Pool Manager script breakdown

This document will break down what is happening in the **Pool Manager script**, which is part of the **Zombie Project session 3** on the **DLE**

If you want to know what **Object Pooling** is and how we are using it in the **Zombie game**, please refer to the **Object Pooling help sheet** on the **DLE** for this week

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;


public class PoolManager : MonoBehaviour
{

    public static PoolManager current;

    public string[] names;
    public GameObject[] pooledObjects;
    public int[] poolAmounts;

    public Hashtable mainPool = new Hashtable();

    public List<GameObject> tempList;

    void OnEnable()
    {
        current = this;
    }

    void Start()
    {
        tempList = new List<GameObject>();

        for (int i = 0; i < names.Length; i++)
        {
            List<GameObject> objList = new List<GameObject>();

            for (int j = 0; j < poolAmounts[i]; j++)
            {
                GameObject obj = (GameObject)Instantiate(pooledObjects[i]);
                obj.SetActive(false);
                objList.Add(obj);
            }

            mainPool.Add(names[i], objList);
        }
    }

    public GameObject GetPooledObject(string name)
    {
        if (mainPool.ContainsKey(name))
        {
            tempList = mainPool[name] as List<GameObject>;

            for (int i = 0; i < tempList.Count; i++)
                if (tempList[i] != null)
                    if (!tempList[i].activeInHierarchy)
                        return tempList[i];

        }
        return null;
    }

    public void ResetPool()
    {
        for (int i = 0; i < tempList.Count; i++)
            if (tempList[i] != null)
                if (tempList[i].activeInHierarchy)
                    tempList[i].SetActive(false);
    }
}
```

Using other libraries

The **Pool Manager** uses the **System.Collections.Generic** library, which is at the top of the script.
So along with the usual **UnityEngine** and **System.Collections**, the **System.Collections.Generic** library is added like so:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

Inheriting from MonoBehaviour

Because we want to use the **Pool Manager** in the **Unity Editor** as a Component, we need to inherit from **MonoBehaviour**
The code for this is added to any script created in the **Unity Editor**, so you will have seen it before
See [MonoBehaviour - Manual]

```
public class PoolManager : MonoBehaviour {
```

Setting up the **properties**

Public properties

Properties we can **edit** in the **Unity Editor**, **declared** by using the **public keyword** at the **beginning** of a **property declaration**
Public properties are also available to **other scripts** to change

current

First we need a **static property**, so other scripts can easily access the **Pool Manager**
I *really* recommend you watching the video if you don't know what statics are!
See Statics - Video

```
public static PoolManager current;
```

names

Names, is a **built-in array** of type **string**
This will hold all the **names** of the **Pooled Objects**
You will **access** the **Pooled Objects** from **other scripts** using the **names** in this **array**
Please refer to the **Array Help Sheet** on the **DLE** for this week

Note: the **built-in array** is **declared** by using **square brackets** ([]) after the **type** (**string**)

```
public string[] names;
```

pooledOjects

We need an array of the Prefabs we are going to fill our Object Pool with
pooledObjects is that **array**, a **built-in array** of type **GameObject** (since we know all **Prefabs** are **GameObjects**!)
When the **Pool Manager** is asked for a **Pooled Object** like a **Bullet** or **Zombie**, it will return a **GameObject** from this **array**

```
public GameObject[] pooledObjects;
```

poolAmounts

We need to know **how many** Bullets, Zombies or any other **Pooled Object** to store in our **Object Pool**
These are stored in the **poolAmounts** array.
Once again, the **poolAmounts** is a **built-in array** of type **int**

```
public int[] poolAmounts;
```

Private properties

Properties only for use in the **class** they are **defined in**.
The **private properties** defined in the **PoolManager class** can only be used in the **Pool Manager class**

mainPool

A "super" array of type Hashtable that contains the 3 built-in arrays **names**, **pooledObjects** and **pooledAmounts**
mainPool is an **array of arrays**.

For more information about **Hashtables**, please refer to the **Array Help Sheet** on the **DLE** for this week

The reason we use a **Hashtable** is:
- We can mix **array types** (an array of type **string** and an array of type **int**)
- We can search the **Hashtable** using a **string** value, with the **ContainsKey()** method

```
private Hashtable mainPool = new Hashtable();
```

tempList

A temporary **list array** used to store a type of **Pooled Object** from the **Hashtable**.
e.g. If I wanted a **Bullet**, the **Pool Manager** would get all the **Bullets** from the **mainPool** array and store those in the **tempList** array,
the **tempList** array would then be **searched** for a **Bullet** that is **disabled** and **return** it ready for use

```
private List<GameObject> tempList;
```

Event functions

The event functions the Pool Manager uses are **Awake()** and **Start()**

Awake()

We need to set the **static** property **current** so other scripts can use it at **run time**
Awake() is ideal for this as it runs as soon as the game starts

Note the use of the **this** keyword.
this is shorthand for the **class referring to itself**
PoolManager is the **class** containing the code, so **this** is set to **PoolManager**

```
void Awake() {
    current = this;
}
```

Start()

Here, we setup our **mainPool Hashtable** with the **Names**, **pooledObjects** and **poolAmounts** set in the **Unity Editor**.

Here is the whole **Start()** method

```
void Start() {
    tempList = new List<GameObject>();

    for( int i = 0; i < names.Length; i++ ) {
        List<GameObject> objList = new List<GameObject>();

        for( int j = 0; j < poolAmounts[i]; j++ ) {
            GameObject obj = (GameObject)Instantiate( pooledObjects[i] );
            obj.SetActive( false );
            objList.Add( obj );
        }

        mainPool.Add( names[i], objList );
    }
}
```

Lets break down what's going on

This line **creates** the **tempList property**, which was **declared** as a **property** earlier

```
tempList = new List<GameObject>();
```

This is the beginning of a **for loop**.
We haven't covered these in any other scripts yet!
See Loops - video

The loop start with the **total amount** of **names** in the **names array** we created earlier
this value is obtained from the **Length property** of the **names built-in array**
For more information about the **Length property**, please refer to the **Array Help Sheet** on the **DLE** for this week

```
for( int i = 0; i < names.Length; i++ ) {
```

For each **iteration** of the **for loop**, we create a **list of GameObjects**
This list is temporary and will only exist until the end of the **Start()** method
We will use the list to copy the **GameObjects** into our **mainPool array**

```
List<GameObject> objList = new List<GameObject>();
```

We now use another for loop, an INNER loop
This is a loop INSIDE a loop.
We will fill our temporary **objList** array with **GameObjects** to pass into the **mainPool**
Note we are using the **poolAmounts** array
In the **Unity Editor**, this was where we entered how many **Bullets** or **Zombies** or **Explosions** we needed
This **loop creates** those **GameObjects** and adds them to the list

```
for( int j = 0; j < poolAmounts[i]; j++ ) {
```

This line creates the **GameObject** (called **obj**) and adds it to the **Hierarchy** using the **Instantiate()** method

```
GameObject obj = (GameObject)Instantiate( pooledObjects[i] );
```

This line **disables** the newly created **GameObject**

```
obj.SetActive( false );
```

Next the new **GameObject** is added to the **objList** array using the **Add()** method
to find out more about the **Add()** method on a **List array**, please refer to the **Array help sheet** on this weeks DLE

```
objList.Add( obj );
```

Next, we **close** the INNER for loop

```
}
```

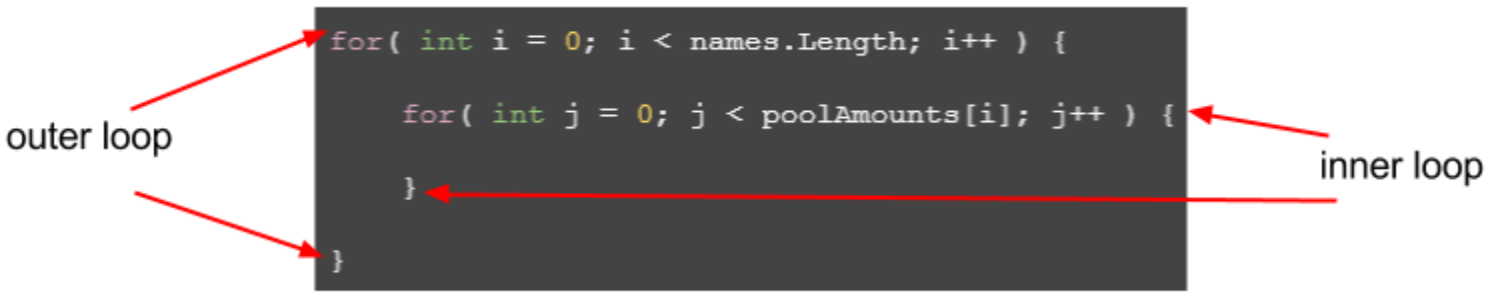
Now, in the OUTER for loop, we add our newly created list of **GameObjects** to the **mainPool**

```
mainPool.Add( names[i], objList );
```

Now, we **close** the OUTER for loop

```
}
```

Lets look at the loops and see how they are nested



Custom methods

The **Pool Manager** has **2 custom methods** which can be accessed by other scripts, **GetPooledObject** and **RestPool.**

GetPooledObject()

This method **returns** a **Pooled Object** from a **string name input**.

Signature

This method is **public**, so it's accessible from other scripts
It will return a **GameObject** or **null** if there are no **Pooled Objects** available
It needs a **name** of type **string** that matches a **value** in the **names** array, otherwise it will **return null**

```
public GameObject GetPooledObject( string name )
```

Usage

If you wanted to get a **Bullet** from the **Pool Manager** (providing you set it up with one), you would enter this code

```
GameObject myBullet = PoolManager.current.GetPooledObject( "Bullet" );
```

Code

Lets go through the code for this method.
Here is the entire method code:

```
public GameObject GetPooledObject( string name ) {
    if( mainPool.ContainsKey( name ) ) {
        tempList = mainPool[ name ] as List<GameObject>;

        for( int i = 0; i < tempList.Count; i++ ) {
            if( tempList[i] != null ) {
                if( !tempList[i].activeInHierarchy ) {
                    return tempList[i];
                }
            }
        }
    }
    return null;
}
```

First line:

```
if( mainPool.ContainsKey( name ) ) {
```

This uses the **ContainsKey()** method to search for the **name** given by the requesting script (like a zombie spawner)
If the name is found, the code proceeds:

```
public GameObject GetPooledObject( string name ) {
    if( mainPool.ContainsKey( name ) ) {


    }
    return null;
}
```

If the **name** is found in the **mainPool** using **Containskey** run the code in the curly braces

Next, within the if statement above:

This line stores all the **PooledObjects** with the **name** in the **tempList** array
If you asked for **Bullets**, the **tempList** array will have a **list** of all those **Bullets** now.

```
tempList = mainPool[ name ] as List<GameObject>;
```

The code **mainPool[name]** returns all values matching **name.**
these are stored in **tempList** as a **List array**

Next is a **for loop** which will go through the newly filled **tempList**, searching for a **disabled GameObject** to **return**

```
for( int i = 0; i < tempList.Count; i++ ) {
```

Within the **for loop**, we first need to check each **GameObject** exists (is not **null**)
This check is necessary because we may have not added our **Prefab** to the **pooledObjects array**,(in the **Unity Editor**) so it will have **no value** (**null**)

```
if( tempList[i] != null ) {
```

If the **GameObject** does exist, we can now check if that **GameObject** is **disabled**
We can use the **activeInHierarchy property** of the **GameObject** to check if it is **active** (or **enabled**) in the **Hierarchy**
Note the use of the exclamation mark (**!**) which means **not true**

```
if( !tempList[i].activeInHierarchy ) {
```

See GameObject.activeInhierarchy
See ! Operator - MSDN Manual

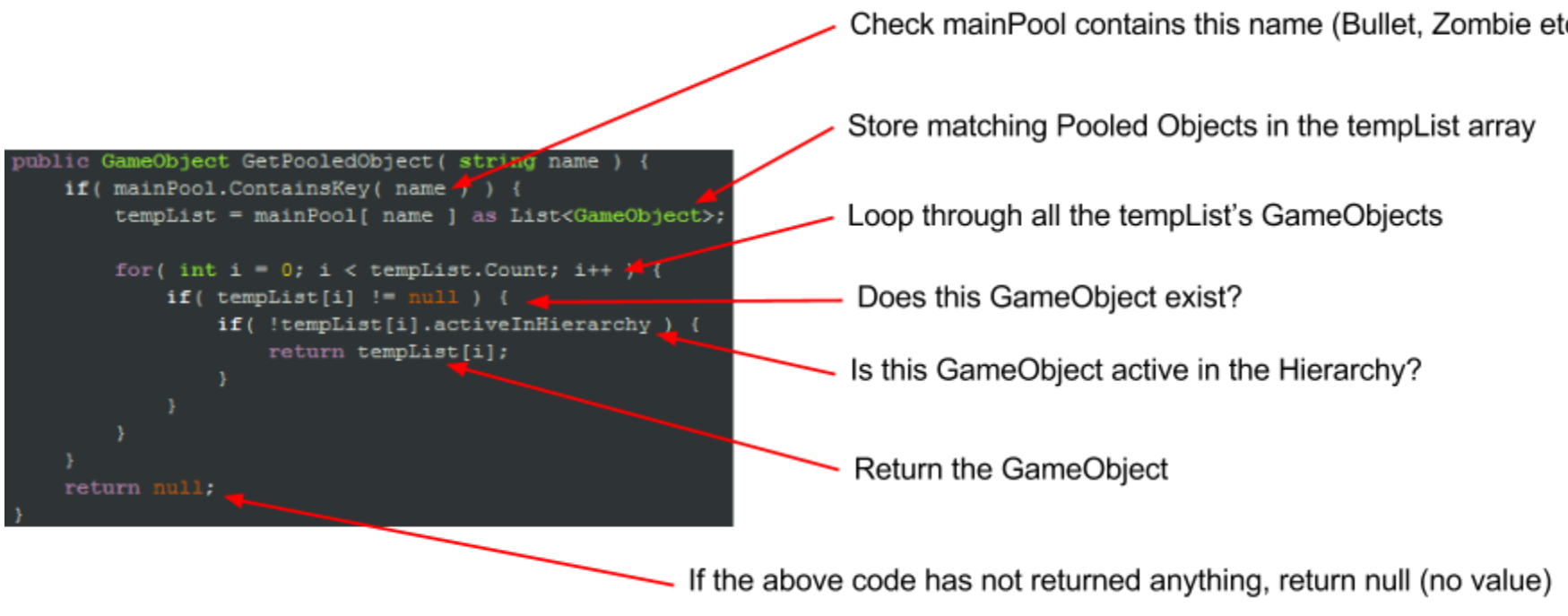
If it is **disabled**, we can **return** the **GameObject** and exit out from this method using the **return statement**

```
return tempList[i];
```

Finally, just before the end of the method, AFTER the preceding code we will return a null.
If the all the previous checks fail, this line will run, returning a null

```
return null;
```

In summary



ResetPool()

This method **resets** all the **GameObjects** in the **Object Pool**.
It will check if each **GameOject** is active in the **Hierarchy**
If it is, it will **disable** it

Signature

This method is **public**, so it is accessible from other scripts.
It **returns nothing**, using the **void** type
It requires no input, so no **parameters** are listed within the **brackets**

```
public void ResetPool()
```

Usage

If you wanted to reset your Object Pool, you would use this code in your script

```
PoolManager.current.ResetPool();
```

Code

Lets go through the code for this method.
Here is the entire method code:

```
public void ResetPool() {
    for( int i = 0; i < tempList.Count; i++ ) {
        if( tempList[i] != null ) {
            if( tempList[i].activeInHierarchy ) {
                tempList[i].SetActive( false );
            }
        }
    }
}
```

First line:
We **loop** through the **names array** to get each **type** of **Pooled Object** (**Bullets**, **Zombies** etc)

```
for( int i = 0; i < names.Count; i++ ) {
```

Next we store our matching **names** from the **mainPool Hashtable** in our **tempList array**

```
tempList = mainPool[ names[i] ] as List<GameObject>;
```

Next we go through our newly filled **tempList array** with a **for loop**

```
for( int j = 0; j < tempList.Count; j++ ) {
```

Check if the **GameObject** at index **tempList[i]** exists

```
if( tempList[j] != null ) {
```

Check if the **GameObject** at index **tempList[i]** is active in the **Hierarchy**

```
if( tempList[j].activeInHierarchy ) {
```

If the **GameObject** does exist and is **active** in the **Hierarchy** (**enabled**), then **disable** it

```
tempList[j].SetActive( false );
```

In summary

```
public void ResetPool() {

    for( int i = 0; i < names.Length; i++ ) {

        tempList = mainPool[ names[i] ] as List<GameObject>;

        for( int j = 0; j < tempList.Count; j++ ) {

            if( tempList[j] != null ) {

                if( tempList[j].activeInHierarchy ) {

                    tempList[j].SetActive( false );
                }
            }
        }
    }
}
```

Loop through our names array

Store our Pooled Object type in tempList (Bullet, Zombie etc)

Loop through our newly filled tempList array

If the GameObject stored at tempList[j] is not null (it exists)

if the GameOject is active in the Hierarchy

Enable the GameObject using SetActive