all of the objects that have the "heart" tag. We set that up in an earlier step!

56 In the body of the Start function, type
`hearts = GameObject.FindGameObjectsWithTag("heart");`
This will set the hearts variable defined earlier in the code to be equal to all of the GameObjects tagged with "heart" that Unity can find. Because we set up the tags, we know that the value of hearts will be an array of the three red hearts on our HealthBar! This function gets the hearts in the order that they are in the scene from top to bottom, so make sure your hearts are in the right order!

```
public void Start()
{
    hearts = GameObject.FindGameObjectsWithTag("heart");
}
```

57 If your game won't start, look in the console for an error. Do you see something like "Cannot implicity convert type" next to a red stop sign? This message means that it is trying to assign the wrong type to a variable.

If yes, then you are probably using FindGameObjectWithTag instead, which only returns one game object and not an array. Go back to the line of code we just typed and make sure the function you are using is FindGameObjectsWithTag and not FindGameObjectWithTag.

We know we have more than one heart, so we need to find GameObjects, not just one GameObject.

58 Now that we have all of the pieces set up, we need to build out a function for the LifeHUD.

The function we need to make is one that runs whenever the player gets hurt.

While the functions can go in any order, we will place this new function after Start and before Update.

In between those two functions type

```
public void HurtPlayer() {

 }
```

Making sure to leave a blank line between the brackets. We decided to name this function "HurtPlayer" because it describes what the function does.

This function will be responsible for subtracting lives from the player, removing hearts from the HUD, and triggering the Game Over screen.

We want this function to be public because it needs to be called, or run, outside of this script, and the void indicates that it does not return a value.

59 Before we code those three functions, add `Debug.Log("Ouch!");` to let us know when the function is run.

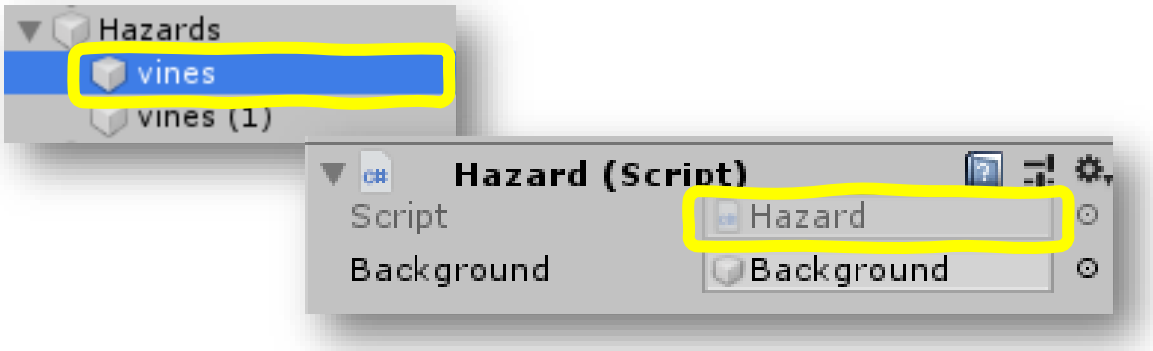
```
public void HurtPlayer()
{
    Debug.Log("Ouch!");
}
```

60 In Unity, open the Console tab 

and play the game. Try to hurt your Ninja by running into the vines.

The console is empty even after running into the vine a hundred times! This is because we never run the HurtPlayer function!

61 What game object should hurt the player? The vines! Find the vines in the Hierarchy and look for the Hazard script in the Inspector and open it.

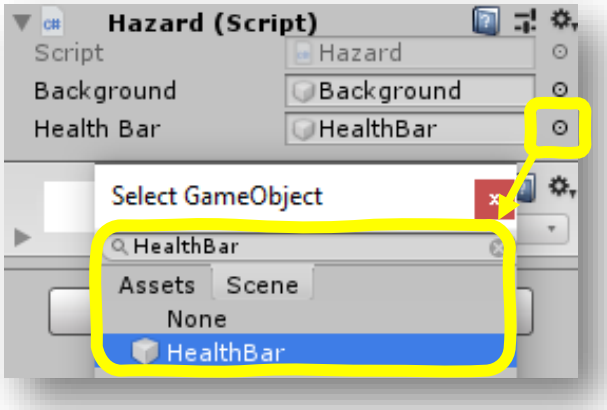


62 Right now, this script uses the background to move the player to a checkpoint whenever the player touches a vine.

Before we can run the PlayerHurt function, we must get a reference to where the PlayerHurt function is defined– the HealthBar object. After public GameObject background; type

```
public
  GameObject
    HealthBar;
```

63 Save the script and go back to Unity. In the Inspector, you should see a new Health Bar property on Hazard (Script). Click the little white circle ⊙ to open the GameObject selector, search for "HealthBar", select the HealthBar result, and close the Selector window.



64 Since we have two vines in the scene, we need to repeat the process. Select "vines (1)" to open this game object in the inspector.

Click the little white circle ⊙ to open the GameObject selector, search for "HealthBar", select the HealthBar result, and close the Selector window.

65 Now that we have the HealthBar game object connected to the Hazard script, we can call our HurtPlayer function.

Since we want to call this only when the player touches vines, we need to write our line of code inside of the OnTriggerEnter function.

```
private void OnTriggerEnter(Collider other)
{
    background.GetComponent<GameManager>().moveToCheckPoint();

}
```

Remember that OnTriggerEnter is a special function that will execute code whenever the Ninja collides with the vines because the vines' collider has the IsTrigger property checked.

66 On the line after `background.GetComponent<GameManager>().moveToCheckPoint();` type `HealthBar.GetComponent<LifeHUD>().HurtPlayer();`.

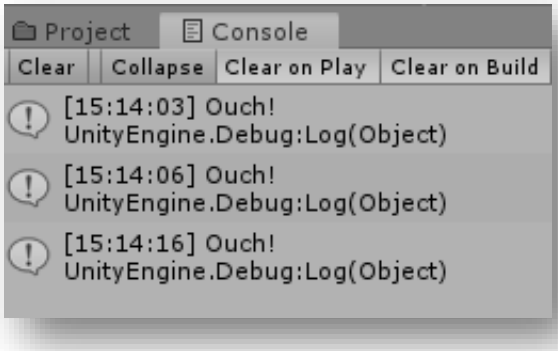
Since the HurtPlayer function lives inside the LifeHUD script which is attached to the HealthBar game object, we have to chain a few commands together.

First we ask the HealthBar variable for the LifeHUD script component by using GetComponent<LifeHUD>(). We then need to ask that script to run the HurtPlayer function with HurtPlayer().

```
private void OnTriggerEnter(Collider other)
{
    background.GetComponent<GameManager>().moveToCheckPoint();
    HealthBar.GetComponent<LifeHUD>().HurtPlayer();
}
```

67 **Play** the game and run into the vines. Does something show up in the console now? Make sure to check both sets of vines.

You should see the "Ouch!" message appear after the Ninja runs into the vines. This means that the vines are connected to the HurtPlayer function and we can continue adding functionality!



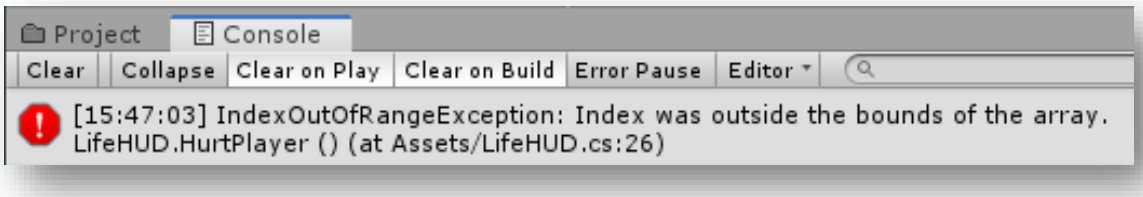
68 Open up the LifeHUD script. Inside the HurtPlayer function, after Debug.Log("Ouch!"); add `lives -= 1;` to subtract a life each time the player gets hurt. Rember that the -= operator reduces the value of the variable by one.

```
public void HurtPlayer()
{
    Debug.Log("Ouch!");
    lives -= 1;
}
```

69 Now that we are tracking when we lose lives, we need to update the hearts in the HealthBar user interface. After each time we remove a life, we need to deactivate the appropriate heart in the UI. We do this by using the hearts array. On the line after lives -= 1 type `hearts[lives]SetActive(false);` to disable the appropriate heart in the UI.

```
public void HurtPlayer()
{
    Debug.Log("Ouch!");
    lives -= 1;
    hearts[lives].SetActive(false);
}
```

70



Play the game and run the Ninja into the vines. It works! What happens when you run into the vines when you are out of hearts? The game should stop and show a game over screen, but that doesn't happen. Instead the Ninja gets sent to the checkpoint and we get an error in the console since the number of lives is less than 0! That's okay, we can fix it!

71 We need to catch when the player has run out of lives. On the line after hearts[lives]SetActive(false);
type

```
if (lives == 0) {

}
```

Make sure to put a blank line inbetween the brackets.

```
public void HurtPlayer()
{
    Debug.Log("Ouch!");
    lives -= 1;
    hearts[lives].SetActive(false);
    if (lives == 0)
    {

    }
}
```

72 The game over screen should be shown only if the player has exactly 0 lives left, so inside the if statement type

```
background.GetComponent<GameManager>().GameOver();.
```

```
if (lives == 0)
{
    background.GetComponent<GameManager>().GameOver();
}
```

Since the GameOver function lives inside the GameManager script which is attached to the background game object, we have to chain a few commands together.

First we ask the background variable for the GameManager script component by using GetComponent<GameManager>().

We then need to ask that script to run the GameOver function with GameOver().

73 **Play** the game and run the Ninja into the vines until there are no more lives left. The Game Over screen pops up!