**Game Development Notes**

**1st Game Notes (Terminal Hacker):**

For C# Basics –

* Array:

1. Array is a storage quantity to store multiple values in a single variable.
2. Above is the syntax to declare an array
3. It is declared by using ‘[]’.
4. The compiler counts the values in an array with as index number, in which counting starts from 0.
5. Here “author” is 0, “study” is 1, and etc.
6. Syntax to use the value stored in the array



1. And to get any value randomly we use a command caller Random.vector3. Syntax for it:



1. “. Length” is a command used in arrays. It is a command which shows the total number of values in the array it is used with

* Enum:

1. Enum is a keyword used to make a variable type which defines that the variable declared using enum will only have the values provided in that particular enum.



1. In the above figure we have declared a variable type called Screen. And the variable which will have Screen type can only have the values in the bracket
2. In the figure below Screen is a variable type and CurrentScreen is the Variable name which will only have values in the Bracket 
3. In the image below we have assigned the value of variable declared above. Syntax is below



* Member Variable:

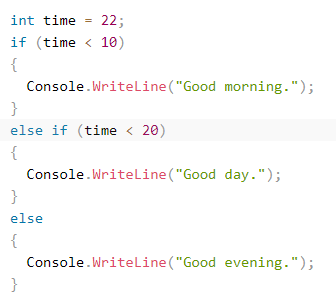
1. It is a variable which is not declared in any method but is declared in the class but not in any method.

1. It is declared like this in the figure above.
2. And it is always declared outside methods because we have to use it in multiple methods and give different values.

* Print Statement: It is a command to print wanted things in the console

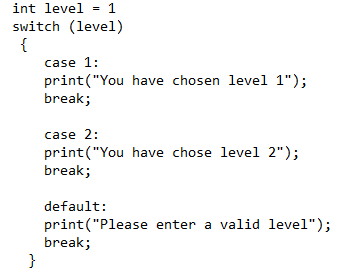
1. Syntax – 
2. To get output in multiple lines with a single print statement we use @ before quotation.
3. Syntax – print (@” hello

 world”)

* Conditional Programming –
* **Using “If” Statement -**

1. To use if statement we provide a condition with “if” “else if” and “else’.
2. It is used like this in figure.

* **Using “switch” Statement –**

1. To use switch statement, we enter a variable name with “switch”, then case which will have the value of variable declared with switch, and then the output we want. Then default if something rather than all the cases.
2. Be careful to place break after every case and default statement.

* Sign Operators –

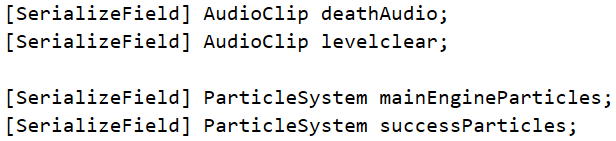
1. “=” is used to assign value
2. “==” is in Boolean value or to show that a variable is equal to the thing on R.H.S
3. “!” this is for not
4. “! =” this is for not equal to
5. “\*” for multiplication
6. “/” for division (gives quotient)
7. “%” it gives remainder
8. “+” for addition
9. “– “for subtraction
10. “||” means OR

* Return Keyword – 'return' is a command telling the method not to give output further than that if the condition follows. It can also return an optional value. If the method is of void type, the return statement won’t work. It can also be used to command to return a value, a variable, or a constant.

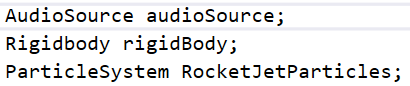
**2nd Game Notes (Mr. Rocket)**

For Unity based C# and Unity Engine Basics –

* “[SerializeField]” keyword –

1. The [SerializeField] keyword is used to mark fields as serializable in the Unity Engine, so that Unity can save and load those values in the Script component on the engine
2. It is used with Member Variables and Variable types provided by Unity Engine.
3. Syntax to use with member variables - 
4. Syntax to use with Variable types provided by Unity Engine –

* Variable type provided by Unity Engine –

1. These are the variable type provided by unity engine, under the package -
2. The variable types provided by Unity Engine are actually Game Components.
3. We use the Variable type provided by Unity Engine by declaring them as a member variable.
4. But to command the engine to use the the Unity Specific Variable we use “GetComponent<>();” command.
5. It commands the Unity Editor to refers to a component which is already added to the game object. And takes the value for the variable form the respective Component to which the desired file is given. It takes value from the component when game is playing.
6. It returns the an already existing component added to the game object in the Unity Editor. The variable name given of Component stores the value which we provide in the Unity Editor. For Example if the variable type is “AudioSource” so the variable of its type will store an audio clip.
7. There are more Unity specific variable type used in this game AudioClip and ParticleSystem. The name after variable type is the name of the variable type.





1. If we use the above two with ‘SerializeField’ they are added to the ‘C# Script’ component and we can insert file according to the type file, they store. For Example:
   * ‘AudioClip’ stores audio files and ‘ParticleSystem’ Store Unity particle effect assets
2. Syntax to play the ‘AudioClip’ and ‘ParticleSystem’ -



1. For audio clip we can also use “levelclear. Play”
2. And for stopping it we can use “. Stop”
3. ‘. PlayOneShot’ commands to play the audio only once
4. ‘levelclear’ is the name of variable type ‘AudioClip’ and ‘successParticles’ is the name of variable type ‘ParticleSystem’

* Transform Component –

1. “Transform” is a component of a game object in which we can determine the size, rotation, and position of the game object
2. It has three parts –
   * Position – to use it we type “transform. Position”. Here we are determining that there is a Transform component which has a section named “Position” for position of the game object. We can use “transform. Position” for value assigning of position of the X, Y, Z axes, for movement of the game object or for assigning the value of a variable of type “Vector3”.
   * Rotate - to use it we type “transform. Rotate”. Here we are determining that there is a Transform component which has a section named “Rotation” for rotation of the game object. We can use “transform. Rotate” for value assigning of rotation of X, Y, Z axes, for rotation of the game object or for assigning the value of a variable of type “Vector3”.
   * Scale- // Not Taught Yet.

* To respond on an input of user

1. For the case above we use "(Input.GetKey(KeyCode.Input))" (at place of ‘Input’ we use the code for a Specific key).
2. "Input. GetKey" will repeatedly return the value which the user holds down the specified key.
3. 'KeyCode.' is a command for specifying the key pressed.
4. Syntax –

* Vector3 –

1. It is used to represent the position and movement of a Game Object in all three axes (X, Y, Z).
2. It is used to determine the position of Game Object which comes under ‘Transform’ component. As when an object will move it will surely change its position.
3. It holds value of the position of Game Object in X, Y and Z axis. And also returns the same if used as a method.

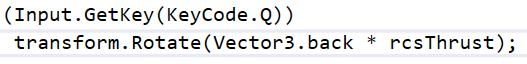
* For Movement of Game Object –

1. "rigidBody.AddRelativeForce(Vector3.direction);" syntax for making an object or player move in x, y, z direction.
2. "rigidBody” is the variable name of the variable type provided by Unity which is RigidBody and it is also a Game Component (Cleared Above).
3. “AddRelativeForce ()” Adds a force to the rigidbody relative to its coordinate system. Wakes up the Rigidbody by default. If the force size is zero, then the Rigidbody will not be woken up.
4. “Vector3.direction” tells the game object to move in x, y, z direction.
5. Syntax to move the object upwards –
6. Can be used with all directions. (above is just an example)

* For Movement of Game Object on an Input –

1. It has a Syntax –
2. If we want the game object to respond on multiple inputs, we can use if statement.

* For Rotation of Game Object –

1. Syntax –
2. “transform” is the Game Component in the Unity Engine and “Rotate” is one part of the Transform component.
3. So the second line in the syntax commands the game object to rotate the game object in the direction specified.
4. “rcsThrust” is the variable which has the value of force which will be applied on the game object for rotation.

* For adding audio –

1. For adding audio to game, first we add a component, then we attach an audio file to the ‘AudioSource’ component.
2. But the ‘AudioSource’ can also be used as a variable.



1. And we can make it respond on an input by using ‘GetComponent<>’ keyword.
2. And to make it respond on an input we use
3. Or to directly play it we use the 2nd line in figure above.
4. To stop we use -
5. And to stop it in a condition we use the condition statement and then the line in the figure above.

* For response on collision –

1. This is a Normal Collision Case where Trigger is Disabled.
2. Condition – the game Object should consist of “Colliders” and “RigidBody” components or else the collision code won’t work
3. For response on collision the Unity Engine Package provides a specific method. It should be exactly like below.

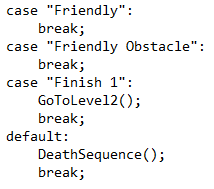


1. 'Collison' is a variable type provided by unity for collision of any game object with value – ‘collision’.

And to make a game object respond on a collision we ‘Tag’ the game object. And use switch statement for response on collision with the game object.

1. First we declare that it is a switch statement. Then use the syntax in the switch statement’s brackets which is used for collision with the tagged object. That syntax is

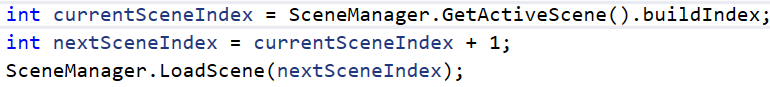


1. Full syntax with switch –
2. Then we use ‘case’ with the tag given to a game object with which the player will collide. Then we type the response if the player collides the tagged game object. Then we use ‘break’. Syntax –
3. The word in double quote is the tag for game object.
4. ‘GoToLevel2’ and ‘DeathSequence’ are methods which have response that we want in each case
5. No response in the case means nothing will happen.

* Invoke Keyword –

1. Invoke Keyword is used to apply a method after the given time.
2. The time should be in seconds and should be a float value.
3. To use Invoke first we declare the statement as invoke.
4. Then in bracket we type the method we want to invoke, it should be in double quote like this – (“MethodName”)
5. Then time quantity of time we want to wait for the method to apply. Like this - (“MethodName”, time)
6. Example - Invoke (“MethodName”, 2f);
7. Syntax-
8. Instead of typing the time we can declare a variable with the time quantity we want write the name of that the variable.

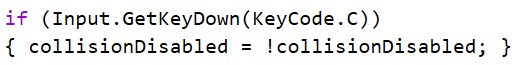
* Scene Management –

1. It is used to move from one to another, it can either be on an input or by collision with some object.
2. The package used for changing scenes is –
3. “UnityEngine.SceneManagement” is a package for management of scene in the whole game.
4. To go from one scene to another scene we first add all the scenes to build scene. And the build scene counts the scene index from 0.
5. Then we give a command which loads the scene according to the number given in the bracket. That command is “SceneManager.LoadScene()”. (The bracket with load scene will have the scene number which is to be loaded)
6. ‘LoadScene ()’ is a command used to load the given scene number, which is given in its bracket.
7. Syntax –
8. Code to load the next Scene without using the above command again and again –
   * Syntax for it –
9. Explanation of the syntax above –
   * First we declared a variable called ‘currentSceneIndex’, which is equal to the active scene.
   * Then we used the keyword(SceneManager) which is used for telling the machine to use the ‘SceneManagement’ package.
   * Then we gave the value of the variable declared as ‘GetActiveScene ()’, which is used to get the currently active Scene or the currently displayed scene.
   * Then we used the ‘buildIndex’ keyword to return the index of the scene in the build settings.
   * Then we declared another variable ‘NextSceneIndex’ which is equal to the ‘CurrentSceneIndex + 1’ means it will be equal to the current scene displayed + 1 means the scene next to the scene displayed
   * And then we commanded the machine to load the ‘NextSceneindex’ using ‘LoadScene ()’ command.
   * So it will load the scene next to the current displayed scene.

* Code to Toggle a Function –

1. First we declared a member variable of type bool and if we want to make it available by default then we have to declare it as true, and if we want to make it available on an input then we have to set it as false by default. (Here we have taken it as false.)



1. Then we use if statement so that what we want can be toggled on an input.
2. Syntax –
3. Explanation –

* If input is C, then ‘collisionDisabled’ which is false then, it will be equal to ‘! collisionDisabled’ (collision not disabled) and if something is false by default then it’s not version will be true.
* And collision will be disabled when the variable name will be added in the ‘OnCollisionEnter’ method by using an if statement. Syntax to use –
* Explanation of the code above – Here we are saying if the state of player is not alive or ‘collisionDisabled’ is true then ‘return’ (not apply further functions of that method).
* Mathf.Epsilon Keyword –

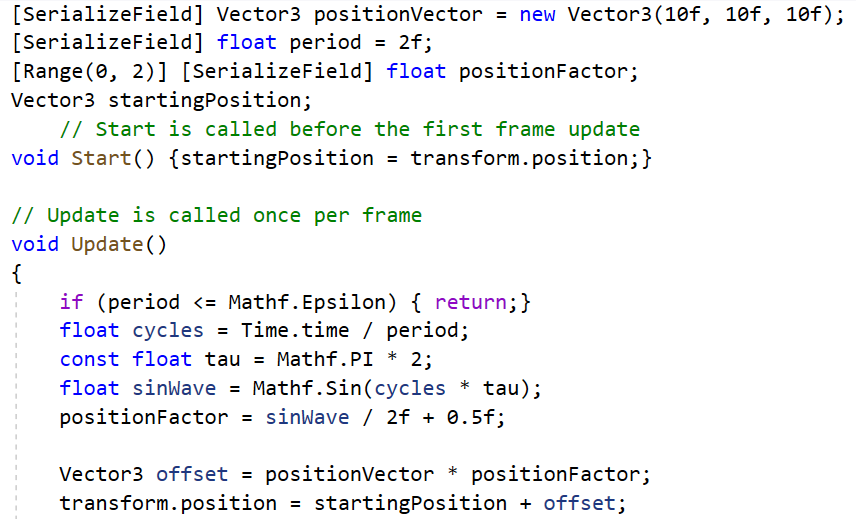
1. It is a tiny floating point value.
2. The smallest value that a float can have different from zero. Which is approximately 0 but no zero like – 0.000000000001.

* Time.time Keyword –

1. This is the time in seconds since the start of the game.
2. This value is undefined when game is awake and it gets started to count the time when game starts. This value does not update if the Editor is paused.

* Mathf.PI – It is the simple pie used in geometry and is the syntax to represent and use it in programming.
* Mathf.Sin – It give out the sin value of the number or variable.
* For Automatic Movement of a Game Object –

1. Code to make an object move automatically is below



1. ‘positionVector’ is a variable of type ‘Vector3’. Because we are defining its X, Y, Z position value, thus we are using ‘Vector3’.
2. ‘period’ is a member variable of type float with “[SerializeField]” keyword so we can change its value in the unity editor, to test different values. The value of ‘period’ is the time taken to travel the given distance, to travel the given distance in X, Y, Z axes, or to complete one oscillation by the game object.
3. ‘positionFactor’ is a member variable of type float. That “[Range(0,2)]” is to get the values in a bar which can be slided with initial = 0, and final point will be 2. (So we use Range () to represent the values in a bar, and the can be from 0 to infinite)
4. We declared another member variable named “startingPosition” of type “Vector3”, as we have to represent the position of game object using a variable thus the variable type is “Vector3”
5. Then we say that “startingPosition” is equal to “transform.Position” (“Transform” is a component of a game object in which we can determine the size, rotation, and position of the game object, “Position” is a part of “Transform” Component which comes under “Transform” Component. And by writing “transform. Position”, we are determining that the value of the variable is the default position of the game object on the X, Y, Z axes which is given in the engine or we can say that the value of the variable will be taken form the “Position” section which will already have the values of X, Y, Z coordinates under “Transform” component.)
6. Then we say “if (period <= Mathf.Epsilon)” return back which means that, it will won’t apply the functions ahead in that method if the condition follows. (this means if period = 0 so then return because value smaller than Mathf.Epsilon will most probably be 0)
7. Then we declare a float variable named “cycles” which is equal to “time. Time/ period”.
8. Then we declare a constant float variable “tau” which is equal to “Mathf.PI \* 2”, which is 3.14\*2 = 6.28. (tau is a quantity like pie which is equal to 2\*pie thus we have declared it constant)
9. Then we declare a float variable “sinWave” which is equal to sin of cycles \* tau (Mathf.Sin(cycles\*tau)). The ‘Mathf.Sin’ gives the sin value of ‘cycles \* tau’. And it also makes the game object move back and forth if it is used with a variable type of ‘Vector3’ because ‘Vector3’ is responsible for movement and rotation, so either the game object will rotate back and forth or it will move back and forth, it depends that ‘vector3’ is used with rotation part in transform component or the position part. And the game object moves back and forth because a sin wave moves like that.
10. Then we assign the value of variable “positionFactor” which is equal to ‘sinWave/2+0.5’ (variable declared in above line). Here we conclude that if the value of the “positionFactor” will be less than 2, then the point on the bar will not cover the total distance of bar in the Unity Engine, or if it is 2 then the point on the bar will cover the whole distance of the bar.
11. Now we declare a variable of type ‘Vector3’ named “offset” which commands the game object to move, and value of the offset tells the game object how much to move, in this case which is “positionVector\*positionFactor”.
12. So we concluded that “offset” variable of type “vector3” command the game object to move and its value tells the game object how much distance to move and the “Mathf.Sin” operator tells the game object how to move, that is, Back and Forth.
13. And then we say that the “transform.Position” which means the value of X, Y, Z axes in the engine will be equal to the default position value (startingPosition) + offset (“positionVector\*positionFactor”).
14. Transform.Position consist of the values of X, Y, Z, which is equal to the default position coordinates + units moved in each axes (offset).

**Third Game (Argon Assault)**

* **Time.deltaTime Keyword –**

1. It is a key word used to make the game smooth for all low and high end PCs. As by default the unity engine plays the game on FPS (Frames Per Second), but “Time.deltaTime” makes the movement slow and smooth so that all systems whether low or high configuration the performance will be same in all systems.
2. The default FPS system loads frames per second but the “Time.deltaTime” loads the frames according to time in result which slows the movement and if the input key not pressed for longer it will only cover a little bit of the given distance given in the code.
3. The FPS system loads everything faster and if the system gives high FPS everything will load so fast that it will look like the game object is teleporting, and if the system is giving low FPS then the movement will look like the game object it moving but it won’t be smooth at all, whereas the “Time.deltaTime” loads everything in small movement, which in result makes the game object moves slow and looks it is moving and not teleporting.
4. It is used to make the movement of game object free of FPS based movement.
5. If we want to increase the speed using the “Time.deltaTime” we can increase the units we want move on the input, in result the speed will increase.
6. Syntax –
7. We multiply the ‘Time.deltaTime’ key word with the value we want the game object to move so it is implemented in the program

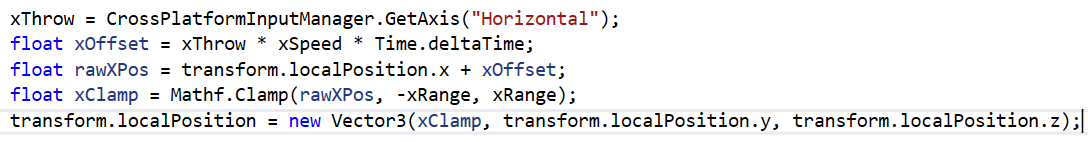
* **Tooltip Keyword** –

1. Tooltip is a keyword used to declare the unit of the value of that variable. Units like metre, m/s, second, kilometre and etc.
2. It is used with a variable to give a unit to the value of the variable.
3. Syntax -



1. Here we have given the m/s unit to the value of “xSpeed”

* **Unity CrossPlatform Input Package** –

1. It is a package provided by unity standard assets to make game for all platforms and get input without keycode by setting the input in the UnityEngine
2. By using it we can set the input in the UnityEngine for all platforms like PC, Mac, Play Station, and mobile.
3. We declare it by –
4. Syntax to respond on an input by using “CrossPlatformInput” package -
5. We just have to declare that in which direction we want to move the game object by assigning it as a value of a variable.
6. Here “Horizontal” is the term used to determine X axis, the same way we can use Vertical for Y axis.
7. And after declaring the value of variable as “CrossPlatformInputManager.GetAxis("Horizontal")” we can set the input in the Unity Editor.
8. And in the Case here the Game Object will move in X axis by the input set in Unity Editor with the following syntax –
9. In the figure above we have given values to different variables, and after getting the final desired value we have given that value of game object in place of X axis in the “new Vector3” as we want the game object to move in X axis, in result the game object will move according to the value of the variable given the place of the axis in which we want the game object to move.
10. **Note –** **‘**Mathf.Clamp’ and ‘transform.localPosition’ explained below.

* **“transform. localPosition” Keyword** –

1. It is the default position of the game object in the Unity Editor.
2. If we want to make a game object to move by using it the syntax is:
3. In the figure above we are saying that local position of game object is equal to a new Vector3 value. In the bracket the first one is for X axis, then Y, and then Z, so in this case we want the game object to move in the X axis, thus we have given the value of position of X axis and we want the object to stay on that place in Y and Z axis.

* **Mathf.Clamp Keyword** –

1. Mathf.Clamp determines the range of movement of the game object. The game object stops moving when the game object reaches the clamp range.
2. For example, the range is 5 units and -5 units in x axis, so when the game object completes travelling 5 units or -5 units in X axis it will stop moving further after that. Or we can give use 2 different variable to give value for the range of clamp
3. Syntax –
4. Here we have used it as a value of variable otherwise it is just like that in RHS.
5. We have to use three values in its first the value we have provided to the game object to move and then the both the range value in which the object should move, it can be -ve and +ve, -ve and -ve, +ve and +ve.

* **DontDestroyOnLoad Keyword** –

1. "DontDestroyOnLoad ()" is a keyword used for not destroying a game object in the next scene in which it is not used. So it preserves the object of the previous scene in the next scene so that it is still applicable and it continues in the next scene.
2. Syntax –
3. Here we have used a variable named "gameObject" of type "GameOject", which is already defined in the UnityEngine Package, so we don’t need to define it in the code.
4. The Game Object here will be the game object to which the script is attached.

* **Quaternion Keyword** –

1. "Quaternion" is a keyword used to represent rotation of object in the X, Y, Z axes. It is the same as "Vector3" which we use to represent position value of object in X, Y, Z axis.
2. Usage –

* **Trigger Collider** –

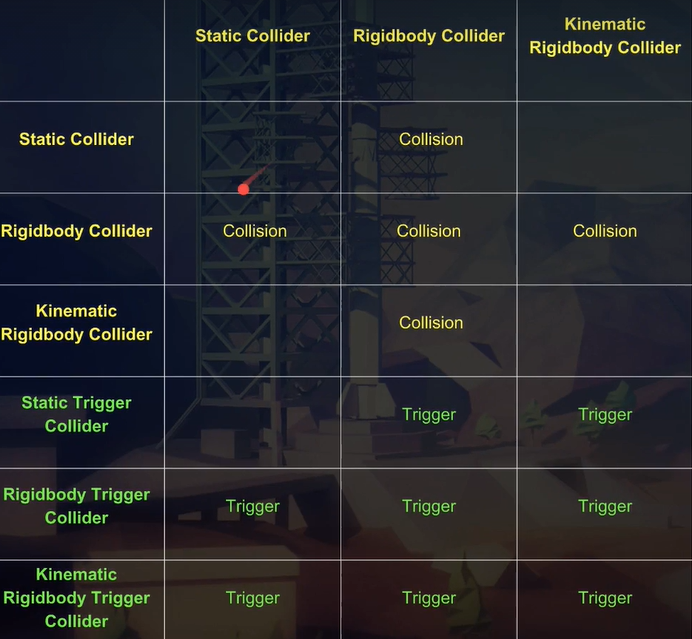
1. When a GameObject collides with another GameObject, Unity calls “OnTriggerEnter ()” method. It the same as OnCollisionEnter, but it is applied on the condition if trigger is enabled in Unity Editor.
2. Condition to apply it - To use the trigger collider Both GameObjects must contain a Collider component. One GameObject must have Trigger enabled, and contain a Rigidbody.
3. Method to use it –

* **All type of Normal Collision and Trigger Collision** –

1. **Normal Collision –** It is the Normal collision case where we use Method –

* The Static Collider – It is a simple collider without Trigger enabled. In it the game object does not have “RigidBody” Component (Which gives Physics to the game Object), and just has the Collider Component. In it Normal collision will occur.
* RigidBody Collider – It is a simple Collider without Trigger enabled. In it the Game Object has the RigidBody component as well as the collider Component. In it Kinematic is disabled. In it Normal collision will occur.
* Kinematic RigidBody Collider – It is a simple Collider without Trigger enabled. In it the Game Object has RigidBody Component, under which the Kinematic is enabled. In it Normal collision will occur.

1. **Trigger Collison –** It is the Trigger collision case where we use Method –

* Static Trigger Collider – It is a simple collider with Trigger enabled. In it the game object does not have “RigidBody” Component and just has the Collider Component. In it Trigger collision will occur.
* RigidBody Trigger Collider – It is a simple Collider with Trigger enabled. In it the Game Object has the RigidBody component as well as the collider Component. In it Kinematic is disabled. In it Trigger collision will occur.
* Kinematic RigidBody Trigger Collider – It is a simple Collider without Trigger enabled. In it the Game Object has RigidBody Component, under which the Kinematic is enabled. In it Trigger collision will occur.
* Table for above cases –
* GameObject Variable Type

1. GameObject is the variable type. By using it we can add a GameObject to the Script in the UnityEditor
2. The variable of type "GameObject". Here – deathFX as variable name stores the game object in it, which is the value of that variable and we provide it in the UnityEditor.
3. Syntax –

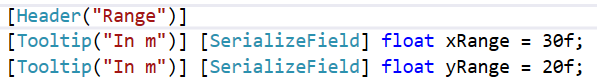
* SetActive Keyword –

1. “SetActive” is a keyword used to make a GameObject work. It makes the game object active that means it will make the GameObject working at the runtime.
2. Syntax –
3. Explanation of Syntax above – Variable name of GameObject type will store the value and that value is the game object which we will provide in the unity editor so we say “variableName.SetActive(true);”
4. It is almost like “. Play” command if with true. It requires Boolean value it can be either true of false. If true then it will work and if false, it won’t work.
5. It only works if the game object is deactivated by default.

* SendMessage Keyword –

1. "SendMessage" is Keyword use to call a method of another script attached to that game object.
2. Syntax –
3. Here "OnPlayerDeath" is a method from another script.

* “Header” Keyword –

1. "Header" is a keyword used to Child the member variable into a topic, which we see in the Unity Editor.
2. Syntax –
3. The member variables below will get childed to the topic “Range” in the UnityEditor.

* Destroy Keyword –

1. Destroy is a keyword used make the game object not working or we can say it destroys the game object.
2. Syntax –
3. We have used a variable with name "gameObject" of type "GameOject", which is already defined in the UnityEngine Package, so we dont need to define it. The gameObject variable here is the Game Object to which the script is attached

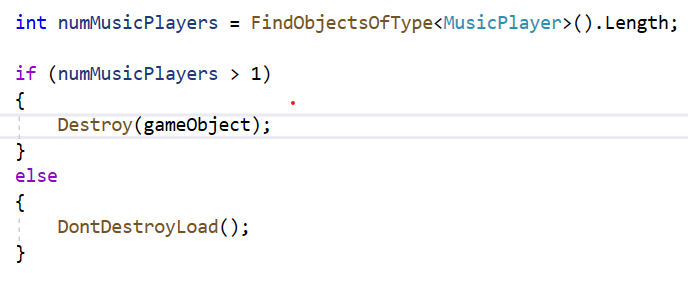
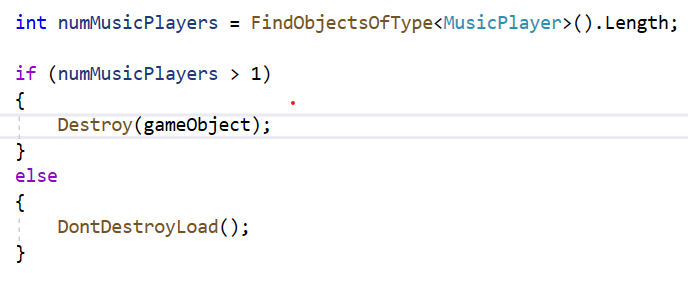
* FindObjectOfType Keyword

1. "FindObjectOfType" is a keyword which returns a game object to which the script is attached. If the class name of the script is used as a variable type. (Here Editor is the class name of another script)
2. So it will return the latest game object attached to that script.
3. Syntax –
4. In the picture above we are saying to find and return the game object to which the script with class name Editor is attached.
5. It is also used to implement the methods of the class specified in it.
6. Example –

* Here we have 2 classes Plane and Cube and currently we are in Cube.
* The we declare an instance variable of type Plane.
* Then we assign the value of variable declared above as ‘FindObjectOfType’ and specify the type of game object or we give it a class name.
* Then we can call any method in that class specified in the ‘FindObjectOfType’, note that the method we are calling should be public.
* FindObjectsOfType –

1. It is same as FindObjectOfType but the difference is if we want multiple game objects in return then we use ‘FindObjectsOfType’ because it may be possible that the same script is attached to many objects.
2. To use it we have to declare the type as an array or List.
3. And rest of the things are same as FindObjectOfType, but it will return all the game objects attached to that script.

* Singleton Concept –

1. In the singleton pattern if something same occurs again so we stop it.
2. It is used when exactly one object is needed but more than that occur so we use it to disable the objects after the first one.
3. Use of Singleton Pattern in our game –
4. It makes the game Object play only once not more than that.
5. Explanation – We declare a variable which will find the all the game object attached to that script. Then we say that, if the game object works more than once, then destroy all the Game Objects which will start working after the first one, or else don’t destroy the Game Object.

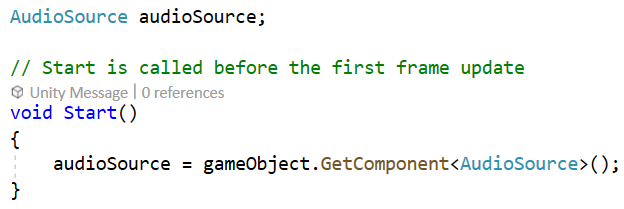
* “OnParticlesCollision” Method –

1. It is a method provided by the Unity engine for response on collision with a particle.
2. Syntax –

* AddComponent –

1. It is almost like GetComponent but with a minor difference.
2. It lets us add a new component through the script whereas GetComponent refers to the already added component.
3. By using it we don’t have to add new components to the GameObject in the Unity Editor.
4. Syntax –
5. Here “Collider” is a component being used as a variable type. “enemycollider” is the variable name. “AddComponent” will add the new component of type “Collider” and we have to also specify that which type we want, here that is “BoxCollider”.

* PlayOneShot –

1. It plays the audio clip works like “. Play” but with requirement of whole syntax explained below.
2. Syntax –
3. In the picture above the “audioSource” is a variable of type “AudioSource” and has the value as the component attached to the game object in the UnityEditor. PlayOneShot is the command to play the audio file.
4. “deathSound” is a variable of type “AudioClip” which will store audio file provided to it in the Editor, as the type is AudioClip.
5. In the syntax below we are saying to get the component “AudioSource” attached to the game object and in the “AudioSource” component store the value of “deathSound” variable and play it only once (Because of “PlayOneShot”).

* Instantiate Keyword –

1. The instantiate keyword commands the game object to work at the position and work as the variable of value the variable Specifeid. Basically instantiate makes the game object working at runtime.
2. Syntax:
3. First word in the bracket is a variable which will have the value as the game object which we want to work (enemyDeathVFX), then the position where we want that game object to work (transform.position).
4. Note – If you are instantiating a particle system, it will enter in the ‘Heirarchy’ tab (means it will spawn in the game) but that particle system wont be played. To play it we declare the ‘Instantiate’ statement as a variable and use the ‘Play’ method with the created variable.

* Text GameObject –

1. Text is a variable type and a component provided by the "UnityEngine.UI" package just like AudioSource, Transform, GameObject.
2. It can only be used after using the "UnityEngine.UI" package.
3. Syntax –
4. “scoreText” is the variable name

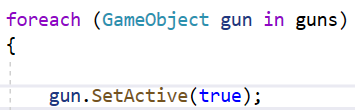
* ToString Method –

1. ToString is a method which makes a non-string value to a string value.
2. Syntax –
3. We have used “ToString” here because "ScoreText.text" asks for a String value but "score" is of type "int".
4. The "ScoreText.text" defines the value of "text" part Under the "Text" component in the UnityEditor.

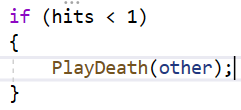
* Code to take reference from Class of another script attached to a different GameObject –

1. Declare the class name as a variable type with a name in another script. And it should be declared as a member variable.
2. Here we have taken reference from the class "ScoreBoard" of "ScoreBoard.cs" script and declared the class name as Variable type with name 'scoreBoard'.
3. Then use “FindObjectOfType” with the class name so that it can return the functions in that class, and keep it equal to the variable declared above.
4. To implement a method of the class referenced here, use the ‘variableName.MethodName();’ syntax in the method in which we want to implement the desired method.
5. 'scoreBoard' is a variable name of variable type "ScoreBoard" which is a class of another script and used here to take reference.
6. 'ScoreHit' is a method in the ScoreBoard script and we have used it here because we want to implement "ScoreHit" method in a method of the script which is taking reference from the class of another script.

* “foreach” Keyword –

1. "foreach" executes the statement for each element in it. It is used to iterate (repetition of a process) the statement or the output on the input.
2. First we specify the variable type, then the variable, which has its value as collection which we want as the output and it can be array or variables, then a collection which holds the output we want. It can be an array or variable with multiple values.
3. Syntax –
4. It is like there is a collection called “guns” which hold the output value and the variable which will hold the collection as its value and then its type.
5. "GameObject" is variable type, gun is variable name and guns is the collection (as an array).
6. The variable “gun” will hold the values of “guns” collection in it.
7. In “foreach” we say as long as we hold the input button till then it will keep on executing the output.
8. The variable above will continuously call(execute) the value of the array as the output.
9. It used in such case where we want to use all the elements of a collection with less lines of code rather than writing code for each and every element of that collection.

* Making Heath System –

1. First declare a Member variable called hits with the wanted value.
2. 
3. Then give its new value after the action, in the code by and by completing that action player’s health changes. We can either increase or decrease it. (Here we have decreased it after an action of increasing score).
4. 
5. In the image above there is an action after the which the value of health decreases. (In the game we say in score increases then value of “hits” decreases).
6. Then we say if value of hit becomes 0 or lesser than 1 destroy the game object.
7. Here we have destroyed the Game Object in the method “PlayDeath” with some effects.

**Defend It Fourth Game**

* ExecuteInEditMode **–** Makes all instances of a script execute in Edit Mode.
  + It executes the script in the edit mode, it doesn’t matter whether we have played or not.
* Mathf.RoundToInt –
* Returns the Round Off the of the number in integer form.
* Range –
* The Range keyword makes the value of the variable ranged between the points provided in it.
* It will also make a bar of range in which we can move the point to change the value of variable.
* Here the Range is 0 to 20.
* TextMesh –
* "TextMesh" is a variable type which is a component in Unity Editor.
* The ".text" is a part of the "TextMesh" Component which shows the text provided, in the Runtime and in editor or we can say it is a place where we can enter the text which we want to show.

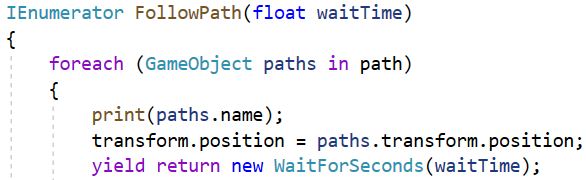


* GetComponentInChildren –
* "GetComponentInChildren" returns the component added to a Game Object childed in the Object to which the Script is attached.
* List –
* It is something same as array but it is more flexible.
* It is a generic collection thus it is more flexible.
* Steps for using it – First declare a List then, declare the variable type in the angular bracket (<>) then specify the variable name, then give its value either in engine if it is [SerializeField] or give the list of values in the script itself.
* If we have to give its value in the script, then we should use the ‘. Add’ method.
* Syntax – ListName.Add(variable);
* Explanation – name of the list the ‘Add’ method the value which we want to put in that element of list.
* To reverse its elements, we use the ‘. Reverse’ method.
* “.name” Keyword –
* It returns the name of the Game Object as the value provided, but it should be used with a variable.
* “gameObject” is the variable it should be used with, “objectName” is the value provided to it. So the name of the Game Object will be changed to the value of the “objectName”.
* InvokeRepeating –
* It works same as invoke but it just makes the method repeating, after a given interval of time.
* We can use it to repeat a method after the given interval of time.



* The ‘1f’ is the time given after which the method will be implemented, and the ‘2f’ is the time after which the method will be repeated.
* Coroutines –
* A coroutine is a function that allows pausing the execution of a method and resuming from the same point after the given time and condition is met.
* **IEnumerator** – To use coroutines we use a return type called “IEnumerator” not “void”. It is same as ‘void’ but it can only be used if we are using coroutine. And to call the method with ‘IEnumerator’ return type we use “StartCoroutine”. (FollowPath is the method name here).



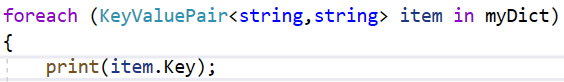
* **StartCoroutine** – It is a method used to call an ‘IEnumerator’ method. It is similar to just calling a simple void method, just the difference is that we can use it only with ‘IEnumerator’ method. (‘FollowPath’ is the ‘IEnumerator’ method here thus we are calling it with ‘StartCoroutine’).
* **Yield** – It pauses the script and continues it after the time given in the "WaitForSeconds ()" variable. It suspends the coroutine execution of the method for the given amount of seconds. It pauses the execution of the method and automatically resumes it after the given time.
* **WaitForSeconds** – It is a variable provided by a package which holds the amount of time after which we want the method to resume or continue. It should be a float value.
* **Full Code for how to use coroutines** –
* First declare method of return type ‘IEnumerator’.
* Print what we want in the output.
* Then use ‘yield’ which makes the method stop and resume after the given interval of time.
* Then use ‘return’ and ‘new’ keyword as it is the syntax for it. Then use ‘WaitForSecond’ to specify after how time we want the method to resume the method (Should be a float value or a float variable).
* RequireComponent –
* When you add a script which uses ‘RequireComponent’ to a GameObject, the required component will automatically be added to the GameObject.
* For example, a script might require that a Rigidbody is always added to the same GameObject. Using RequireComponent this will be done automatically.
* Vector2 –
* It is a variable type which represents 2D coordinates and points.
* It holds value of the position of Game Object in X and Y axis. And also returns the same if used as a method.
* It is used to represent the position and movement of a Game Object in X and Y.
* It is used to determine the position of Game Object which comes under ‘Transform’ component. As when an object will move it will surely change its position.
* Vector2Int –
* It is a variable type which represents 2D coordinates and points using integers.
* It is used where we don’t need floating-point of coordinates.
* It is same as Vector2 but only uses Integers.
* ‘Dictionary’ Generic Collection –
* It is same as array and lists but the major difference is that it works on keys and values, and not on index number and values.
* We use it when we dont want to define the values with index, but with some specific type, which we can keep anything as our choice.
* Declaration of a Dictionary: -
* The first one in angular bracket is the data type of the ‘Key’ and second one is the data type of the value. Then there is the variable name, and that is equal to a new object which is a part of its syntax. (Waypoint is a script used as variable type)
* Declaration of the key and value in the dictionary: -

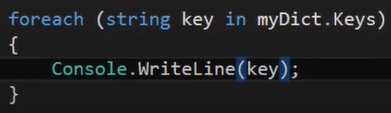
For ‘grid’ Dictionary

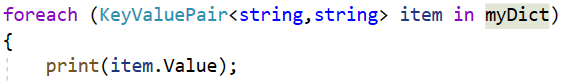


For ‘myDict’ Dictionary

* To declare the key and value we use the ‘variableName.Add’ keyword and then in “C” bracket we enter the of key and value. First one for key and second one for value. And we can have multiple keys and their values.
* If we want the output as if we specify the key name and get its value as output, we simply use.
* If we want only the key name as the output, we use a ‘foreach’ statement, and as we are using a dictionary so we can’t directly give the variable type and name, in this case we use “KeyValuePair<VariableType, VariableType>” and then variable name. And to specifically print the key we use “variableName.Key”

OR



* For specifically printing the value: -

OR

* For printing both simultaneously in the same line: - (the foreach is common in both)

OR

* If we want the number of keys and values in the Dictionary as the output, we use ‘DictionaryName.Count;’. (the variable name of the dictionary should be used with the ‘. Count’ method).
* Debug.Log Statement –
* It is same as print statement. It will work same as print.
* It has 3 types –
* Debug.Log – It works as simple print statement which will print the output in the console in the same way print does.
* Debug.LogWarning – It will also print the output in the console but as a Warning. (The yellow one).

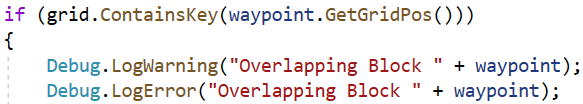


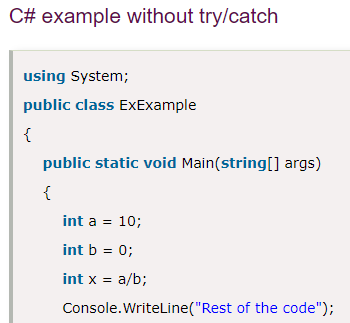
This is a Warning

* Debug.LogError – It will also print the output in the console but as an Error. (The red one).



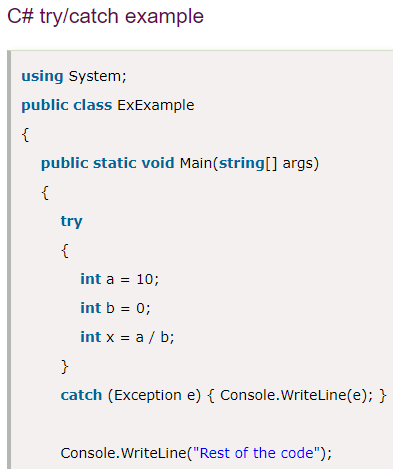
This is an Error

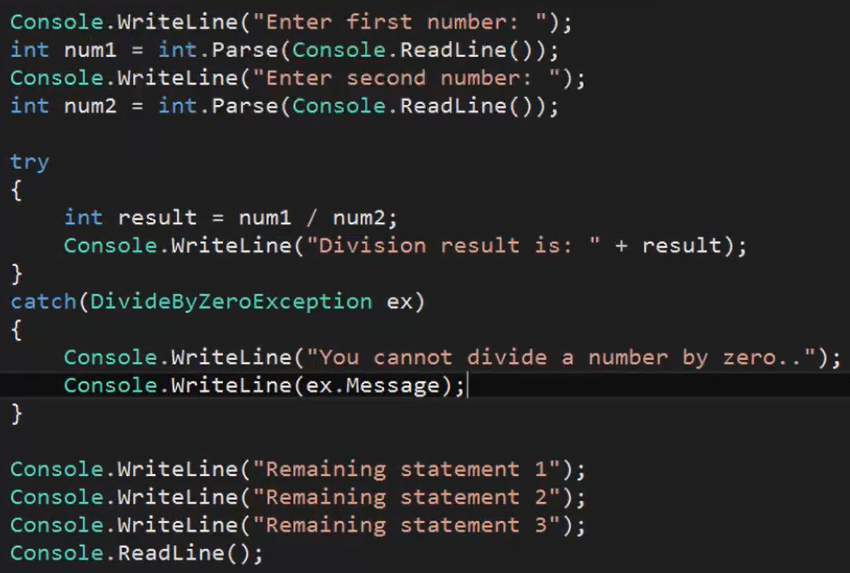
* ContainsKey Method –
* ‘ContainsKey’ is a Dictionary method in C#.
* It is used to check whether a key in the Dictionary exists or not.
* Or we can say we need to check the existence of a particular element in the Dictionary. So for that, we use the ‘ContainsKey()’ method.
* It is used with the variable name of the Dictionary in which we want to check whether the key exist in it or not.
* Example 1 –
* Explanation of Ex1 – We are saying if the ‘grid’ Dictionary has one of its key as the method ‘GetGridPos’ then show the following. (Here the data type of key is ‘Vector2Int’ and we have used a method of the same type)
* Example 2 –
* Ex2 Explanation – Here we are saying if the dictionary ‘dict’ contains a key as “Laptop” of type string or we can say if ‘dict’ has a key as “Laptop” and it is true then show the value of the key specified.
* ContainsValue Method –
* It works same as ‘ContainsKey’ but instead of checking the key of the dictionary it checks the value in it. And if condition follows it gives the output.
* MeshRenderer –
* The ‘MeshRenderer’ is a component and a variable type provided by Unity Engine to access the MeshRenderer component. It is the same as ‘Transform’, ‘RigidBody’, ‘AudioSource’ and other components like this.
* ‘. material’ – It is a part of the ‘MeshRenderer’ component which holds colors as a value. It can be provided to it in the Unity Editor or from the Script. If we want to use it, we should use the variable of type ‘MeshRenderer’ with it in this way –
* ‘. color’ – It is a keyword provided by Unity Engine to set the colour of the ‘material’ (it is a part of ‘MeshRenderer’ component) and the color will be applied to the game object at run time. Syntax is in the picture above. (The ‘color’ in RHS is variable and the ‘. color’ in LHS is the key word).
* ‘Color’ variable type –
* It is a variable type which holds its value as colours.
* For example, if there a variable ‘color’ of type ‘Color’ it will not hold a value of int, string, float or any other, it will hold its value as colour names like ‘red’, ‘green’, ‘cyan’ and etc.
* And after we have assigned it a value of ‘red’ (as it will accept its value as color names only) then it will take red colour and after applied to the Game Object it will make that object red or the colour provided to it as its value.
* Syntax to assign the value of a variable of type ‘Color’.
* ‘transform. Find’ Method –
* "transform. Find" is a method which finds the child GameObject of the parent GameObject to which the script is attached and returns the child game object.
* Here “Top” is the name of the child Game Object which we want the code to find. Note the name of the Game Object which we want our code to find should be a string.
* To check that if the code is correct or not, or to get the output just for checking we can make it a print statement.
* Exceptions –
* Exception is a unique situation which disrupts the normal flow of the program.
* It is a situation when our code of the program encounters a situation with which it can’t execute the further part of the program (as that situation is not explained in the code) then an exception occurs.
* It is a situation when something which is not explained in the code occurs it is called an exception.
* For example, we have written a program to take two numbers as input then divide them and give the quotient as the output, but this simple code only works for natural numbers not for ‘0’. So then an error will occur, program will be terminated and the further statements will not occur.
* When an exception occurs in the program either it must be handled within the code otherwise the compiler will show an error, the program will be terminated and the further statements will not be executed.
* **Exception Handling –** Exception Handling is a mechanism which is used to handle the exceptions in the program so that the program will run smoothly and the further statements will be executed even after occurring of an exception.
* We perform exception handling so that normal flow of the application can be maintained even after runtime errors.
* Keywords to use Exception Handling – ‘try’ keyword and ‘catch’ keyword.
* The try keyword is the block which hold the statement where an exception may occur.
* The ‘catch’ keyword is the keyword which will hold the output for the situation when the exception will occur. Or we can say it is a block in which we handle the exception if it occurs in the ‘try’ block. (It will not be executed when there is no exception).
* The catch block must be preceded by try block. Or The try block must be succeeded by catch block.

Output will be an error but the code is still correct which means an error will occur at the run time. And the statement further will not be executed. (Error will come because we haven’t used try/catch it the code).

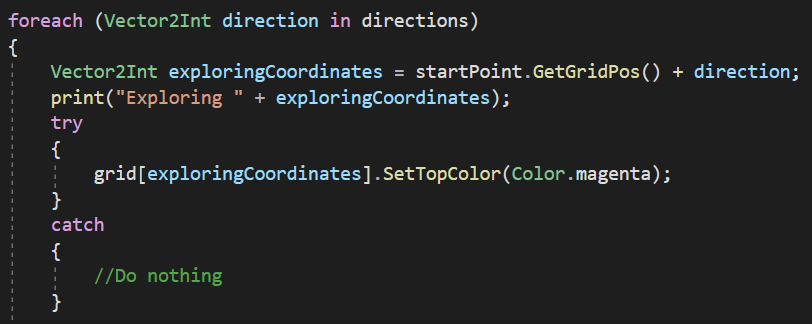
Example 1 – Without ‘try/catch’

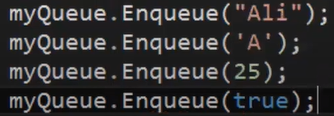
Example 2: – With ‘try/catch’

* Output of the code above: –

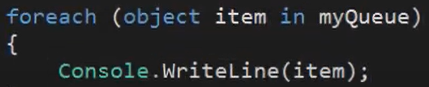
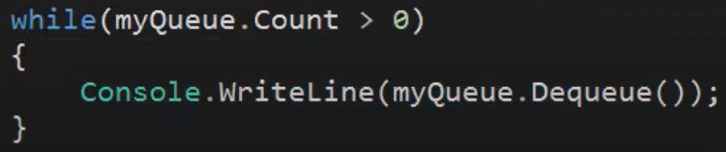
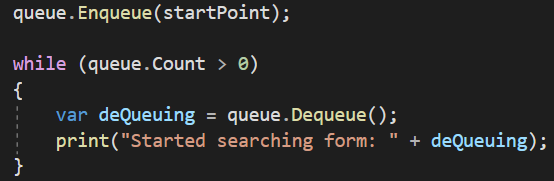
Example 3: –

* Explanation of Example 3 – In the code above we are taking the input directly from the user. So if the user enters the value of ‘num2’ as ‘0’ then it will be an exception but still there will not be any RunTime Error as we have used ‘try’ and ‘catch’ block so the further statements will be executed. And if the input given by the user is a natural number then it will simply work by giving the answer of the division and print the further statements.

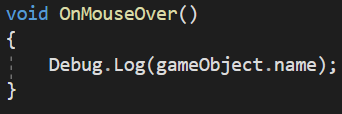
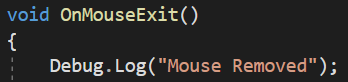
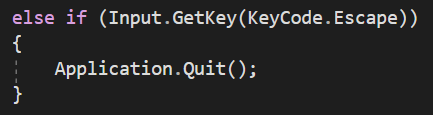
Example 5: –

* Here we are saying that if the 'exploringCoordinates' blocks are not present in the direction specified in the 'directions' array then dont show any RunTimeError (as we have used 'try/catch' keyword so no error will be shown) and do nothing (as we have commanded nothing in the 'catch' block for the exception) so the program will continue as it is, the program won't be terminated and there won’t be any RunTimeError.
* The ‘startPoint’ is the variable which refers to the method of another class (here the class is ‘Waypoint’) as it’s data type is that class only. Thus it is able to refer to the method of that class (here that method is ‘GetGridPos’ under ‘Waypoint’ class which is also the data type of the value of the variable ‘startPoint’).
* The ‘grid’ is a dictionary which has its value of type of a class (here the class is ‘Waypoint’) thus it is able to refer to the method of that class. (The method here referred is ‘SetTopColor’ and it is under ‘Waypoint’ class which is also the data type of the value of the ‘grid’ dictionary).
* Direction in the ‘directions’ array: –
  + ‘. up’ denotes the game object position + (0, +1) units.
  + ‘. down’ denotes the game object position - (0, -1) units.
  + ‘. right’ denotes the game object position + (1, 0) units.
  + ‘. left’ denotes the game object position - (-1, 0) units.
* Queue Collection –
* It is a collection type same as array and list if its Non-Generic type is used and if the Generic type is used it is like dictionary.
* Declaration
* It holds its values in the order its values are provided, for example, people are standing in a line so it is always the turn of the person who is at the first place in that line. It is same in ‘Queue’ the value provided first will be returned first and then in the order in which the value is provided.
* It works on the ‘FIFO’ concept, which means First in – First Out. It means the value provided first will be returned first and the rest in the same order.
* Adding values in queue. Note, it accepts all types of values in a single collection. It can be of any data type.
* To declare the elements of the queue we use the ‘Enqueue’ method
* The ‘Enqueue’ method is used to add an element the queue.
* ‘Dequeue’ Method – It removes and returns the element at the first position in the queue. We use it to retrieve value from the queue.
* If we have already Dequeued one element after that, the first element will be the one which is after the element which is dequeued.
* Count Method – Returns the total numbers of element in the queue. If it is used after the Dequeue method then it will show the number of elements after removing the dequeued ones.
* Peek Method – Returns the first item in the queue. 
*  Contains Method – it checks whether an item is in the queue or not. If our queue contains that element specified, then the output will be true or else it will be false.
* Clear Method – It removes all the items from the queue. It will clear the whole queue and if we want to check it we can use ‘Count’ Method after using it the output will be zero.



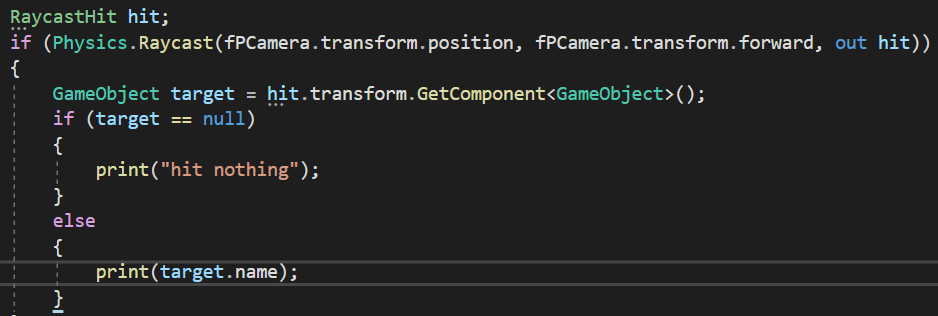
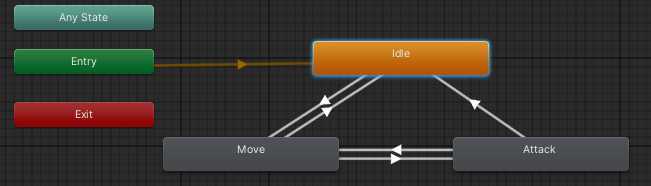
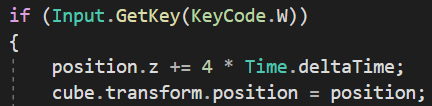
* Usage of Queue with ‘foreach’ – We are using foreach because we want all the Enqueued elements in the output.
* Usage of queue with while loop – In the code below we are saying that, if the number of elements in the queue is more than 0 then return all the element of that queue and also remove that element from the queue. To check that we can use ‘count’ method (outside while loop) it will show the number of elements in the queue is 0.
* Example of the above code in our Unity –
* First declare a queue of type ‘Waypoint’.
* Then we Enqueued a value in that queue. And then we return and remove it in a while loop. (in the while loop it is same as the example above).
* Reverse Method –
* Reverses the order of the elements in the List<T> or a portion of it if that portion is specified.
* Syntax – ‘ListName.Reverse();’.
* LookAt Method –
* To use it first we declare variable of type ‘transform’ and insert its value in the inspector or we can use the ‘gameObject’ variable in place of that variable if we want that whole game object to move or if we want 1 specific part to move we use a variable that holds its value as ‘transform’ component of that game object as its value.
* It rotates the game object in the direction of another game object that is specified in the code.
* It will make the game object (the game object specified before that method) rotate, in the direction of the game object specified in that method.
* Syntax –
* ‘objectToPan’ is the game object which will rotate in the direction of the ‘targetEnemy’ game object.
* Vector3.Distance Method –
* It the method which returns the distance between the game objects specified in it. Note, to specify the variables in it we use them with their ‘transform. position’ keyword.
* This method is also used to make the range for a game object. For example, if something is in the range of that game object then do that action.
* Syntax –

Vector3.Distance(a. transform. position, b. transform. position);

* Example – Here: a = targetEnemy, b = gameobject
* “. emission” Module –
* It is a keyword used to call the emission module of a particle system.
* Syntax – variableName.emission
* Example –
* To use it we set the variable (here ‘emissionModule’) as ‘true’ by using the ‘. enabled’ keyword. (Here ‘isActive’ is a boolean variable which can either be true or false depending upon the present condition).
* ‘. enabled’ Keyword – It specifies whether the emission module under particle system is enabled or disabled. Note, if it is equal to ’true’ the emission of the particle system will be done, if it is equal to false the emission of the particle system will not be done.
* OnMouseOver –
* It is a method provided by unity engine to make a response on putting the mouse cursor on a game object to which that script is attached.
* This method will work at the moment when we will put the cursor on the game object.
* Note it will only work if the game object has colliders on it.
* OnMouseExit –
* It is a method provided by unity engine to make a response on removing the mouse cursor from a game object to which that script is attached.
* This method will work at the moment when we will remove the cursor on the game object.
* Accessing modules and components–
* For accessing any of the desired module or components of any game object we can use the name of that module, component or any module under that component we use the name of that module after a full stop (.).
* Examples – transform. position, transform. rotation, transform.scale.
* Like in the examples above we can access to any module of any game object or any module under of the component.
* ‘this’ Keyword –
* It a keyword used to refer to the class in which it is used.
* In the example below we are saying to apply the ‘PlaceTower’ method of class ‘TowerFactory’ on the class in which this statement is written, or apply that method to the game object to which this class is attached. (Here that class in which this statement is written is ‘Waypoint’).
* Childing an instantiated game object –
* To do this first we declare a public variable of type ‘Transform’ in the script in which we are instantiating the game object.
* Then we set the parent of instantiated game object as the variable declared above. (‘newTower’ is the instantiated game object).
* Post Processing –
* It is a way of giving visual polish to your game, or making it look good.
* It is installed from package manager. Steps – Window > Package Manager > find post processing and install it.
* To make its effect working –
* Create a new game object, add the ‘Post Process Volume’ component to that game object. Make its layer ‘Post Processing’. Turn on ‘isGlobal’
* Then add the ‘Post Process Layer’ to the main camera, and set the layer of the ‘Post Process Layer’ as Post Process Layer.
* Then add effects to the ‘Post Process Volume’ component.
* ‘Debug.Break’ Statement –
* It is a statement used to automatically pause the game after the execution of the statement after which it is written.
* In this example the break statement will be executed only after the execution of the statement above it. The moment the above statement will be executed the break statement will be executed and it will pause the game at that moment.
* ‘PlayClipPoint’ Method –
* It is a method provided by Unity Engine to play an ‘AudioClip’.
* Here it is used because we are destroying the game object, and if the game object is destroyed the ‘PlayOneShot ()’ or ‘Play ()’ method won’t work because the game object will be destroyed. So we use it as it creates a new game object at the runtime and plays the audio.
* This function creates an audio source and automatically disposes of it once the clip has finished playing.
* It is almost like Instantiate, but the one which is only for Audios. As it plays an ‘AudioClip’ at a given position in world space and also asks for a volume (type float), but it is not necessary.
* Note – Its position should always be given as the same as that of the ‘Main Camera’ game object, otherwise it will not be heard by the ‘Audio Listener’ component of the camera.
* ‘Application. Quit ()’ Method –
* This method is already provided by unity engine for quitting the game or application.
* It will only work after building the whole game not in the Unity Editor game view.
* NavMeshAgent –
* It is a component used to set the area on which a game object can move. It is a game object which the player is not controlling.
* Note – The game object with ‘NavMeshAgent’ will only move on those things which are set as static.
* As it is component we can use it as a variable type also in the scripts.
* Making a game object move towards a target –
* For that we declare a public instance variable of type ‘Transform’ and give its value as the game object towards which we want the game object to move.
* Then declare an instance variable of type ‘NavMeshAgent’. To do that we use a namespace otherwise it won’t work.

Variables Declared –

Namespace –

* Then call the ‘GetComponent’ method in the ‘Start’ Method.
* Then in Update call the ‘SetDestination’ method with the variable of type ‘NavMeshAgent’ declared above. Then, pass the position (‘. position’) of the variable of type ‘Transform’ declared above, in the parameter of that method.
* 'stoppingDistance' is a module under the 'NavMeshAgent' component which defines that a game object will stop following the player if the distance between the player and that game object is equal to the value given in 'stoppingDistance'
* ‘SetDestination’ Method –
* It is a method under the ‘UnityEnfine.AI’ namespace.
* It is used to make a game object move towards the referred game object in that method’s parameter.
* It will make the game object to which the script is attached move towards the game object given in its parameter.
* It will make the game object (to which the script is attached) follow the referred game object.
* Note – The referred game object should be of type ‘Transform’ and the ‘. position’ should be called, and if it is of type ‘GameObject’ we can use the ‘. transform. position’.
* Example – Here we have called it with a variable of type ‘NavMeshAgent’.
* Getting FPP or TPP camera view for the player–
* To get the FPP or TPP camera view for the player we use the Standard Assets package.
* First we import that package in the game.
* In it we get the ‘Standard Assets’ folder, in it we go to the ‘Characters’ folder, in it we get 3 folders ‘FirstPersonCharacter’ folder, ‘ThirdPersonCharacter’ folder and ‘RollerBall’ folder choose and open the one which we want in our game. By opening any of those we will surely find the ‘Prefabs’ folder, then open it and drop the desired prefab into the scene.
* Scripts for movement and walking audio will already be attached to the prefab dropped in the scene. And also we can rotate the camera (Means – Look around) with movement of mouse, the same way as it is in GTA games, FPS games, games with TPP and FPP camera.
* ‘OnGizmoSelected’ Method –
* It makes a shape of the range (the range of the game object, in which, if the player is in it will do its action) of the game object.
* For example, if there is an enemy is our game it will show the detect or attack range of the enemy in the game to detect the player.
* It will only show the range of the game object if that game object is selected, as selected word is in its name.
* There is another version of it, which is without selected, it doesn’t matter whether you have selected that game object or not it will always show the range of that game object.
* Code to show the range of the game object of that game object –
* ‘Gizmos’ is the variable type used to show the range of the game object.
* In the first line of the code above we are declaring the colour of the ‘Gizmo’. That the colour of the range of the game object will be the value given of the variable.
* Then we call the ‘DrawWireSphere’ method and give the value of its centre, which will be the position of the game object, and then we give the value of its radius.
* ‘DrawWireSphere’ method – Draws a wireframe sphere with centre (centre as the position of the game object) and radius (radius as the range given).
* If we don’t want a wireframe sphere and we want a full sphere or we can say it a solid sphere so then we use the –
* ‘DrawSphere’ method – Draws a solid sphere with centre (centre as the position of the game object) and radius (radius as the range given).
* ‘DrawWireSphere’ and ‘DrawSphere’ are the main methods which set the gizmo of the range of the game object.
* To assign the value as Infinite –
* To assign a variable as infinite we use the ‘Mathf.Infinity’ keyword.
* For making range of a game object –
* For making range of a game object –
* First we declare an instance variable of type ‘float’ whose value will be the range of that game object in which it will do its action.
* Then we declare an instance variable of type ‘Transform’ which will hold its value as transform of a game object provided in the unity editor.
* Then we declare an instance variable of type ‘float’ and assign it as infinity (variable = Mathf.Infinity;).
* It will hold its value as the distance between the two game objects.
* To set the distance between the two game objects we use the; ‘Vector3.Distance(a. position, b. position)’, which will return the distance between the two game objects.
* Then we say if the distance between the two game objects is less than the range value then executes the code in that if statement.
* Raycasting in Unity for shooting –
* A raycast is a ray that gets sent out from a position in 3D or 2D space and moves in a specific direction. Unity 3D has built-in functions that can be used to implement a Raycast in your game.
* For doing a raycast we use a line of code –
* The ‘Physics.Raycast’ is use to start the raycast, and parameters in it are used to specify that from where will the raycast start, direction towards which will it move, return the game object which is collided with the raycast, and the range of the raycast that for how much units the raycast will go.
* The ‘Physics.Raycast’ returns true if the ray hits a Collider, otherwise false.
* The first variable in the parameter is the position from where we want our raycast to start, the second one is the direction in which we want the raycast to move, the third one (‘out hit’) is the variable referring to the game object with which our raycast will collide (it should be of type ‘RaycastHit’ only), the last one is range of the raycast (means the raycast will last till the value of the range variable, if value of the range is 100 so then the raycast will last till 100 units).
* There is a specific variable type which represents the game object which we hit with our raycast, it is ‘RaycastHit’. And the type of the ‘hit’ variable is of this type only. 'RaycastHit' is a variable type which is used to store information about what we hit with our ray.
* Proper structured and conditional code if raycast collides with any referenced game object or not, if collides then give the given output. We can use any variable type in place of GameObject, it can be of type of a class or a component.
* Creating a ‘RaycastHit’ Impact–
* To create a ‘RaycastHit’ impact we instantiate the effect we want on collision of the raycast. To instantiate that game object, we use the statement below –
* In place of above statement, we can also use the below statement.
* ‘. point’ – It is the impact point in world space where the raycast hits the collider. Mainly used to instantiate a game object at the place where the raycast hits any game object.
* ‘LookRotation’ method – This method is used to set the rotation of the game object towards the direction we want. And the direction we have set is the ‘normal’ of the surface where the raycast hits.
* ‘. normal’ – It is the side of the surface the ‘Raycast’ collided with. In the code above we have set the rotation of the game object towards the normal of the surface of the game object, so that its rotation is in such a way that it faces the side from where the ‘Raycast’ initiated.
* Adding Animations to the game object –
* To add animations to the game object we add the animator component to the game object.
* ‘Animator’ Component – It is a component used for adding animation to the game object. It is also used as a variable type to handle animations through the code.
* Then we assign the ‘Animator Controller’ as value of the ‘Controller’ variable under the ‘Animator’ component.
* ‘Animator Controller’ – Used for controlling animations on a particular condition.
* After setting the conditions for the animations in the controller we make animations for the game object.
* For making animations for a game object we first create an ‘Animation’ prefab in the assets.
* Then we assign it as the value of ‘Motion’ variable in the state, which is in the ‘Animator Controller’.
* The boxes connected to each other in the image below are the states. To add animation, we select it and assign the ‘Animation’ prefab as the value of the ‘Motion’ variable of that state.
* Handling Animations from the code –
* To control animations from the code, first of all we declare a variable of type animator and then, we get the ‘Animator’ component.
* Then we set conditions for going from one state to another.
* To set conditions for states we add parameters to that particular ‘Animator Controller’. It can either be trigger or bool or float or few other variable types that are given to use in the parameter.
* Then whatever condition is given in the ‘Animator Controller’, is satisfied then go to the next state. For example, if we added parameter of type bool, and when it is true we have set it to go to the next state, so it will do it accordingly.
* And to do that in code we use the ‘SetTrigger’ method to turn on the trigger in the parameter of that controller. It takes the parameter name as string only, and turns on that trigger parameter. Here that parameter in the controller is of type trigger named “Move”.
* For boolean we use the ‘SetBool’ method. It takes the name of the parameter in the controller of type string, then we set it to either true or false according to the condition given in the ‘Animator Controller’.
* Here we have set it false. And the name of the parameter is ‘Attack’ and it is of type boolean thus it is taking either true or false.
* SetBool method –
* It is used to set the value parameter of type boolean in the ‘Animator Controller’ either true or false according to the Condition given in the ‘Animator Controller’.
* SetTrigger method –
* It is used to turn on the parameter of type trigger in the Animator Controller.
* Code for movement of a game object –
* If our game is 3D so we declare an instance variable of type Vector3, and if our game is a 2D game we declare an instance variable of type Vector2.
* Then we assign the value of that variable as the position of the game object which we want to move.
* Then we write our if statement and write the response in those statements.
* In the picture above we are saying that if input key is ‘W’ then add 4 units to the ‘z’ component of that variable of type Vector3 declare above. This is just for movement of game object in ‘Z’ axis, if we want the game object to move in different direction we can specify ‘x’ or ‘y’ in the place of ‘z’, according to which direction we want the game object to move.
* Then make the ‘transform. position’ of the game object equal to the variable declared above which is of type Vector3.
* Setting the behaviour of the cursor –
* To set the behaviour of the cursor we this statement –
* In the statement above we are saying that –
* The ‘lockState’ variable in the ‘Cursor’ class, of type ‘Cursor’, is the variable whose value will determine the behaviour of the actual cursor of the system.
* The value of the ‘lockState’ variable will determine the behaviour of the cursor of the system.
* The ‘lockState’ variable holds only three values of behaviour of the cursor:

1. None – The cursor will behave normally as it behaves the way in the normal windows, without any restriction.
2. Confined – The cursor will only be able to move in the game window, and it can’t go outside the game window as it is locked to move outside in the game window.
3. Locked – Locks the cursor in the centre of the game window.

* Controlling the visibility of the cursor in the game –
* To control the visibility of the cursor in the game we use this statement below –

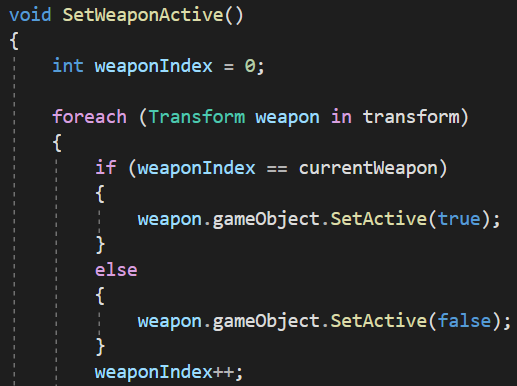
- To make the cursor visible

- To make the cursor invisible

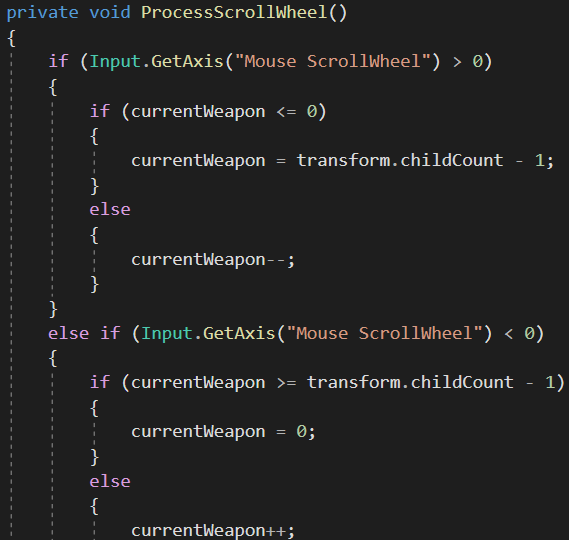
* Under the ‘Cursor’ class ‘visible’ is the variable which sets the visibility of the cursor to visible (true) or invisible (false).
* Controlling visibility of a game object –
* For controlling visibility of a game object we the statement below:

- To make the game object invisible

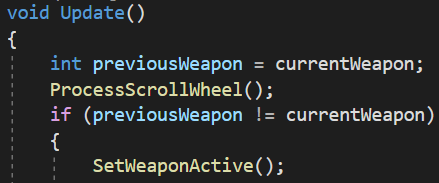
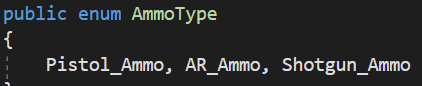
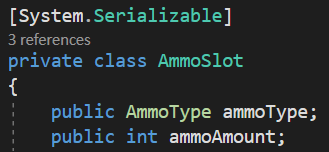
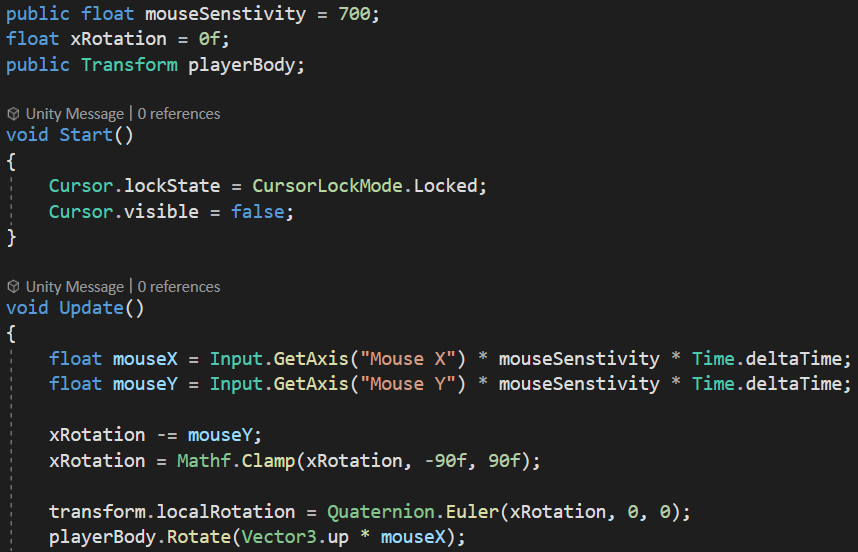
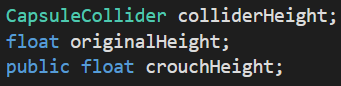
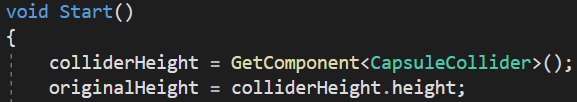
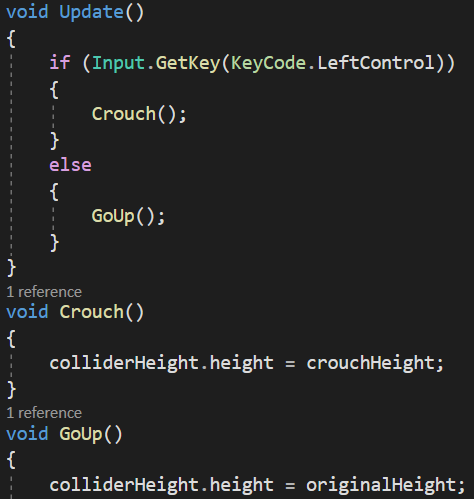
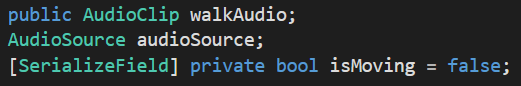
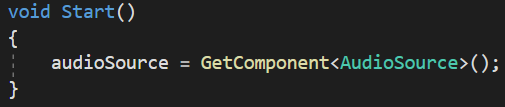
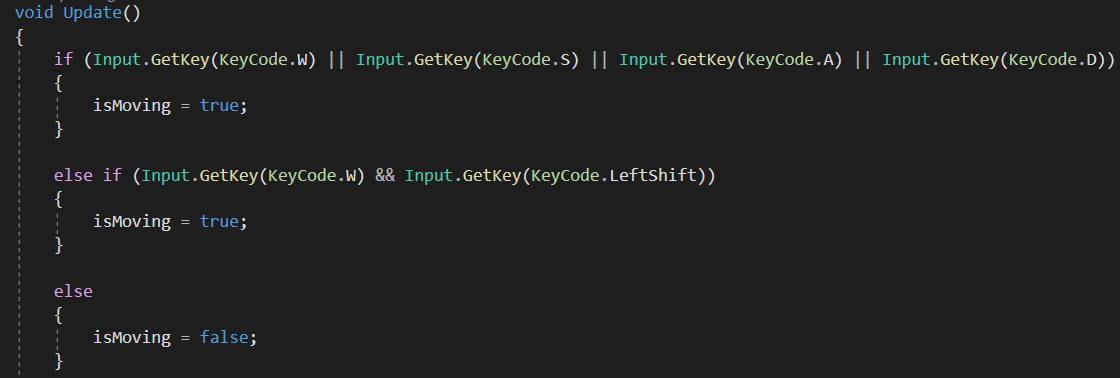
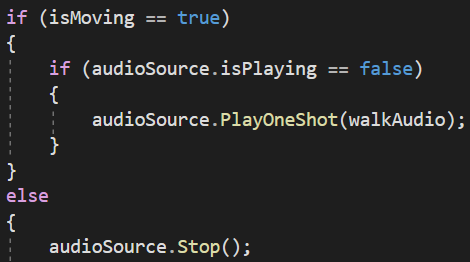
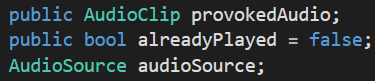
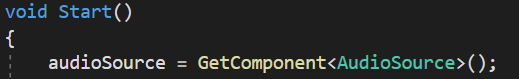
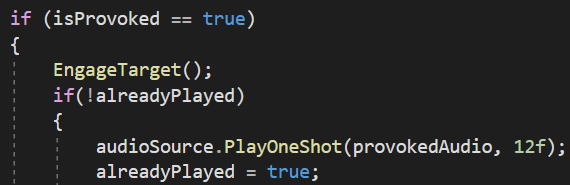
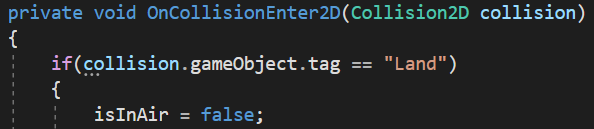
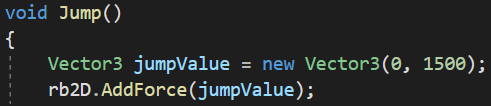
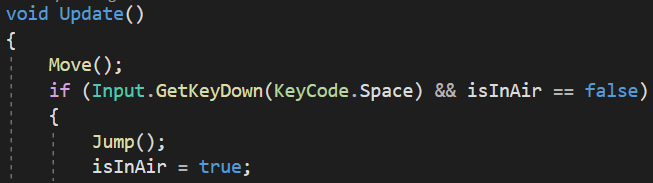
- To make the game object visible

* For this we first declare a public variable, assign it in the inspector.
* Then we set its ‘enabled’ state as true (visible) or false (invisible).
* BroadCastMessage method –
* It is used to call any public method under any script attached to that game object or any of its children.
* It takes method name as a string.
* Here we are calling a method named ‘OnDamageTaken’.
* Switching weapons –
* For switching weapons, the steps are following –
* Declare an instance variable of type ‘int’ named ‘currentWeapon’.
* Then make a method given in below –
* Explanation of the code above –

1. Declare a variable of type ‘int’, named ‘weaponIndex’ with value as 0.
2. Then start a foreach loop which will loop through all the child game objects under the game object to which this script is attached. And the game object to which this script is attached, should be the parent of the game objects through which the foreach loop will go.
3. The ‘weapon’ is a variable of type ‘Transform’, which represents to the ‘transform’ array in the end of the parenthesis of the foreach loop. The ‘transform’ array which holds its value as the child game objects of the game object to which this script is attached.
4. Then we write an if statement and in it we say, if the value of ‘weaponIndex’ is equal to the value of the current weapon then we set the game object’s ‘SetActive’ as true. The game object referred here is the variable in the ‘foreach’ loop, named ‘weapon’ which represents the child game objects of the game object to which this script is attached.
5. So the command in the if statement will make the game object’s (whose element number in the ‘transform’ array is equal to the value of ‘currentWeapon’) ‘SetActive’ as true.
6. And if the ‘weaponIndex’ not equals to ‘currentWeapon’ the make the ‘SetActive’ of the game object as false.
7. And keep on increasing the value of ‘weaponIndex’ by 1.

* This method will make the game object’s (whose element number is same as the value of ‘currentWeapon’) ‘SetActive’ as true. And we call it in the start method.
* Then we make a method which will switch the weapon on scrolling the mouse wheel either up or down –
* Explanation of the code above –

1. We are saying if the mouse wheel is scrolled upwards (i.e. more than 0) then, in it we say if ‘currentWeapon’ is lesser than or equal to 0 then current weapon is ‘‘transform. childCount - 1’’ (‘transform. childCount’ means the number of children game object under a game object, here the parent game object is the one to which this script is attached). Then write an else statement and say increase the value of current weapon by 1 (i.e. ‘currentWeapon++’).
2. We are saying ‘childCount - 1’ because the element number of the last child game object in the array in the foreach loop is 1 less than the value of ‘childCount’, that is because in an array counting starts from 0 and in the ‘childCount’ the counting of the children game object starts from 1. So obviously the total in the ‘childCount’ will be 1 more than the total count of an array.
3. Then we say else if the mouse wheel is scrolled downwards (i.e. less than 0) then, in it we say if value of ‘currentWeapon’ is more than or equal to the “transform. childCount - 1” then value of ‘currentWeapon’ will be equal to 0. Then we say else increase the value of current weapon by 1.
4. We are saying ‘childCount - 1’ because it is the last element of the transform array in the foreach loop as array counting starts from 0 and in the ‘childCount’ the counting of the children game object starts from 1. So obviously the total in the ‘childCount’ will be 1 more than the total count of an array.

* Then call the method which takes the input from the player (here that method is ‘ProcessScrollWheel’) in the ‘Update’ method.
* Then declare a variable of type int named ‘previousWeapon’ and assign its value as ‘currentWeapon’.
* Then call the method which takes input from user.
* Then write an if statement in which the condition should be that the value of ‘previousWeapon’ is not equal to the value of ‘currentWeapon’, then call the method which sets the game object’s ‘SetActive’ as true or false or we can say the one in which we had started a foreach loop (here that method is “SetWeaponActive”).
* Making Different Ammo Types for Different weapons –
* First of all, make a new script and delete everything in that script including the class name, and the namespaces which are being used, it should be totally empty.
* Then declare public Enum named ‘AmmoType’ and in it specify the types of ammo you want in your game.
* Then come in the ‘Ammo’ class which gives the mechanism of decreasing the ammo and getting the ammo.
* Then declare a private class named ‘AmmoSlot’ in the ‘Ammo’ class itself. Declare it private so that only its parent class can access it.
* Then declare a public instance variable of type of the enum declared above (the type of that variable will be ‘AmmoType’). And name that variable as ‘ammoType’.
* Then again declare a public instance variable of type ‘int’ name ‘ammoAmount’.
* Note that, both of the variables declared above should be declared in the private class created, named ‘AmmoSlot’. And should be public so that the ‘Ammo’ class can access those variables.
* Make the private class ‘Serializable’, by using ‘[System.Serializable]’ command.
* The ‘[System.Serializable]’ makes the ‘AmmoSlot’ class available to show its public variable in the inspector so that we can assign the value of the variables in the inspector. Or we can say it makes the public variables of a the ‘AmmoSlot’ class (or any child class under its parent) available to use assign in the inspector.
* It just enables us to use, assign and show the variables in the inspector of the class it is used with.
* And to see and assign those variables in the inspector we make a public variable or collection of type of that class with which ‘[System.Serializable]’ is used (here that class is ‘AmmoSlot’).
* But note that, it just makes the variables of the child class available to show and use in the inspector, it won’t be shown in the inspector until unless a variable or a collection of type of that class (here ‘AmmoSlot’) is defined. Here we have declared an array because we have multiple weapons and their respective ammo type.
* Code for Camera Rotation –
* First we declare 2 variables of type float and make them hold the value of movement of mouse, one will hold that of mouse in X axis and other one will hold for Y axis.
* Then, to rotate the camera in Y axis (Horizontal rotation), we declare a variable of type ‘Transform’ named ‘playerBody’, then child the camera game object in an empty game object with character controller component on it and assign the empty game object as the value of ‘playerBody’. Then call the ‘Rotate’ method from the ‘playerBody’ and give the value of its parameter as ‘Vector3.up \* mouseX’ (mouseX determines the movement of mouse in X axis). Here we have used ‘up’ as the parameter value of the ‘Rotate’ method because horizontal rotation of a game object is determined by the Y axis and the ‘Vector3.up’ determines its value as 0 units in x, 1 unit in y, and 0 units in z. It is a short Hand for writing ‘new Vector3(0, 1, 0)’. Thus it will make the camera rotate in the y axis (horizontal rotation) when mouse is moved in the X axis, as ‘mouseX’ determines the movement of mouse in X axis.
* Rotate Method – Applies a rotation to an axis according to the value given in its parameter to the specified axis, it can be x, y, z.
* To rotate in the X axis (Vertical rotation) we declare a variable of type float named ‘xRotation’ and assign its new value as ‘mouseY’ (it determines the movement of mouse in Y axis) subtracted from the initial value of ‘xRotation’.
* Then set the ‘localRotation’ of the camera to ‘Quaternion.Euler(xRotation, 0, 0)’
* The Euler method – It returns rotation or we can say it sets the rotation of the game object to rotate in the specified axis according to the value.
* And finally clamp the ‘xRotation’ variable from -90° to 90°.
* And disable the cursor in the ‘Start’ method for comfortability.
* Crouching of the player –
* Create a new script named ‘CrouchScript’. Attach it to the Player, and make sure that a capsule collider component is attached to the player.
* Declare 3 variables 1 of type ‘CapsuleCollider’ named ‘colliderHeight’, and other two of type float, 1 named ‘originalHeight’, and last one named ‘crouchHeight’.
* Then in the ‘Start’ method assign the ‘colliderHeight’ as ‘CapsuleCollider’ by calling the ‘GetComponent’ method.
* Then set the original height equal to the value of the height of capsule collider.
* The set the collider height to ‘crouchHeight’ if key is pressed else keep the collider height as its original height.
* Tackling disappearing point light –
* Point light disappears from a long distance to tackle that follow the following steps –
* Select the Light object
* Go to Inspector
* Then go the light component
* And change rendered mode "Auto" to "important"
* Adding Character Walking Audio-
* First of all, we declare 3 variables –
* ‘AudioSource’ for playing the audio, ‘AudioClip’ for holding the audio file which is to be played and a boolean variable to determine whether the player is moving or not.
* Then in start method we get the ‘AudioSource’ component attached to the game object.
* Then in Update method we make the ‘isMoving’ true if the following condition is followed else it is false.
* Then make another condition in which we say if the is ‘isMoving’ is true then play the audio, else stop the audio.
* 'isPlaying' is a boolean variable in the 'AudioSource' class which determines whether the AudioClip is playing or not.
* In the picture above we are saying that if the ‘isMoving’ is true and the ‘AudioSource’ is not playing (‘isPlaying’ is false) then play the ‘walkAudio’.
* Note – This code is not for only playing audio for walking but can also be used to play an audio repeatedly on a proper interval on a condition or input in such a way that it sounds realistic.
* Playing the desired audio only once –
* First of all, declare few variables, one of type bool, one ‘AudioSource’, and ‘AudioClip’.
* Get the ‘AudioSource’ component in the ‘Start’ method.
* Then in the condition after which the audio to be played, we say if ‘alreadyPlayed’ is false then play the ‘AudioClip’, with the given Volume, and then assign the ‘alreadyPlayed’ as true.
* In the picture above we say that if the enemy’s ‘isProvoked’ is true then do something, and if the ‘alreadyPlayed’ so then play the audio and then assign ‘alreadyPlayed’ as true.
* **Adding Player Jump –**
* To add player jump in your game –
* Declare 2 instance variables, one of type Rigidbody2d in 2D project or Rigidbody in 3D project and a bool named ‘isInAir’.
* Then open the OnCollisionEnter2D method (if 3D project then the OnCollisionEnter only, here we are doing in 2D thus we are using 2D). Assign the ‘isInAir’ as false of the player collides with game object tagged with land.
* Then make a new method ‘Jump’ in which we will give the jump mechanism of jumping. Here we use the ‘AddForce’ method of the Rigidbody class, and give its parameter value that how much to jump, which here is ‘jumpValue’ of type Vector3.
* To make the player jump only when it is on ground –
* In the Image above we say that if space is pressed and ‘isInAir’ is false then only make the player jump and we assign isInAir as true so that we cannot jump in air because the condition of ‘isInAir’ being false, so that condition will not be followed if the player is in air, and then after touching the land, the player will be able to jump as ‘isInAir’ will become false due to the ‘OnCollisionEnter’ method and when on ground and space is pressed then we can jump.