Service Locator

What it does?

In games, we have a lot of services running in background. Therefore it acts as a centralized place from where can retrieve that service, rather than referencing it everytime.

Why do we need it ?

Some good things :

1) Besides having a private constructor, we can still make references outside of class with the addition of Secure Allocation.
2) Works for Monobehaviour and non-Monobehaviour classes
3) Time Complexity is better than GameObject.FindGameObjectOfType<?>();
4) Does more than just storing services
   - ↳ Create if not exists, while retrieving that service.
   - ↳ Override any existing service {optional}
   - ↳ Destroys upon Application Quit

Why don't just use Singleton ?

A game does not have only one service, therefore using singleton with multiple classes is a bad practice. Instead referencing them in such a way where the reference exposure is minimal and having only a single reference throughout the game is like something we all think of achieving.

What kind and How many services can be stored ?

Any kind of Service can be stored, since this is a Generic Service Locator.
There is no limit set for amount of services can be stored.

Does it require any pre-requisite ?

Not much. In order to use this service locator

- ↳ A Monobehaviour class needs to implement an interface
- ↳ A Non-Monobehaviour class needs to inherit a class which is just being used for identifying whether the service is valid or not.
- ↳ Non-Monobehaviour service needs to have a function explicitly.

A Deep Dive in Code

The interfaces.

```
// Interface for Service Locator itself
public interface IProvideService
    {
        public void PushService<T>(T service, bool canOverwrite) where T : class;   // Register a Service
        public T PullService<T>(bool createIfNULL) where T : class;  // Retrieve a Service

        public void ClearService<T>(); // Clear a specific registered service
        public void ClearAllServices(); // Clear all register services
        public bool PeekService<T>();  // Check if a service is already registered

    }
```

Service Reference

Overwrite if already an alike service exists

Create one, if no such service exists

```
// Interface for service inheriting Monobehaviour
public interface IMonoService
{
    public void RegisterService();
    public void UnregisterService();
}
```

Self Explanatory

```
// Interface for services not inheriting Monobehaviour
public abstract class INonMonoService
{
    protected INonMonoService() { }

}
```

Main Class.

```
public sealed class ServiceLocator : IProvideService , IDisposable
  {
    private IProvideService i_Instance;
    private static ServiceLocator m_instance;
    public static ServiceLocator Instance
    {
        get
        {
            if (m_instance == null)
            {
                m_instance = new ServiceLocator();
            }
            return m_instance;
        }

    }
```

Avoids being a Base class

Interface reference, so no overhead of type-casting.

Singleton Architecture

```
    private readonly Dictionary<System.Type, object> map; // Hashmap where all services will be kept.

    private ServiceLocator()
      {
          i_Instance = this;
          map = new Dictionary<System.Type, object>();
      }
```

private constructor for avoiding creation of instance outside of class.

```
    private T CreateMonoInstance<T>()
        {
            GameObject obj = new GameObject(typeof(T).Name);
            obj.AddComponent(typeof(T));
            return obj.GetComponent<T>();
        }
```

Method for creating a Monobehaviour Instance

```
    private T CreateNonMonoInstance<T>(Type t) where T : class
        {
            if (t.IsSubclassOf(typeof(INonMonoService))) // Working of 'INonMonoService' class as a Notifier.
            {
                System.Reflection.MethodInfo m = t.GetMethod("CreateInstance", System.Reflection.BindingFlags.Static | System.Reflection.BindingFlags.Public); // Making a method info such that we can call it.

                object[] caller = {this.GetType()}; // creating params // Supplying params

                object obj = m.Invoke(null, caller); // Invoking required function // Invoking that method and getting its instance return.

                return obj as T;  // Casting it to required type.
            }
            return null;
        }
```

Method for creating a Non-Monobehaviour Instance

```
public bool PeekService<T>()
{
    return map.ContainsKey(typeof(T)); // Checks if a Service exists based on the corresponding key {its Type}.
}
```

```
public void ClearAllServices() // Clear all Services
{
    if (map != null)
    {
        map.Clear();
    }
    else
    {
        Debug.Log("Error Clearing Service Map! \n Check Initialization");
    }
}
```

```
public void ClearService<T>()
{
    if (this.i_Instance.PeekService<T>()) // If a Service exists then, remove it.
    {
        map.Remove(typeof(T));
    }
    else
    {
        Debug.Log("Service does not exist in Map! \n Consider revising");
    }
}
```

```
public T PullService<T>(bool createIfNULL) where T : class
{
    if (this.i_Instance.PeekService<T>()) // If a Service exists, return it
        return map[typeof(T)] as T;

    if (createIfNULL) // If creation when not available is checked, then...
    {
        if (typeof(T).IsSubclassOf(typeof(MonoBehaviour)))
            this.i_Instance.PushService<T>(CreateMonoInstance<T>(), false);

        else
            this.i_Instance.PushService<T>(CreateNonMonoInstance<T>(typeof(T)), false);

        return map[typeof(T)] as T;
    }
    Debug.Log("Service Does not exist for a Pull.");
    return null;

}
```

Creation of Instance according to the Type of the Service

```
public void PushService<T>(T service, bool canOverwrite) where T : class
{
    if (service == null) // Checking whether supplied service is not empty.
    {
        Debug.Log("Null Service was intended to be registered!!!");
        return;
    }

    if (!this.i_Instance.PeekService<T>()) // Checking if service already exists.
    {
        map[typeof(T)] = service;
        Debug.Log("Service of Type " + typeof(T) + " has been registered");
        return;
    }

    if (canOverwrite) // Checking if overwriting param is true or not.
    {
        map[typeof(T)] = service;
        Debug.Log("Service of Type " + typeof(T) + "has been overridden");
        return;
    }

    //Debug.Log("Service already exists");

}
```

How to use it :

Consider having a "MonoBehaviour" class say, an InputService which will be responsible for providing input events and so some GetAxis responses.

```
public class InputManager : MonoBehaviour , IMonoService <-- Implementing the interface.
{
    public float HorizontalInput
    {
        get; private set;
    }

    public float VerticalInput
    {
        get;
        private set;
    }

    [SerializeField] private float horizontalSens, verticalSens;

    private void Awake()
    {
        RegisterService(); // Registering itself in Awake()
    }

    void Update()
    {
        MonitorInput();
    }

    private void MonitorInput()
    {
        HorizontalInput = Input.GetAxis("Horizontal");
        VerticalInput = Input.GetAxis("Vertical");
    }

    private void OnDisable() // Unregistering it in OnDisable()
    {
        UnregisterService();
    }

    public void RegisterService()
    {
        (ServiceLocator.Instance as IProvideService).PushService(this, true); // Pushing its reference in the service locator
    }

    public void UnregisterService()
    {
        (ServiceLocator.Instance as IProvideService).ClearService<InputManager>(); // Clearing its reference in the service locator.
    }
}
```

Consider having a "Non-MonoBehaviour" class say, an EventService which will be responsible for allowing multiple listeners, listen to a single / multiple subjects

```
public class EventService : INonMonoService <-- Inheriting the class which acts notifier
{
    private EventService()
    {

    }

    public static EventService CreateInstance(System.Type caller) <-- Function for creating instance, based on the type of caller
    {                                                                  we are supplying to it. {Needs to be specify explicitly in
        if (caller == typeof(ServiceLocator))                          all Non-Monobehaviour class}.
        {
            return new(); // create only when caller is of type
        }                         'Service locator'.
        else
        {
            Debug.Log("Instance creation access is forbidden for the type : " +caller);
            return null;
        }

    }

}
```

Getting a Service from Service Locator

Consider a User class "SwordHandAnim' which requires InputService to function.

```
public class SwordHandAnim : MonoBehaviour
{
    [SerializeField] private InputManager _inputManager;
    [SerializeField] private Transform orientation;
    [SerializeField] private Vector3 calcPosition;
    [SerializeField] private Vector3 horizontal_Ve;
    private EventService _eventService;

    void OnEnable()
    {
        _inputManager = (ServiceLocator.Instance as IProvideService).PullService<InputManager>(false);
        _eventService = (ServiceLocator.Instance as IProvideService).PullService<EventService>(true);

       //_____________________________
       _eventService.OnSlash.AddListener(SimulateTrail);

    }

    void Update()
    {
        CalculateHandTargetMovement();
    }

    void OnDisable()
    {
         _eventService.OnSlash.RemoveListener(SimilateTrail);
    }

    private void CalculateHandTargetMovement()
    {...}

   private void SimulateTrail()
   {...}

}
```