SINGLETON MANAGER

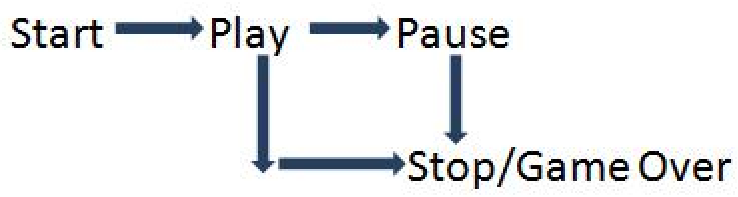
Jump To:

Objective

Main objective of this blog post is to give you an idea about how to use Managers Using Singleton

Need of Game Managers

Any game made by anyone has a natural working flow



All these steps can be considered as states of game. To manage all these states we need a proper manager who can provide a mechanism to know when to change the state from **state 1** to **state 2** during gameplay and if required does some stuff like event trigger (e.g. store global data like score) before the change in state.

For functionalities like playing audio, managing UI elements (Main menu, Sub menus, Score panel etc.), managing list of objects in more than one scripts and scenes it is recommended to create a manager because

- It gives reusability of code without redundancy.

- It is easy to understand, refactor and change without any ambiguity.

E.g.

- Texture manager, Audio manager, Configuration manager, Input manager, UI manager, Game manager and some others as per the requirement.

Step 1 Why Singleton?

In developing games we need global access to managers provided that it is instantiated only once across the game.

For Example:

- To play an audio in the game, we need single global instance of the Audio manager because it is not convenient to make different audio managers for different scenes and it may cause game to behave incorrectly.

In short it means that managers need to persist across all scenes and shouldn't be destroyed when new scene is loaded.

Solution 1: The most common approach to make a single globally accessible instance in unity is to **create a prefab that has all manager scripts** attached to it. You may have dozens of other manager-like scripts that you can bind with a single game object and use it as follows:

- Include that prefab in every game scene.

But, singleton is an advantage because it creates only one instance.

Solution 2: Second option is to **make manager class static** but it is having the following disadvantages.

- Static classes can't be instantiated.
- Static classes can't inherit other class so it cannot inherit Mono Behavior and can't be a component.
- You can't pass a static class as a parameter.

Whereas, Singleton class can inherit Mono Behaviour and instance can also be passed.

Solution 3: Third option is to **make all the members of the class public static**. But,

- Static values are initialized before instantiation which is not required.
- If a class inherits interface then the implementation of methods of interface can't be static so it can't be used in other classes without making instance. For this kind of scenario we have to create instances where we want to use those methods i.e. it can't use inheritance of interfaces or abstract classes.

Singleton also gives a solution for this kind of problem.

Step 2 Introduction to Singleton

The name Singleton itself gives the meaning **A single thing of the kind under consideration**. Singleton pattern lets you write a class, which can only be instantiated once. Singleton pattern is a very useful, yet it’s a very simple design pattern.

It restricts instantiation outside the class by making constructor private.

```
public class Singleton : MonoBehaviour{
//create an object of Singleton
public static Singleton instance = new Singleton();
//make the constructor private so that this class cannot be
//instantiated for outside the class.
private Singleton(){}
}
```

The biggest reason singletons are used is for code clarity. Any programmer that realizes something is a singleton is instantly informed of how it should be used. Beyond conceptual aids a singleton can also be an efficient means of allowing global access of an object.

The singleton design pattern facilitates global access to an object while ensuring that only one instance of the object ever exists at any one time. If an instance of the singleton doesn’t exist when it is referenced, it will be instantiated.

- Initialization of Singleton

Initializing singleton at startup:

```
void Awake()
{
// Save a Reference to the component as our singleton instance
Instance = this;
}
Public static Singleton Instance{get;private set;}
```

Step 3 Example

E.g.

```
Using UnityEngine;

Public class AudioManager : MonoBehaviour
{
Public AudioClip Clip;

// Static singleton property.
Public static AudioManager Instance { get; private set; }

Void Awake()
{
// Save a reference to the AudioManager component as our //singleton
instance.
Instance = this;
}

// Instance method, this method can be accessed through the
//singleton instance
Public void PlayAudio(AudioClip clip)
{
audio.clip = clip;
audio.Play();
}
}
```

Now to access this we need to write:

```
AudioManager.Instance.PlayAudio(AudioManager.Instance.Clip);
```

This is instantiation in awake whether we want to use it from start or not, so for this lazy initialization is the only solution.

- Lazy Initialization

Lazy initialization is a technique where one postpones the instantiation of an object until its first use. In other words the instance of a class is created when it’s required to be used for the first time. The idea behind this is to avoid unnecessary and costly process of instance creation till it is not required.

```
public class UIManager : MonoBehaviour
{
public static UIManager UIManagerInstance ;
public static UIManager Instance {    // Single Instance assurity
get {
if (UIManagerInstance == null) {
UIManagerInstance = FindObjectOfType ();
if (UIManagerInstance == null) {
GameObject obj = new GameObject ();
UIManagerInstance = obj.AddComponent ();
}
}
return UIManagerInstance;
}
}
void Awake ()
{
GameObject.DontDestroyOnLoad (gameObject);
}
```

The above implementation first searches for an instance of the Singleton component in the scene if a reference doesn’t already exist. If it doesn’t find a Singleton component, a **GameObject** is created and a Singleton component is attached to it.

Next line initializes the public static reference of Singleton. One line code in ***Awake()*** Method makes UIManager to persist in all scenes.

It can be very lengthy process to copy and paste this code to every manger class. C# supports generic class so we can create generic class which also provides protection against race conditions as follows:

```
public class Singleton : MonoBehaviour where T : MonoBehaviour
{
private static T _instance;

private static object _lock = new object ();

public static T Instance {
get {
if (applicationIsQuitting) {
Debug.LogWarning ("[Singleton] Instance '" + typeof(T) +
"' already destroyed on application quit." +
" Won't create again - returning null.");
return null;
}

lock (_lock) {
if (_instance == null) {
_instance = (T)FindObjectOfType (typeof(T));

if (FindObjectsOfType (typeof(T)).Length > 1) {
Debug.LogError ("[Singleton] Something went really wrong "+
" - there should never be more than 1 singleton!" +
" Reopenning the scene might fix it.");
return _instance;
}

if (_instance == null) {
GameObject singleton = new GameObject ();
_instance = singleton.AddComponent ();
singleton.name ="(singleton)" + typeof(T).ToString();

DontDestroyOnLoad (singleton);


Debug.Log ("[Singleton] An instance of " + typeof(T)+
" is needed in the scene, so '" + singleton +
"' was created with DontDestroyOnLoad.");
} else {
//Debug.Log("[Singleton] Using instance already created: " +
_instance.gameObject.name);
}
• }
•
• return _instance;
• }
• }
• }
•
• private static bool applicationIsQuitting = false;
• public void OnDestroy ()
• {
• applicationIsQuitting = true;
• }
• }
```

Conclusion

Some games require dozens of manager scripts so it is necessary to reduce the amount of duplicated code and standardize on a method for setting up, referencing, and tearing down these managers. A generic singleton base class is one such solution that has served us well.

Although Singleton is a design pattern, it is not the perfect solution. Use of singleton in some scenario can be helpful and in some cases it is not recommended.

However Singleton pattern gives flexible way of managing objects as per our requirement.

I hope you find this post very helpful. Got an Idea of **Game Development**? What are you still waiting for? Contact us now and see the Idea live soon. Our company has been named as one of the best Game Development Company in India.
- See more at: http://www.theappguruz.com/blog/managers-using-singleton#sthash.eQCDOrZH.dpuf