UNITY: MONO-SERVICE LOW-LEVEL DOCUMENTATION

BY: HUMANITARIAN OPERATIONS

CONTENTS

OVERVIEW:

This documentation will allow the new system’s users to add more services and to also maintain it.

Note: you will need to read through the high-level documentation before start reading this.

REQUIREMENTS:

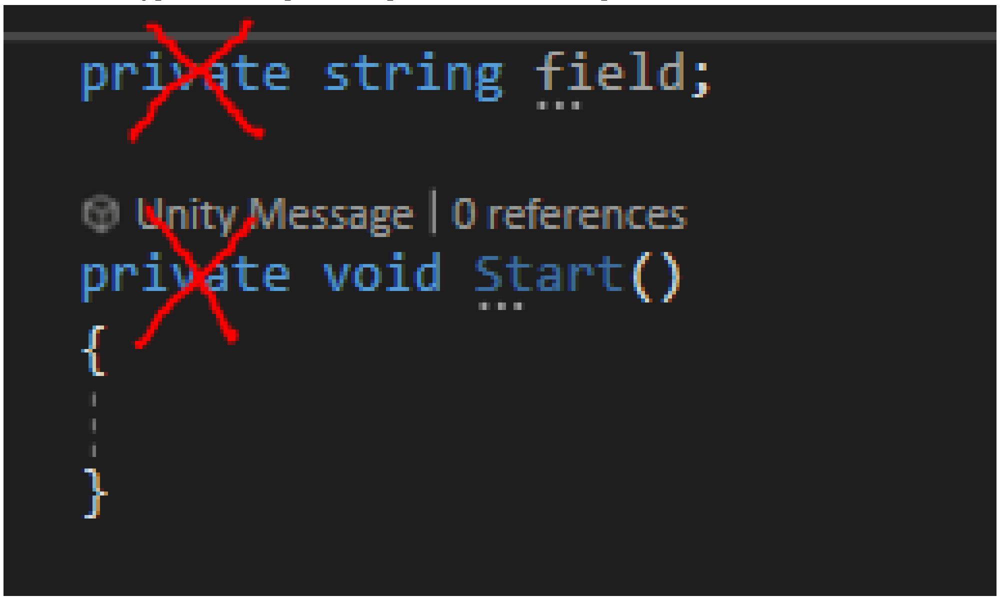
For you to become a maintainer of this system, you will need to know the basics of using Unity:

- All the previous requirements from the Hight Level Documentation.
- Strong understanding of C# and its SOLID principles.
- Strong understanding of the different design pattern, especially Command and Factory Patterns.
- Good understanding of Unity and its limitations.

CODING STANDARDS:

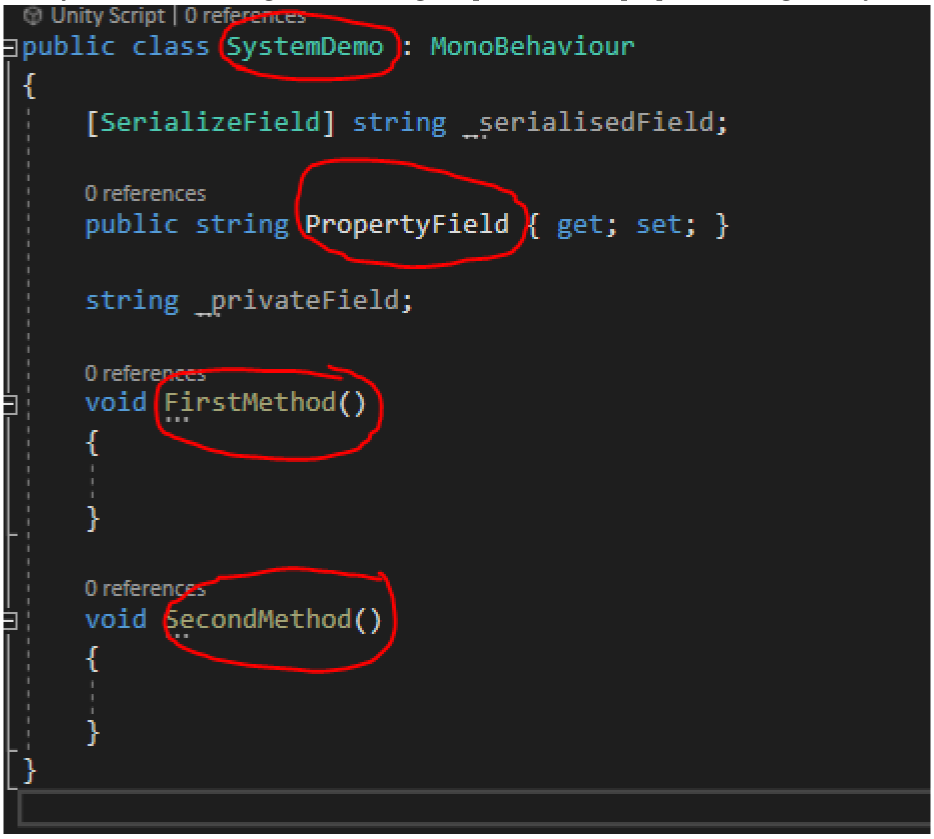
We’re using a specific format when writing code, please follow this format for consistency:

- Never use the private keywords when naming fields and methods, it just useless information, if a certain data type is nether public or protected, it will be private:



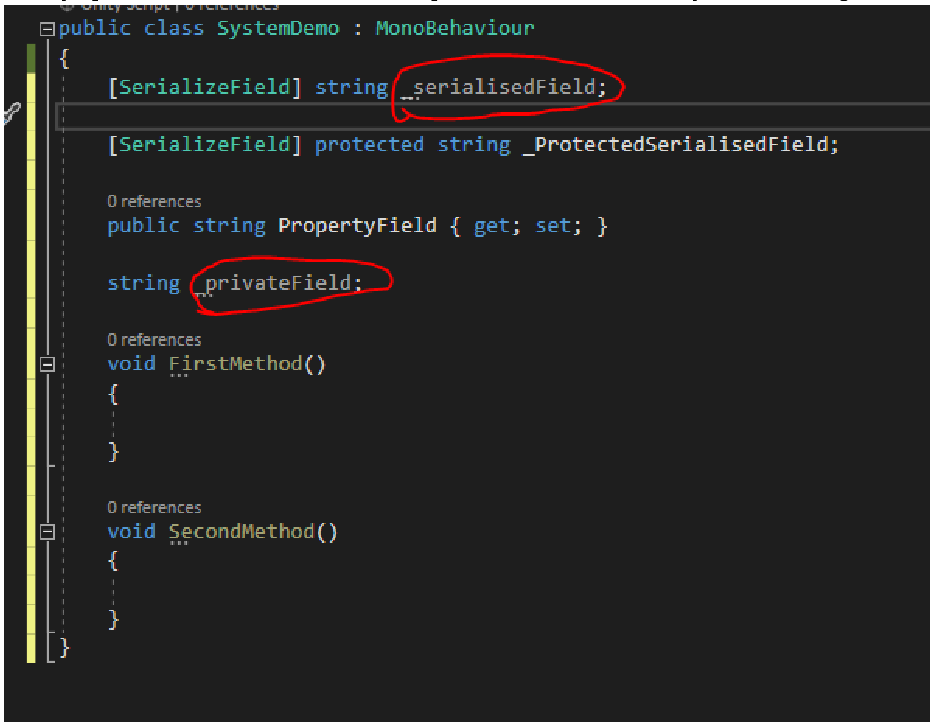
```
private string field;

Unity Message | 0 references
private void Start()
{

}
```

- Always use PascalCasing when naming scripts, methods, properties and gameobjects in the scene:



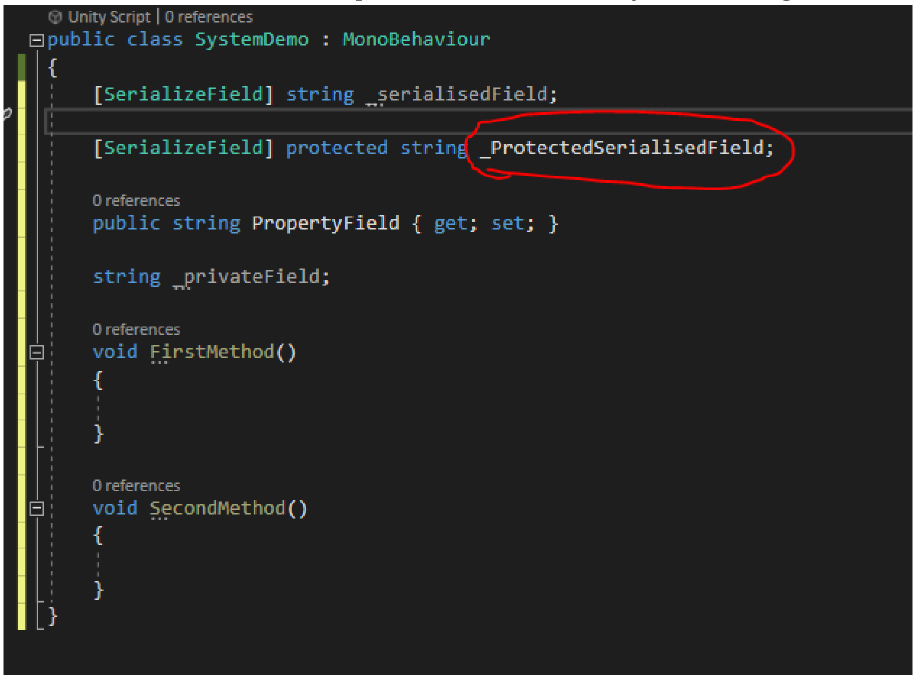
```
Unity Script | 0 references
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;

    0 references
    public string PropertyField { get; set; }

    string _privateField;

    0 references
    void FirstMethod()
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```

- Always put an undersore in front of the private fields followed by camel casing:



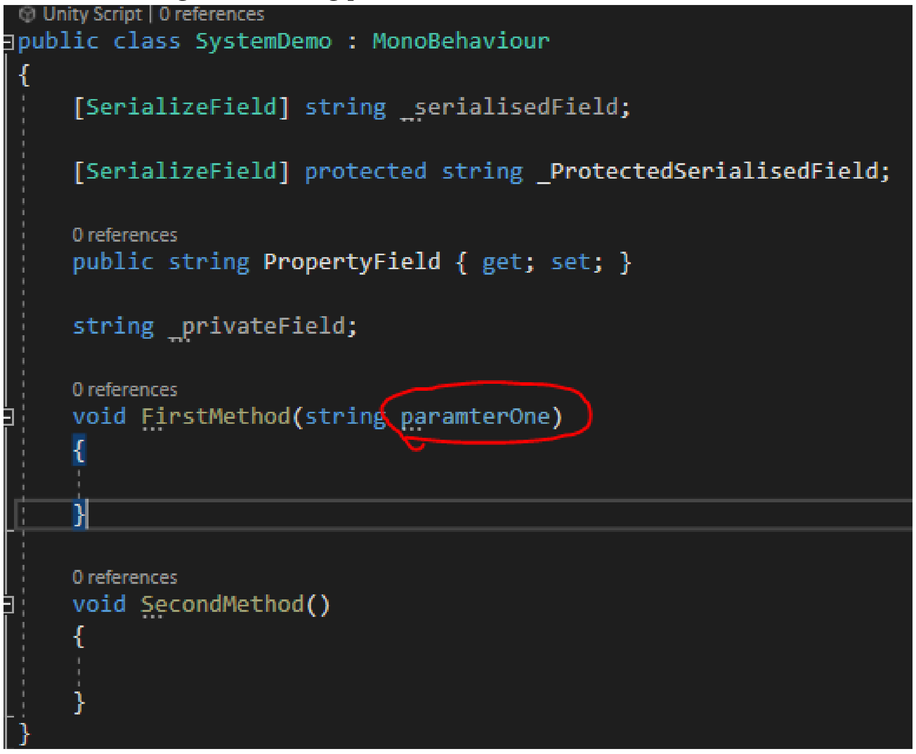
```
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;

    [SerializeField] protected string _ProtectedSerialisedField;

    0 references
    public string PropertyField { get; set; }

    string _privateField;

    0 references
    void FirstMethod()
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```

- Put an underscore in front of the protected field followed by PascalCasing:



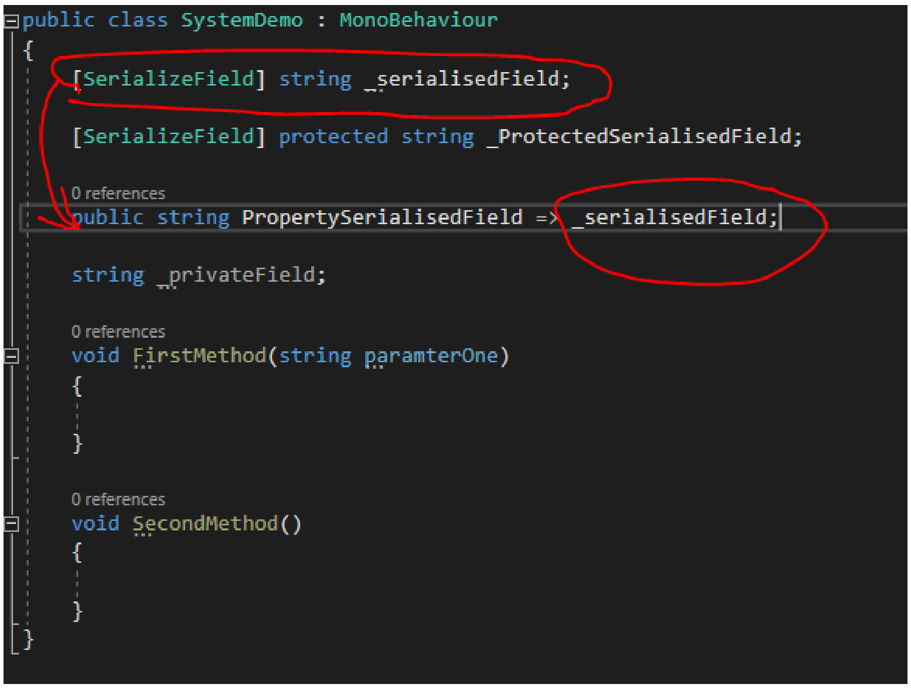
```
Unity Script | 0 references
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;

    [SerializeField] protected string _ProtectedSerialisedField;

    0 references
    public string PropertyField { get; set; }

    string _privateField;

    0 references
    void FirstMethod()
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```

- Use camelCasing when naming parameters:

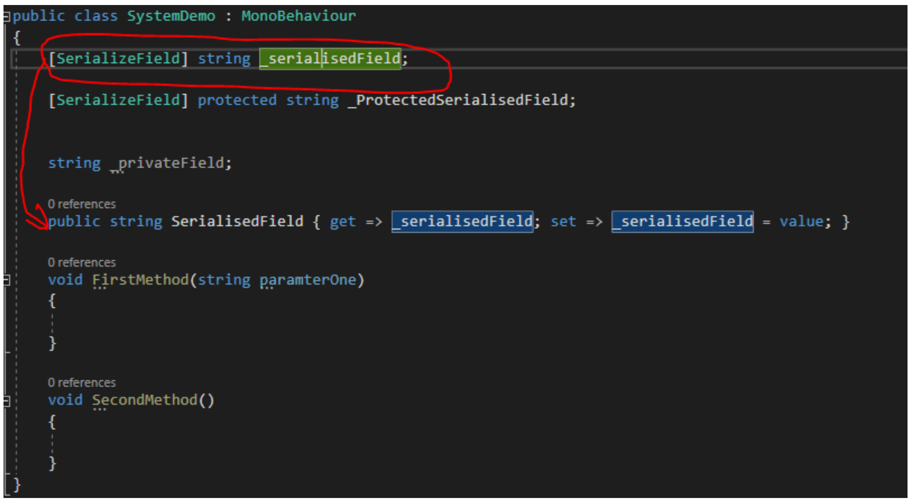


```
Unity Script | 0 references
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;

    [SerializeField] protected string _ProtectedSerialisedField;

    0 references
    public string PropertyField { get; set; }

    string _privateField;

    0 references
    void FirstMethod(string paramterOne)
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```

- Never make fields public, make them private or protected and encapculate them into a public property, therefore you can make it read only or read and write: the first example is read only and the second example is read and write:

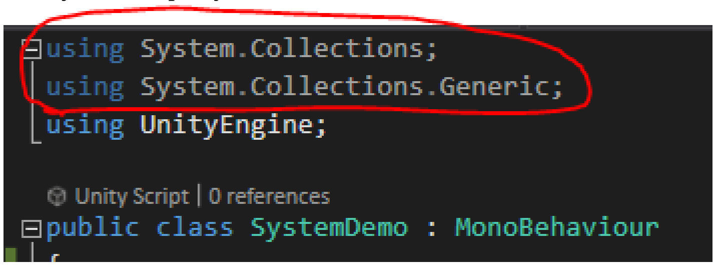


```
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;

    [SerializeField] protected string _ProtectedSerialisedField;

    0 references
    public string PropertySerialisedField => _serialisedField;

    string _privateField;

    0 references
    void FirstMethod(string paramterOne)
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```



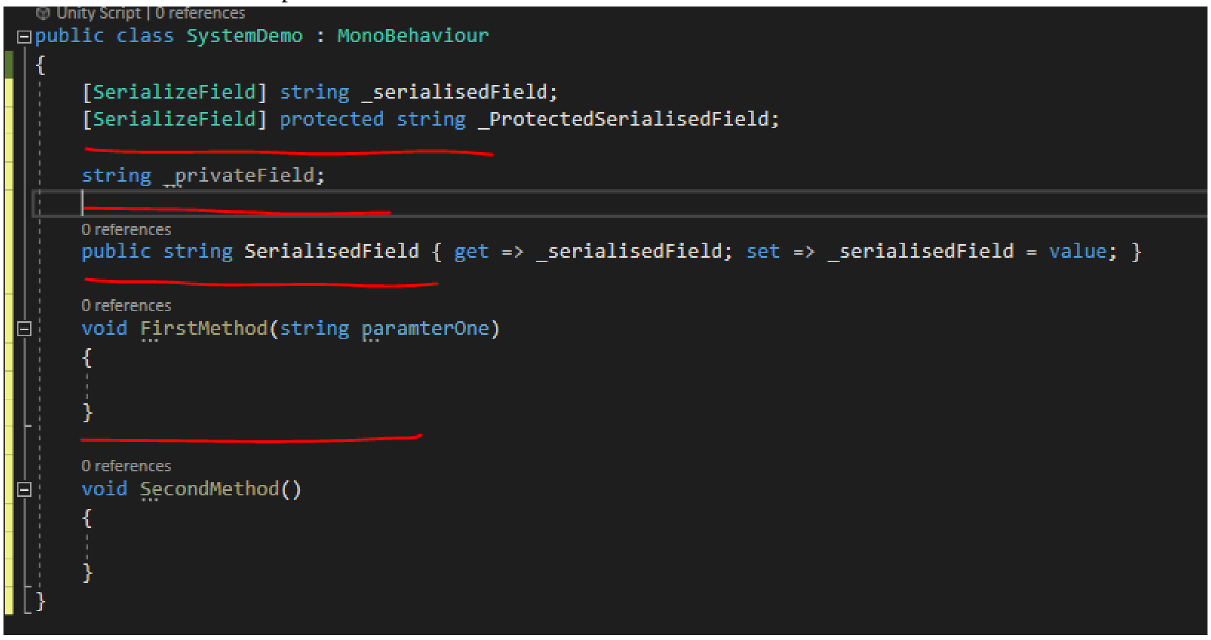
```
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;

    [SerializeField] protected string _ProtectedSerialisedField;

    string _privateField;

    0 references
    public string SerialisedField { get => _serialisedField; set => _serialisedField = value; }

    0 references
    void FirstMethod(string paramterOne)
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```

- Please be more considerate when naming classes, fields and methods by making them more descpritive and informative. I should know what is the class is reponsible for just by reading its name.
- Avoid using short words when naming like system “sys” unless the term is globally known.
- Always clean up any unsed libraries, methods or fields:



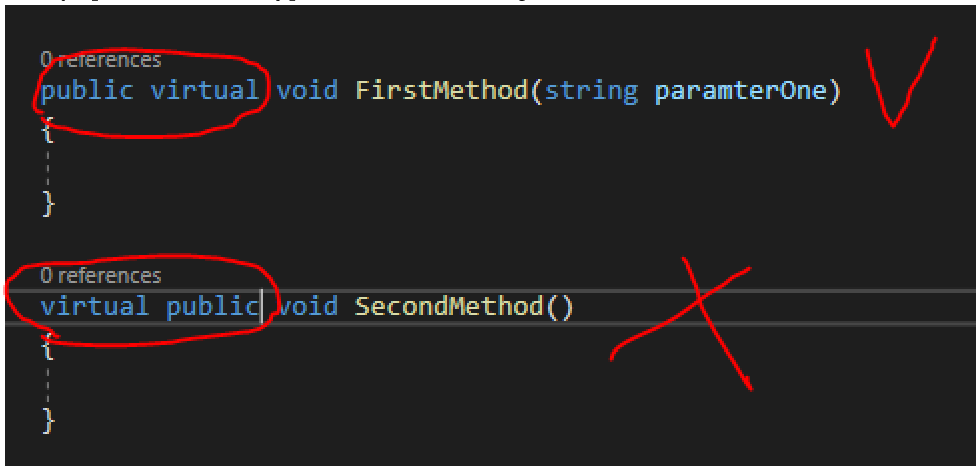
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script | 0 references
public class SystemDemo : MonoBehaviour
{
```

- Serialised fields should always be on the top of the class, properties second, private fields third and methods last. Also leave space between them:



```
Unity Script | 0 references
public class SystemDemo : MonoBehaviour
{
    [SerializeField] string _serialisedField;
    [SerializeField] protected string _ProtectedSerialisedField;

    string _privateField;

    0 references
    public string SerialisedField { get => _serialisedField; set => _serialisedField = value; }

    0 references
    void FirstMethod(string paramterOne)
    {

    }

    0 references
    void SecondMethod()
    {

    }
}
```

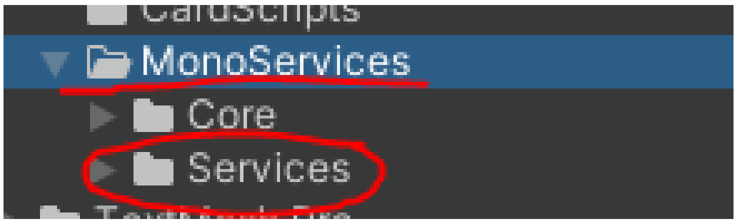
- Avoid using update and use coroutines instead so you can pause them or stop them.
- Never use classes to manage other gameobjects, every class should only be reponsible for the gameobject that is attached to and its children.
- Avoid using comments, you code should be self documented meaning that the names of the class, fields and methods are informative enough.
- Always place the access type in front when using virtual or overridden methods:



```
0 references
public virtual void FirstMethod(string paramterOne)
{

}

0 references
virtual public void SecondMethod()
{

}
```

SYSTEM WORKFLOW:

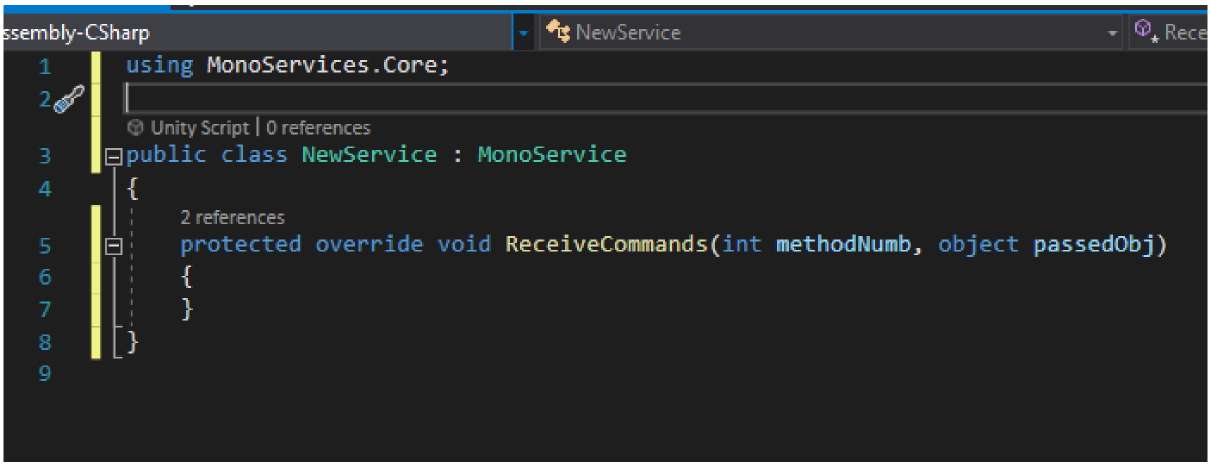
Most unity components have extra services that are available in the services folder within the MonoServices Root folder:



When you create a new service, please add it to the services folder.

Note: never add a new sevice before checking if that service already exist to avoid redundancy.

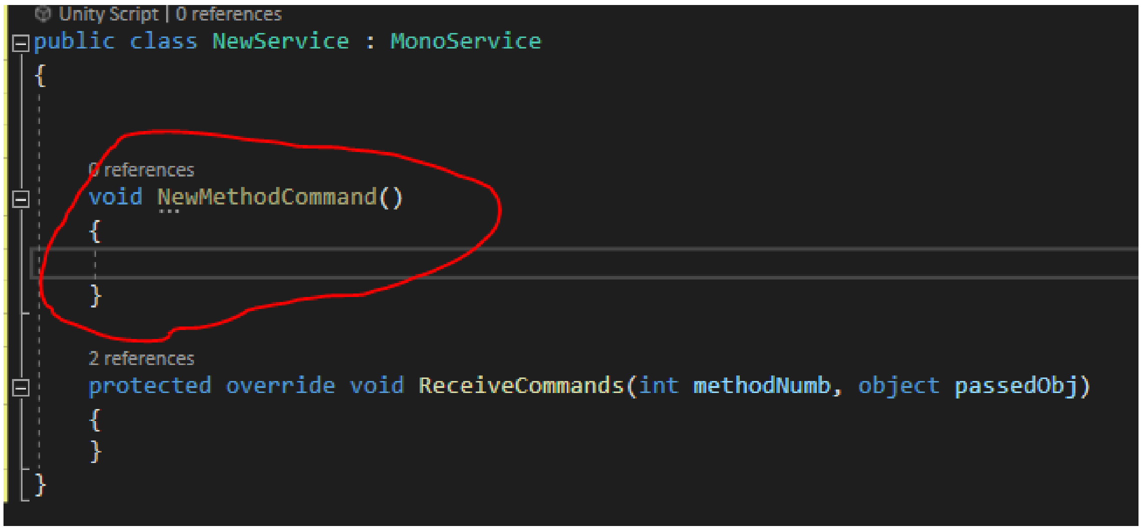
When creating a new service, it should always inherent for the MonoService class , add the MonoService.Core Namespace and implement the abstract class:



```
using MonoServices.Core;

public class NewService : MonoService
{
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
    }
}
```

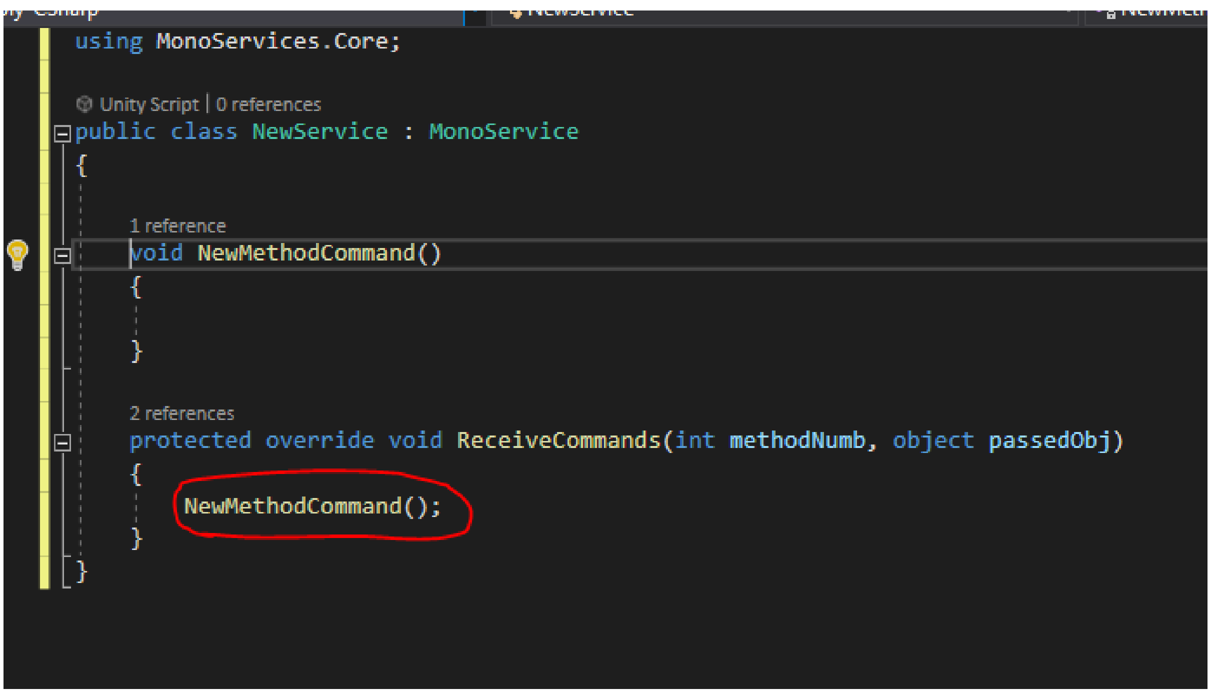
As menthioned in the high level documentation, services can invoke and receive commands.

To add a new receive command, write a new method but it must end with the “Command” key word for it to show up as an invoker command:



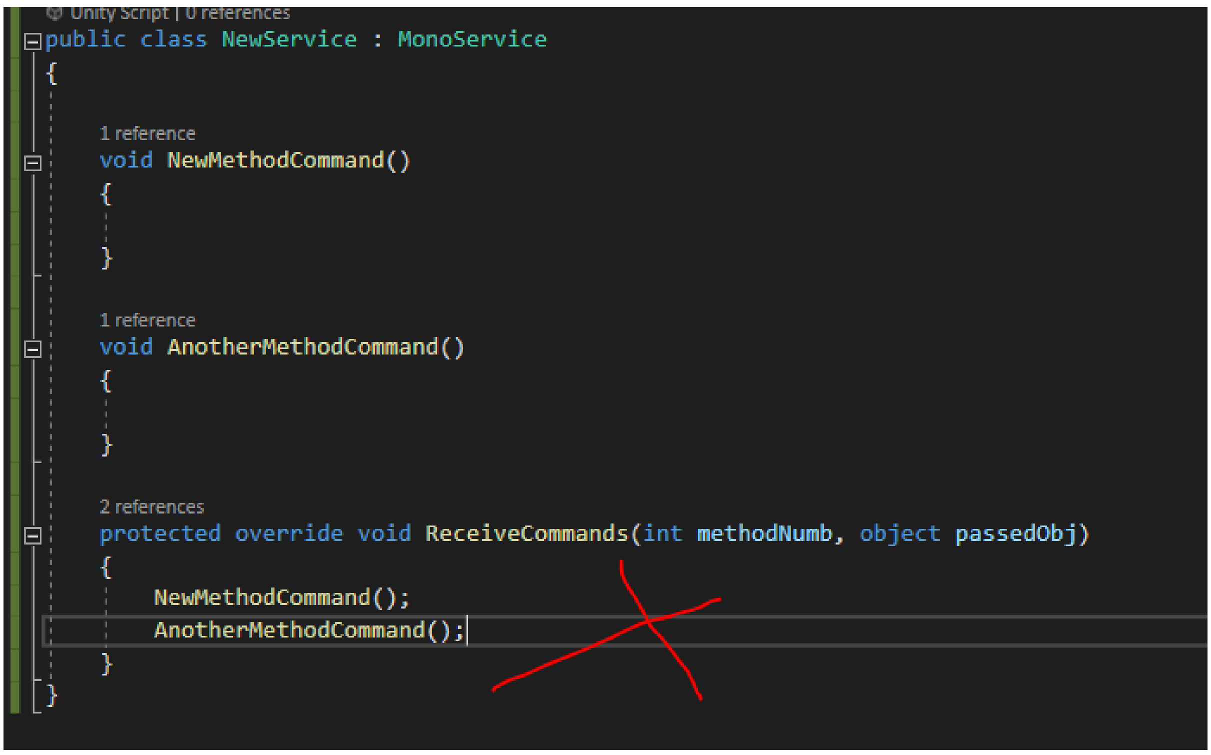
```
public class NewService : MonoService
{

    void NewMethodCommand()
    {

    }

    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
    }
}
```

For the new method to be called, it needs to be added to the receive commands method:



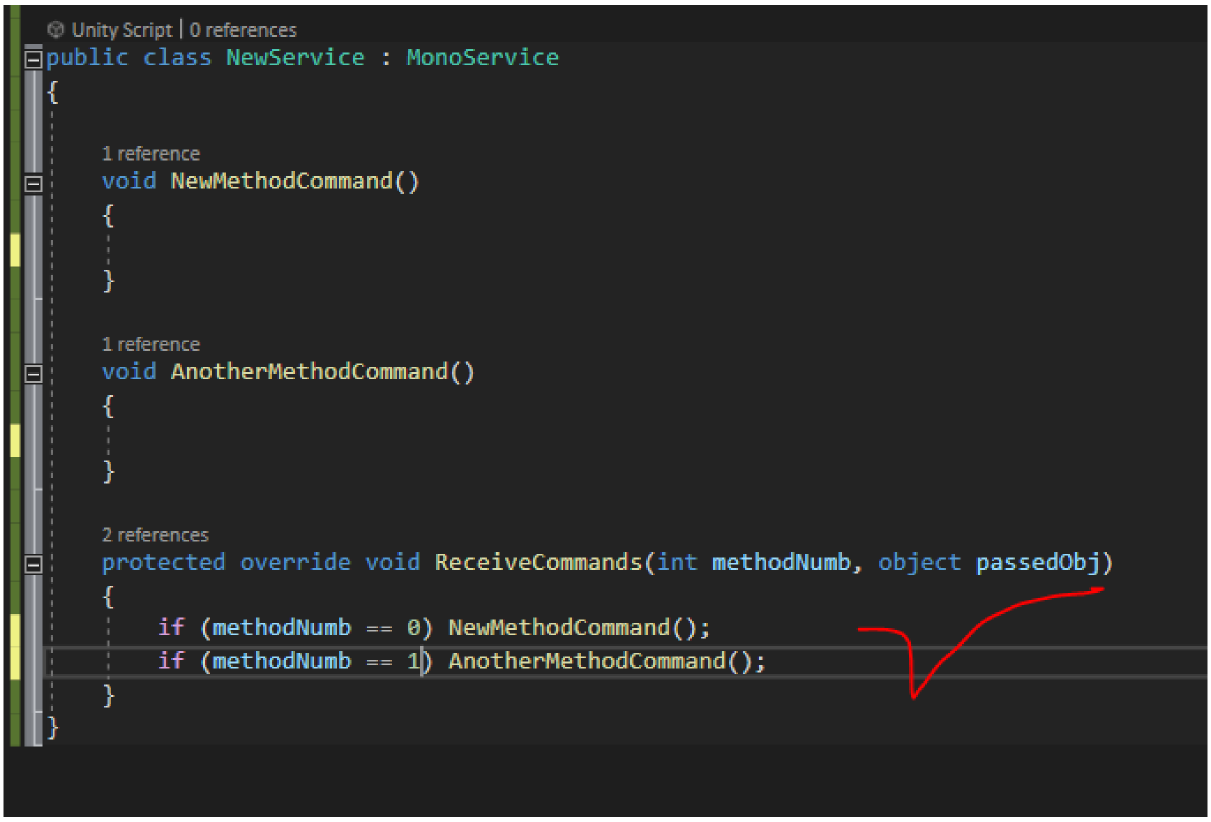
```
using MonoServices.Core;

Unity Script | 0 references
public class NewService : MonoService
{

    1 reference
    void NewMethodCommand()
    {

    }

    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
        NewMethodCommand();
    }
}
```

But what if you had 2 methods to call? You can’t just call them at the same time:



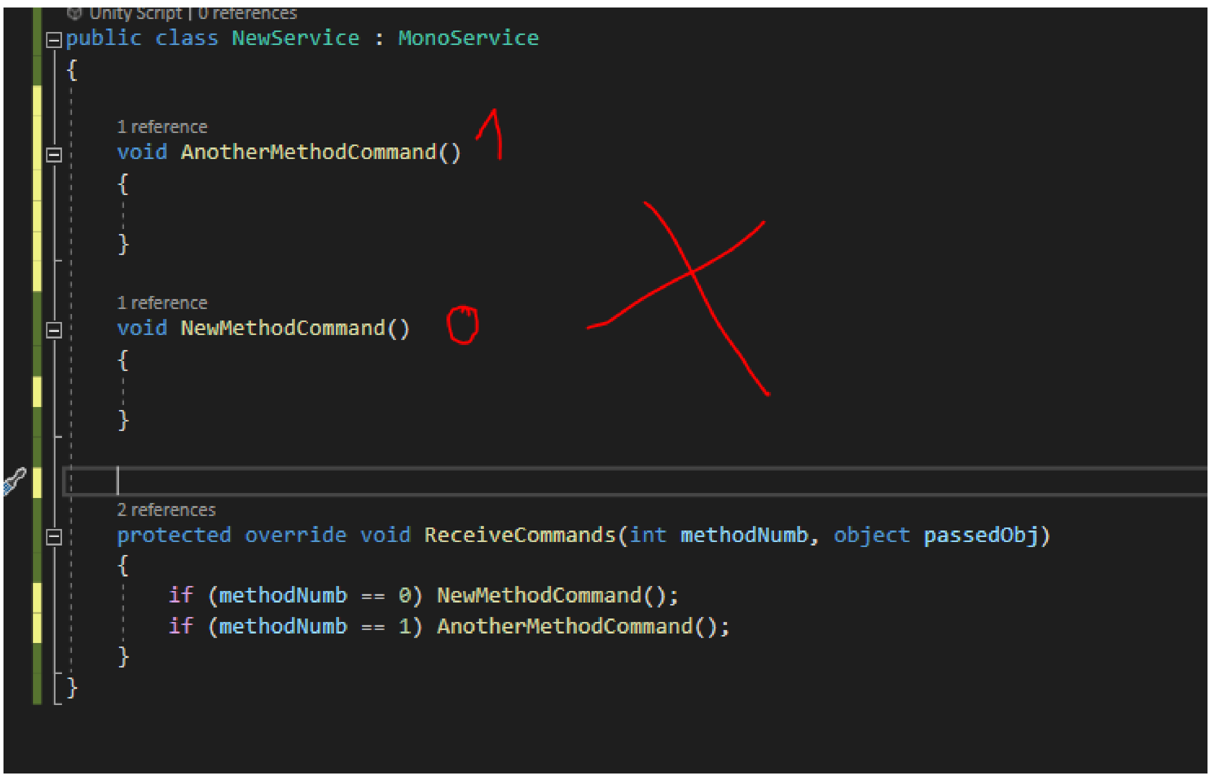
```
Unity Script | 0 references
public class NewService : MonoService
{

    1 reference
    void NewMethodCommand()
    {

    }

    1 reference
    void AnotherMethodCommand()
    {

    }

    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
        NewMethodCommand();
        AnotherMethodCommand();
    }
}
```

That’s what the methodNumb is used for, you could check if the method number is 0 or 1:



```
Unity Script | 0 references
public class NewService : MonoService
{

    1 reference
    void NewMethodCommand()
    {

    }

    1 reference
    void AnotherMethodCommand()
    {

    }

    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
        if (methodNumb == 0) NewMethodCommand();
        if (methodNumb == 1) AnotherMethodCommand();
    }
}
```

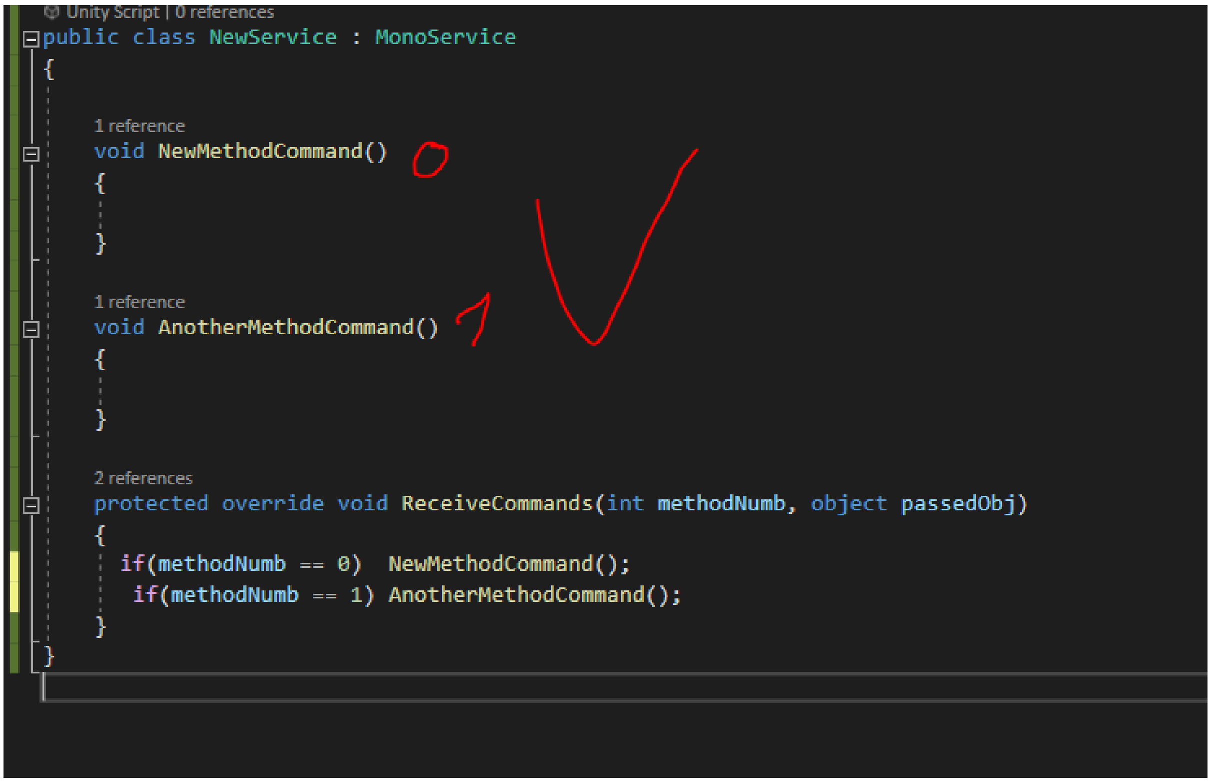
Note: the order of the methods is important, it won't work if the AnotherMethodCommand is before the NewMethodCommand:



```
Unity Script | 0 references
public class NewService : MonoService
{

    1 reference
    void AnotherMethodCommand()
    {

    }

    1 reference
    void NewMethodCommand()
    {

    }


    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
        if (methodNumb == 0) NewMethodCommand();
        if (methodNumb == 1) AnotherMethodCommand();
    }
}
```

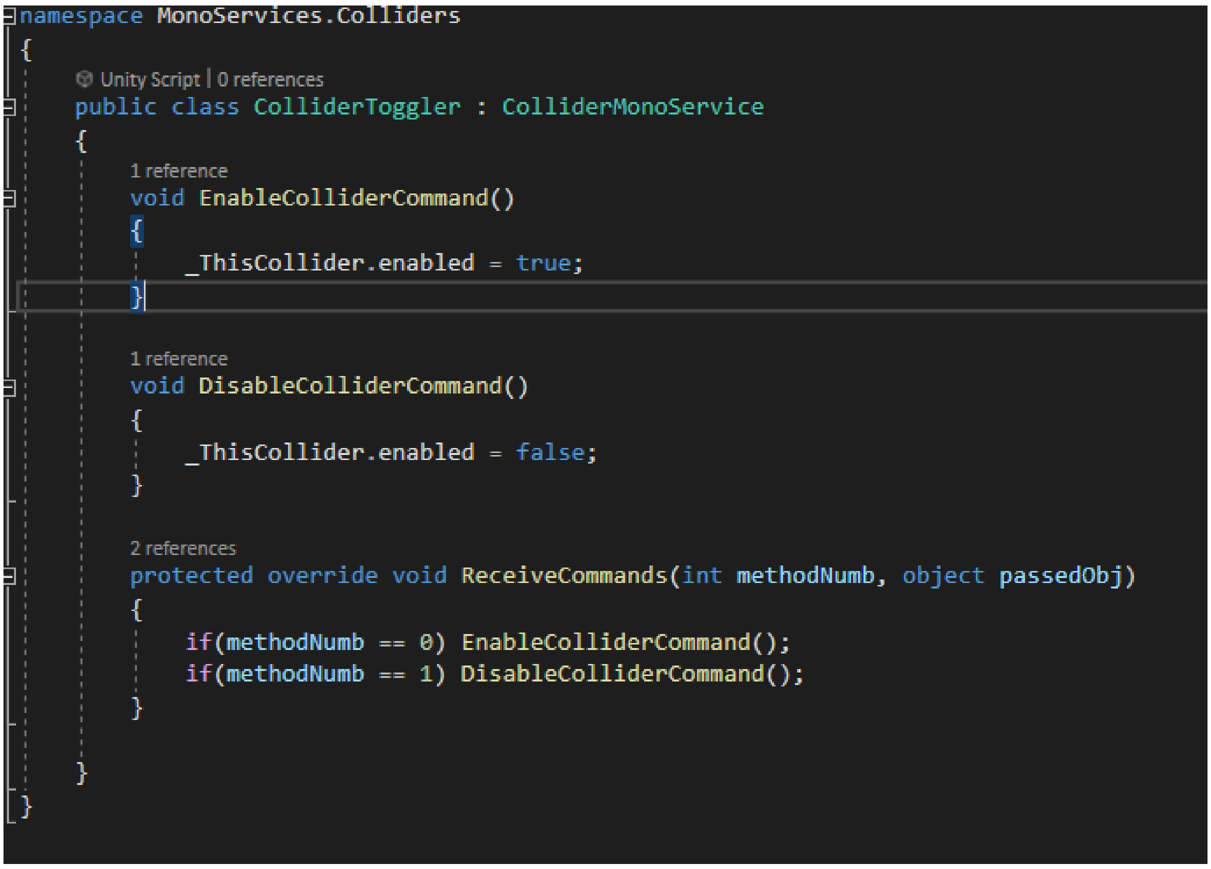
1

0



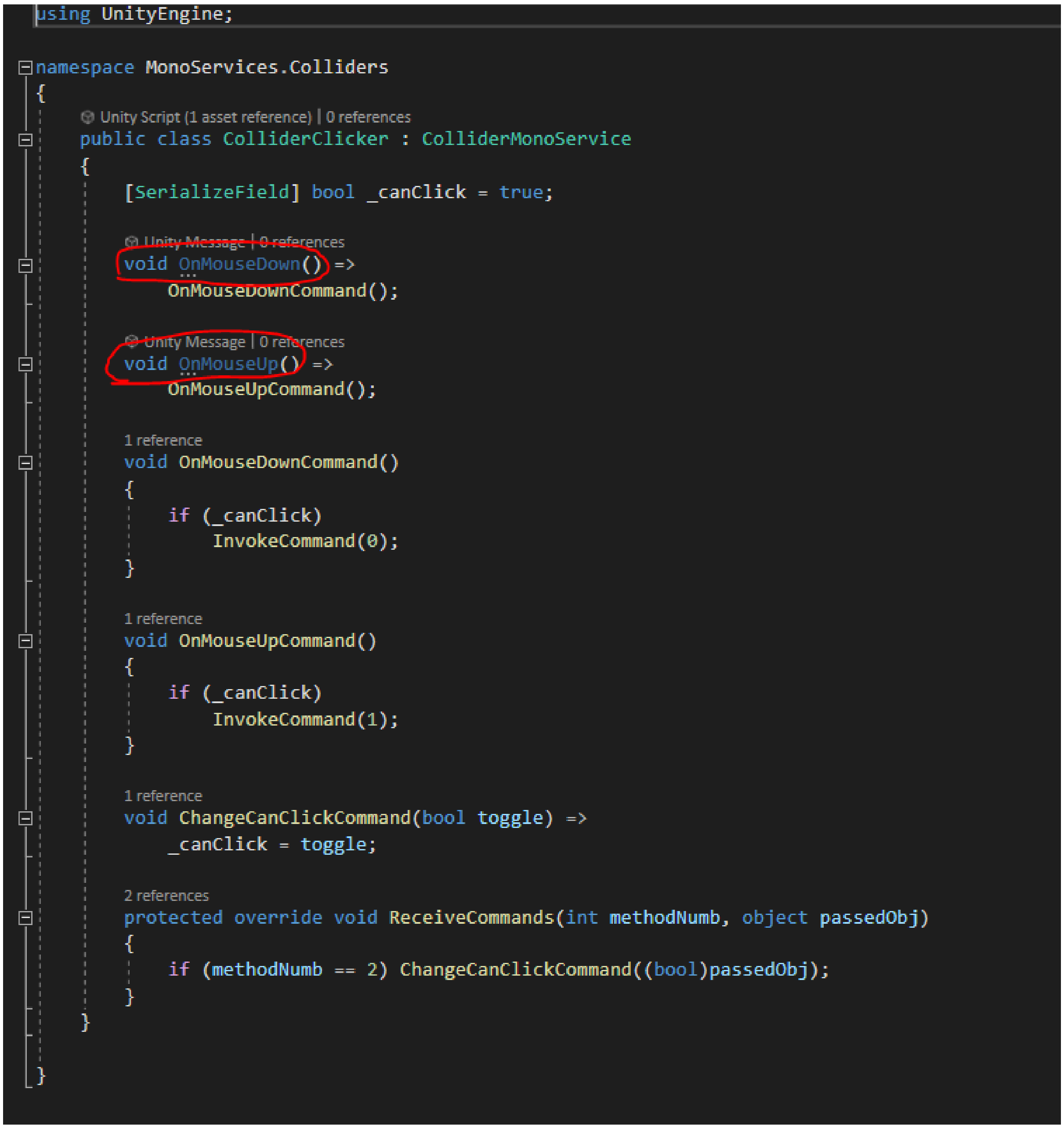
```
Unity Script | 0 references
public class NewService : MonoService
{

    1 reference
    void NewMethodCommand()
    {

    }

    1 reference
    void AnotherMethodCommand()
    {

    }

    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
      if(methodNumb == 0)  NewMethodCommand();
       if(methodNumb == 1) AnotherMethodCommand();
    }
}
```

This is an example for turning a collider on and off:



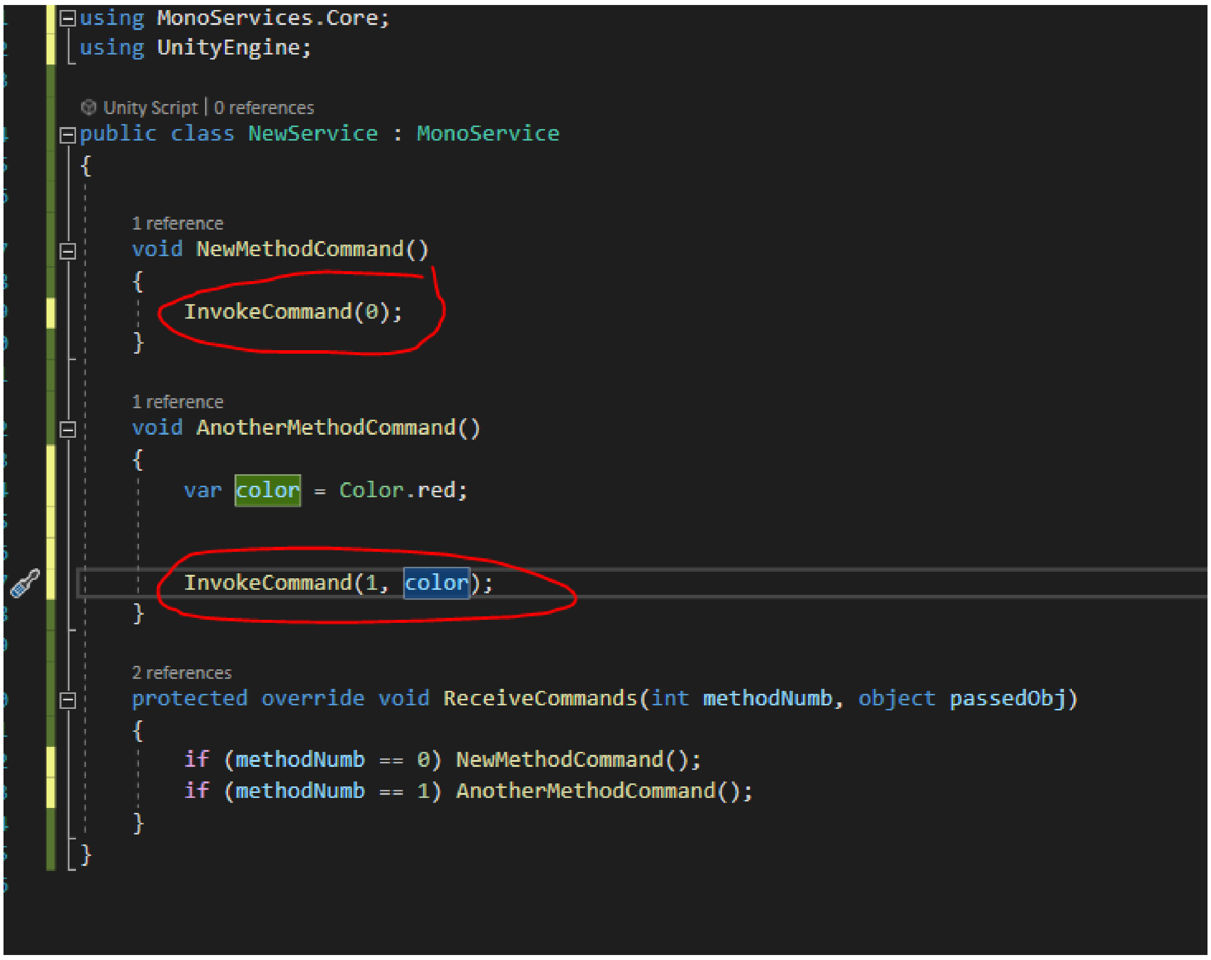
```
namespace MonoServices.Colliders
{
    Unity Script | 0 references
    public class ColliderToggler : ColliderMonoService
    {
        1 reference
        void EnableColliderCommand()
        {
            _ThisCollider.enabled = true;
        }

        1 reference
        void DisableColliderCommand()
        {
            _ThisCollider.enabled = false;
        }

        2 references
        protected override void ReceiveCommands(int methodNumb, object passedObj)
        {
            if(methodNumb == 0) EnableColliderCommand();
            if(methodNumb == 1) DisableColliderCommand();
        }

    }
}
```

In some case, commands don’t need to be received, like the on mouse down event for example, the command will be listening to that event:



```
using UnityEngine;

namespace MonoServices.Colliders
{
    Unity Script (1 asset reference) | 0 references
    public class ColliderClicker : ColliderMonoService
    {
        [SerializeField] bool _canClick = true;

        Unity Message | 0 references
        void OnMouseDown() =>
            OnMouseDownCommand();

        Unity Message | 0 references
        void OnMouseUp() =>
            OnMouseUpCommand();

        1 reference
        void OnMouseDownCommand()
        {
            if (_canClick)
                InvokeCommand(0);
        }

        1 reference
        void OnMouseUpCommand()
        {
            if (_canClick)
                InvokeCommand(1);
        }

        1 reference
        void ChangeCanClickCommand(bool toggle) =>
            _canClick = toggle;

        2 references
        protected override void ReceiveCommands(int methodNumb, object passedObj)
        {
            if (methodNumb == 2) ChangeCanClickCommand((bool)passedObj);
        }
    }
}
```

Now usually receive commands can also notify other monoservices that it has been called, to do this you must call the protected invoke command method and pass in the method number plus if you would like to pass an object with it, here I chose to pass a colour in the second method:

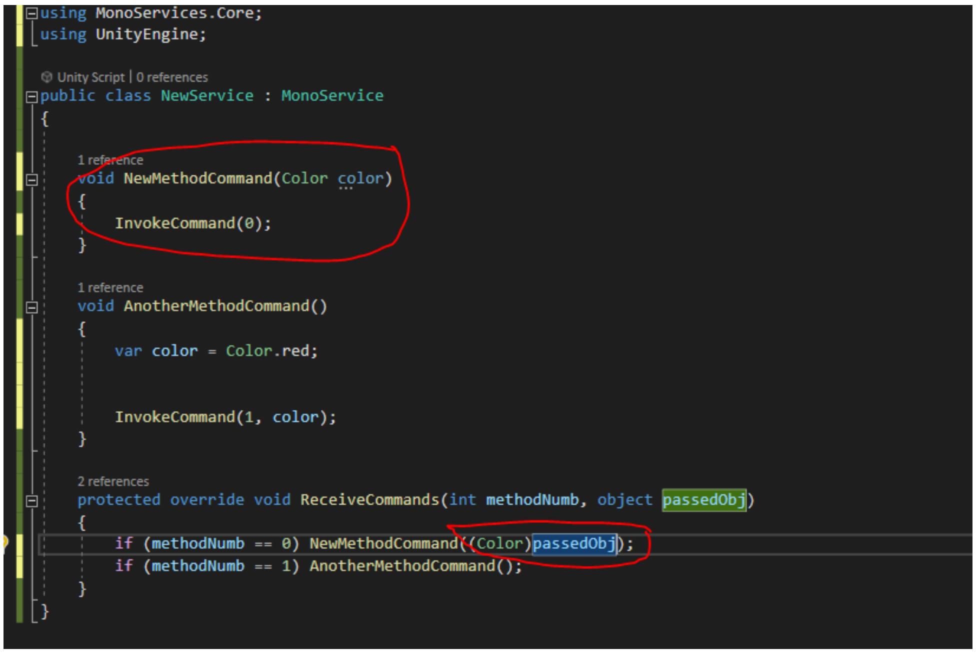


```
using MonoServices.Core;
using UnityEngine;

Unity Script | 0 references
public class NewService : MonoService
{
    1 reference
    void NewMethodCommand()
    {
        InvokeCommand(0);
    }

    1 reference
    void AnotherMethodCommand()
    {
        var color = Color.red;

        InvokeCommand(1, color);
    }

    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
        if (methodNumb == 0) NewMethodCommand();
        if (methodNumb == 1) AnotherMethodCommand();
    }
}
```

Note: Only use the InvokeCommand method when you want other commands to listen to this command.

InvokeCommand method is to notify other mono services that "AnotherMethodCommand" have been called.

You can also pass objects, a colour for a example, all you have to do is to cast the passedObj parameter to a colour and then call the method that contains the colour as a parameter:



```
using MonoServices.Core;
using UnityEngine;

Unity Script | 0 references
public class NewService : MonoService
{
    1 reference
    void NewMethodCommand(Color color)
    {
        InvokeCommand(0);
    }

    1 reference
    void AnotherMethodCommand()
    {
        var color = Color.red;

        InvokeCommand(1, color);
    }

    2 references
    protected override void ReceiveCommands(int methodNumb, object passedObj)
    {
        if (methodNumb == 0) NewMethodCommand((Color)passedObj);
        if (methodNumb == 1) AnotherMethodCommand();
    }
}
```

- Note: instead of using find object of type or tag, implement a gameobject getter class and pass in the gameobject as a parameter, but you wouldn’t necessarily need to find other objects in the scene if every class is only responsible for the gameobject that is attached to.

The whole system is basically following the command design pattern.