Unity Project Optimization

1. Variable Attributes

- `[SerializeField]:`
  - attribute you put before a field in your script to make it visible and editable in the **Inspector**, even if that field is **private** or **protected**.
- `[System.NonSerialized]`:
  - tells Unity **don't serialize this field**, so **Inspector hides it**.
  - Fields marked `[NonSerialized]` (or with `static`, `const`, `readonly`) won't show up in the inspector
- `[Header]:`
  - an Inspector attribute that lets you add a text heading above a field (or group of fields).
  - It doesn't affect your code at all — it's only for organizing how things look in the inspector.

---

2. Unity Event Functions

- `Update()`:
  - Called **once per frame**.
  - Frame-rate dependent.
  - Good for **regular gameplay logic** (movement, input checks, timers, visuals, UI).
- `LateUpdate()`:
  - Called **after all** `Update()` **methods** have run in that frame.

- Good for things that must happen **after movement/logic updates** (e.g., camera follow).
- Use when something must happen after other objects have updated.

> 💡 ▼ **Logic:**
>
> - Unity loops through **all active scripts** in the scene and calls their `Update()` methods.
>   - Every `MonoBehaviour` script that has an `Update()` method will have that method called **once per frame**.
> - After Unity has finished calling **every single `Update()` across all scripts**, then it moves on and calls all `LateUpdate()` methods.
>
> So the order is:
>
> 1. Unity: calls `Update()` on Script A
> 2. Unity: calls `Update()` on Script B
> 3. Unity: calls `Update()` on Script C
>
>   … continues for all scripts with Update
> 4. Unity: once all Updates are finished → calls `LateUpdate()` on Script A
> 5. Then `LateUpdate()` on Script B, etc.

- `FixedUpdate()`:
  - Runs at a **fixed timestep** (default 0.02s = 50 times/sec).
  - Independent of frame rate.
  - May run **more than once per frame** (if your frame takes longer than 0.02s), or **less often** (if frame rate is very high).
  - Good for: **physics, Rigidbody movement, forces**.

💡 ▼ Example Scenarios

Case 1: High FPS (e.g., 120 FPS)

- `Update()` → called **120 times/sec**.
- `FixedUpdate()` → still only **50 times/sec** (unless you change Time.fixedDeltaTime).
- 👉 So yes, `Update()` is called more often.

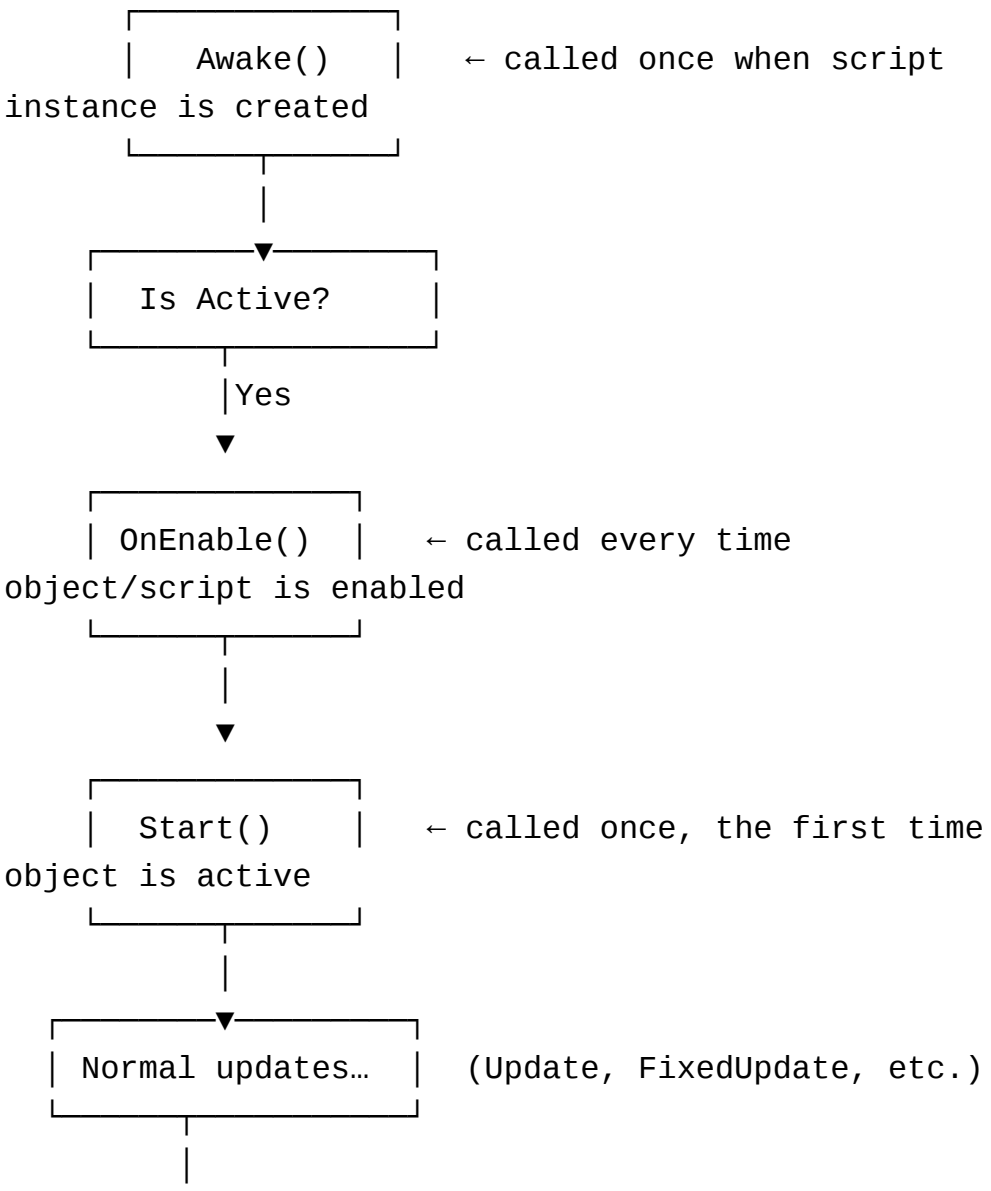
Case 2: Low FPS (e.g., 20 FPS)

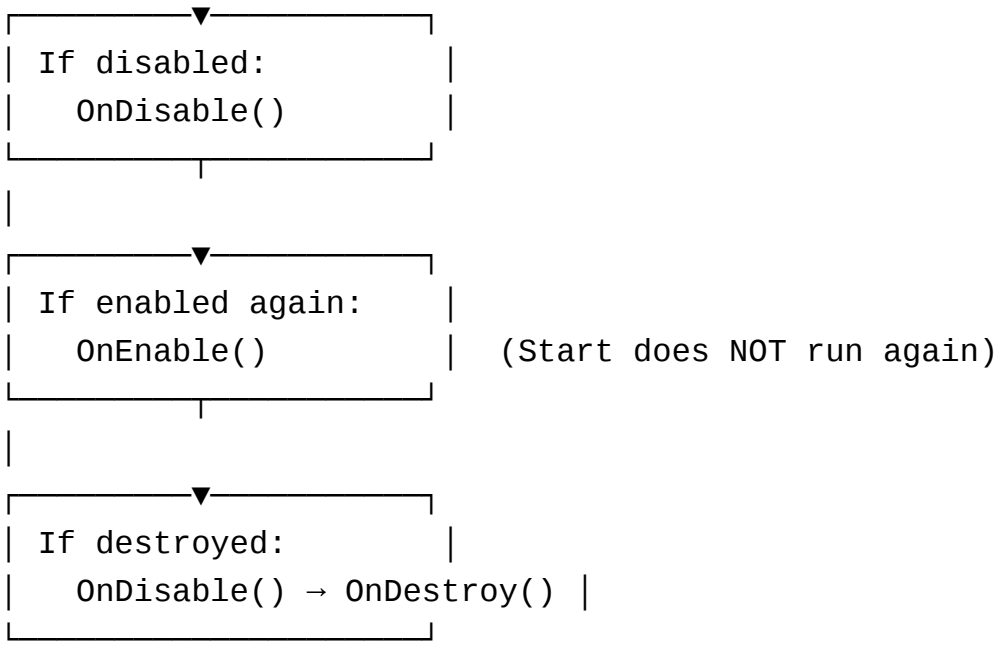
- `Update()` → called **20 times/sec**.
- `FixedUpdate()` → still **50 times/sec**.
- Unity will run multiple FixedUpdates before rendering the next frame, to “catch up” physics.

- `Awake()`:
  - Called **before** `Start()`.
  - Called **even if the GameObject is disabled** (unlike `Start()`).
  - Runs **only once** in the object's lifetime (unless the object is destroyed and re-instantiated or the scene is reloaded).
  - Used for **initialization that doesn’t depend on other scripts being enabled**.
- `OnEnable()`:
  - Called **every time** a script or GameObject becomes **enabled and active**.
  - This includes:
    - When the scene starts (if the GameObject is active)
    - Every time you call `gameObject.SetActive(true)`
    - Every time you enable the **component** (`myScript.enabled = true`)
- `OnDisable()`:

- A **MonoBehaviour method** Unity automatically calls.
- Runs **whenever** a script or its GameObject is **disabled**.
- That means it’s triggered when:
  - `gameObject.SetActive(false)` is called (whole GameObject disabled).
  - `myScript.enabled = false` is set (just that script disabled).
  - The object is **destroyed** (`Destroy(gameObject)` or scene unload).

▼ Flowchart:





Execution Order: `Awake()` → `OnEnable()` → `Start()`

---

3. Terminology

- **Singleton:**
  A **singleton** is a design pattern where you ensure that **only one instance** of a class exists in your project and provide a **global point of access** to it.
- **Singleton reference:**
  It's simply the **global reference** (usually a `static` variable) that other scripts can use to access that one unique instance of the singleton.

---

4. Object Pooling

Object pooling is a design pattern used in **game development** (and other software) to improve performance and reduce memory usage.

Instead of repeatedly **instantiating** (creating) and **destroying** objects at runtime (which is expensive in Unity and can cause garbage collection spikes), an **object pool** keeps a collection (pool) of pre-instantiated objects that can be reused.

How it works:

1. At the start, you **create a pool** of objects (e.g., 20 bullets).
2. When you need one, you **take (activate)** an available object from the pool.
3. When it’s no longer needed (e.g., bullet hits something), instead of destroying it, you **deactivate and return it** to the pool.
4. Next time, the same object can be reused instead of creating a new one.

💡 ▼ Example Code

```
public class ObjectPool : MonoBehaviour
{
public GameObject prefab;
public int poolSize = 10;
private List<GameObject> pool;

void Start()
{
   pool = new List<GameObject>();
   for (int i = 0; i < poolSize; i++)
   {
      GameObject obj = Instantiate(prefab);
      obj.SetActive(false);
      pool.Add(obj);
   }
}

public GameObject GetObject()
{
   foreach (var obj in pool)
   {
      if (!obj.activeInHierarchy)
      {
         obj.SetActive(true);
         return obj;
      }
   }

   // Optionally expand pool if none available
   GameObject newObj = Instantiate(prefab);
   newObj.SetActive(false);
   pool.Add(newObj);
   return newObj;
}
```

```
}
```

Benefit:

Faster performance, reduced lag, less garbage collection.

Downside:

Uses memory upfront to keep objects ready, even if not always used.