Unity3D: MonoBehaviour Lifecycle & Coroutines

Key event functions, execution flow, and practical coroutine patterns

eleonora.chitti@unimi.it

Laboratorio Realtà Virtuale 2025/2026

Learning goals

Understand what MonoBehaviour event functions are and when Unity calls them.

Choose correctly between Awake(), OnEnable(), Start(), Update(), FixedUpdate(), and OnDisable().

Understand what a Coroutine is, how yield return pauses execution, and how nested coroutines work.

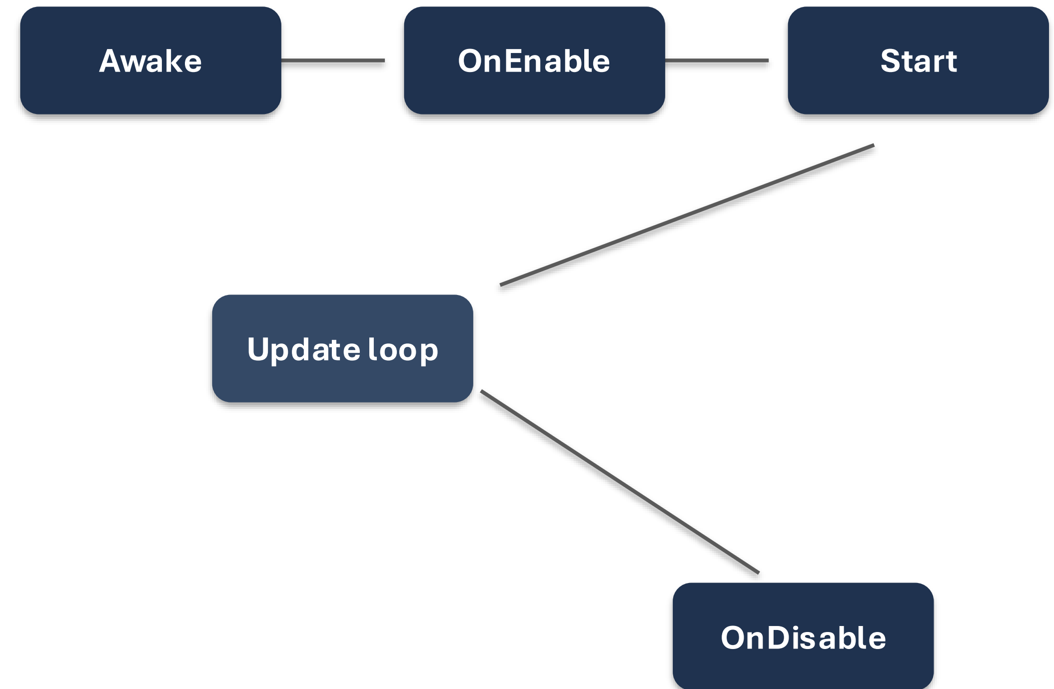
Read and write practical C# examples for gameplay sequences, timers, and asynchronous-style operations.

What is MonoBehaviour?

MonoBehaviour is the base class used by most Unity scripts attached to GameObjects.

Unity calls specific methods automatically at defined points in the object lifecycle.

These methods are not usually called manually by your code: they are “event functions” managed by the engine.



Core idea

Think of these methods as a structured timeline. Good Unity code uses each function for the job it was designed for.

Awake(), OnEnable(), Start()

Awake(): first initialization step. Use it to cache components and set internal state.

OnEnable(): called every time the component/GameObject becomes active. Good for subscribing to events.

Start(): called before the first frame update, after Awake/OnEnable. Good when initialization depends on other objects being ready.

Initialization example

```
Rigidbody rb;

void Awake()
{
    rb = GetComponent<Rigidbody>();
}

void OnEnable()
{
    GameEvents.OnDamage += HandleDamage;
}

void Start()
{
    UIManager.Instance.ShowHealth(currentHealth);
}
```

Rule of thumb: Awake = own setup; OnEnable = attach to events; Start = scene-level setup after other Awake calls.

Update(), FixedUpdate(), LateUpdate()

Update(): once per rendered frame. Use for input and non-physics frame-based logic.

FixedUpdate(): fixed timestep. Use for Rigidbody physics and forces.

LateUpdate(): after Update. Often used for cameras that follow objects after they move.

Frame vs physics timing

```
void Update()
{
    input = new Vector3(Input.GetAxis("Horizontal"), 0,
                        Input.GetAxis("Vertical"));
}

void FixedUpdate()
{
    rb.MovePosition(rb.position + input * speed * Time.fixedDeltaTi
}

void LateUpdate()
{
    cameraRig.position = target.position + offset;
}
```

Avoid applying Rigidbody forces in Update(), because Update is frame-rate dependent while physics uses a fixed timestep.

OnDisable() and cleanup

OnDisable(): called when the component or GameObject becomes inactive.

Use it to unsubscribe from events, stop local logic, or release temporary state.

Pair every subscription in OnEnable() with a matching unsubscription in OnDisable().

Safe event handling

```
void OnEnable()
{
    GameEvents.OnGamePaused += Pause;
}

void OnDisable()
{
    GameEvents.OnGamePaused -= Pause;
}

void Pause()
{
    // Local response to a global event
}
```

Why it matters: missing unsubscriptions can cause duplicated calls, memory leaks, or errors when disabled objects still receive events.

What is a Coroutine?

A Coroutine is an IEnumerator method that can pause execution with yield return and resume later.

It is not a separate thread: it runs on Unity's main thread and cooperates with the engine scheduler.

It is useful for delays, timed sequences, gradual animations, cooldowns, and waiting for conditions.

Basic coroutine

```
IEnumerator FlashDamage()
{
    sprite.color = Color.red;
    yield return new WaitForSeconds(0.2f);
    sprite.color = Color.white;
}

void TakeDamage()
{
    StartCoroutine(FlashDamage());
}
```

The yield line is the pause point. Unity resumes the method when the yielded instruction is complete.

Does StartCoroutine wait before continuing?

Code

```
void Start()
{
    Debug.Log("A");
    StartCoroutine(Example());
    Debug.Log("D");
}

IEnumerator Example()
{
    Debug.Log("B");
    yield return new WaitForSeconds(2f);
    Debug.Log("C");
}
```

Console order:

A is printed by Start().

The coroutine begins and prints B before its first yield.

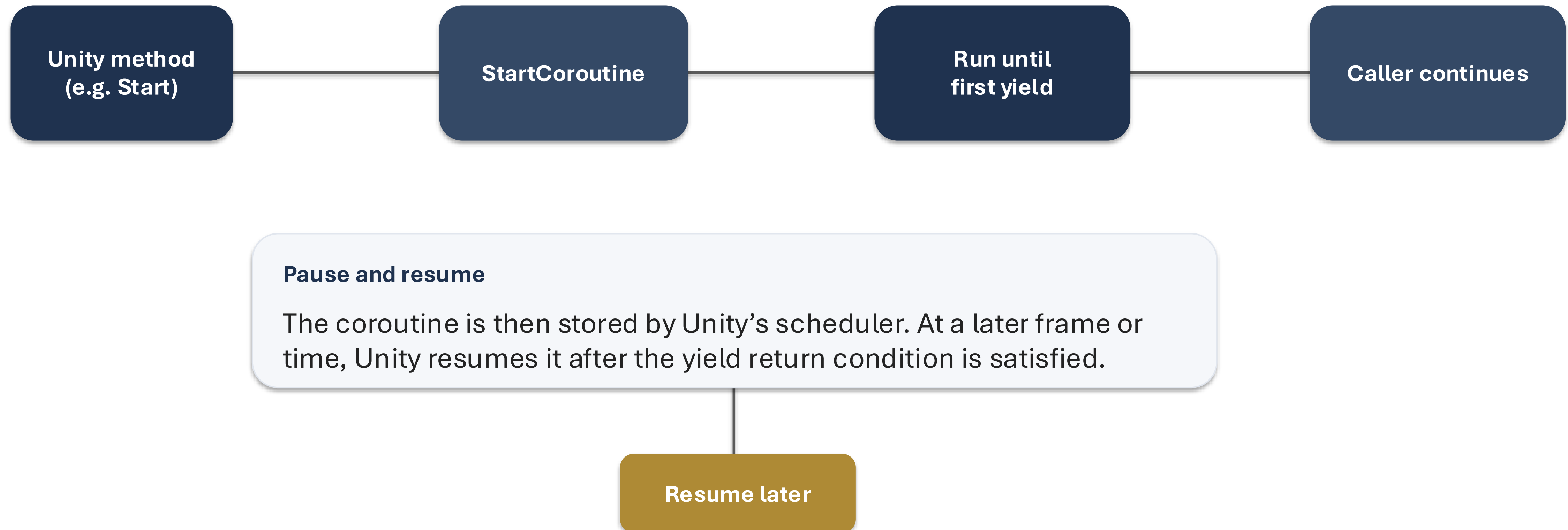
StartCoroutine returns at the first yield, so Start() continues and prints D.

After 2 seconds, the coroutine resumes and prints C.

Answer

Final order: A → B → D → C. StartCoroutine does not wait for the coroutine to finish unless you yield it from another coroutine.

How Unity schedules a Coroutine



Common yield return options

Yield instruction	Meaning
`yield return null`	Resume on the next frame.
`new WaitForSeconds(t)`	Wait scaled game time; affected by Time.timeScale.
`new WaitForSecondsRealtime(t)`	Wait real/unscaled time; useful during pause menus.
`new WaitForFixedUpdate()`	Resume at the next fixed physics step.
`new WaitForEndOfFrame()`	Resume at the end of the frame, after rendering work.
`new WaitUntil(() => condition)`	Resume when the condition becomes true.
`new WaitWhile(() => condition)`	Resume while true; continue when it becomes false.
`yield return StartCoroutine(Other())`	Wait until another coroutine finishes.
`yield return asyncOperation`	Wait for async loading, e.g. scene loading.
`yield break`	Exit the coroutine immediately.

A coroutine that waits for another coroutine

Sequential gameplay sequence

```
IEnumerator BossIntro()
{
    hud.Hide();

    yield return StartCoroutine(FadeOut(1f));
    yield return StartCoroutine(ShowBossTitle(2f));

    yield return new WaitUntil(() => playerPressedReady);
    StartFight();
}

IEnumerator FadeOut(float duration)
{
    float t = 0f;
    while (t < duration)
    {
        t += Time.deltaTime;
        screen.alpha = t / duration;
        yield return null;
    }
}
```

BossIntro pauses until FadeOut completes.

Then it waits until ShowBossTitle completes.

Then it waits for a player condition.

Only after all waits does StartFight() run.

When to use it

This is ideal for cutscenes, tutorials, scripted events, and ordered sequences that would become messy inside Update().

Starting vs waiting: parallel and sequential calls

Two different patterns

```
IEnumerator Parent()
{
    StartCoroutine(FadeOut(1f));
    Debug.Log("Runs immediately: FadeOut is parallel");

    yield return StartCoroutine(FadeIn(1f));
    Debug.Log("Runs after FadeIn has finished");
}
```

Equivalent waiting ideas

```
// Alternative nested style often used:
yield return FadeIn(1f);

// Explicit style with a Coroutine handle:
Coroutine c = StartCoroutine(FadeIn(1f));
yield return c;
```

StartCoroutine(Child()) alone: fire-and-forget; the parent continues after the child reaches its first yield.

yield return StartCoroutine(Child()): the parent coroutine pauses until the child coroutine has finished.

yield return c: waits for a stored Coroutine handle.

Use waiting when order matters. Use fire-and-forget when two timed behaviours can run concurrently.

Complex example: ability cooldown with UI fill

PRACTICAL EXAMPLE

Cooldown + per-frame UI update

```
Coroutine cooldownRoutine;
bool canDash = true;

void Update()
{
    if (Input.GetKeyDown(KeyCode.LeftShift) && canDash)
    {
        Dash();
        cooldownRoutine = StartCoroutine(DashCooldown(3f));
    }
}

IEnumerator DashCooldown(float seconds)
{
    canDash = false;
    float elapsed = 0f;

    while (elapsed < seconds)
    {
        elapsed += Time.deltaTime;
        dashIcon.fillAmount = elapsed / seconds;
        yield return null;        // update UI every frame
    }

    dashIcon.fillAmount = 1f;
    canDash = true;
}
```

Uses Update() only for input.

Uses a coroutine for the timed cooldown.

yield return null allows smooth UI progress over multiple frames.

No manual timer logic is needed in Update().

Complex example: wave spawner with nested routines

Waves, waits, and conditions

```
IEnumerator SpawnWaves()
{
    for (int wave = 1; wave <= maxWaves; wave++)
    {
        yield return StartCoroutine(SpawnSingleWave(wave));
        yield return new WaitUntil(() => enemiesAlive == 0);
        yield return new WaitForSeconds(2f);
    }

    ShowVictoryScreen();
}

IEnumerator SpawnSingleWave(int wave)
{
    int count = wave * 3;
    for (int i = 0; i < count; i++)
    {
        Instantiate(enemyPrefab, spawnPoint.position, Quaternion.identity);
        enemiesAlive++;
        yield return new WaitForSeconds(0.4f);
    }
}
```

The parent coroutine waits for each wave to finish spawning.

Then it waits until all enemies are defeated.

Then it adds a delay before the next wave.

The code reads like a timeline of the game event.

This pattern is useful for level scripting and encounter design.

Stopping coroutines and ownership

CONTROL

Store a handle if you need control

```
Coroutine routine;

void OnEnable()
{
    routine = StartCoroutine(AutoSaveLoop());
}

void OnDisable()
{
    if (routine != null)
        StopCoroutine(routine);
}

IEnumerator AutoSaveLoop()
{
    while (true)
    {
        SaveGame();
        yield return new WaitForSeconds(60f);
    }
}
```

StartCoroutine returns a Coroutine handle that can be stored.

StopCoroutine(handle) stops that specific coroutine.

StopAllCoroutines() stops all coroutines started by that MonoBehaviour.

Deactivating the GameObject stops its coroutines; setting enabled = false on the MonoBehaviour does not automatically stop them.

Design tip: the MonoBehaviour that starts the coroutine owns its lifetime.

When should you use a Coroutine?

Use Coroutines for timed, sequential, or condition-based flows.

Use Update() for continuous input reading and logic that must run every frame indefinitely.

Use FixedUpdate() for physics changes.

Do not use Coroutines for heavy CPU work: they still run on the main thread.

Avoid starting new coroutines every frame unless you explicitly intend to create many running routines.

Key takeaways

Awake(), OnEnable(), Start(), Update(), FixedUpdate(), and OnDisable() organize the lifecycle of a Unity component.

StartCoroutine begins a coroutine; the caller continues when the coroutine reaches its first yield.

yield return StartCoroutine(Other()) makes a parent coroutine wait for a child coroutine.

yield return options let you wait for frames, seconds, physics steps, end of frame, conditions, other coroutines, or async operations.

Coroutines make gameplay sequences easier to read, but they are not background threads.