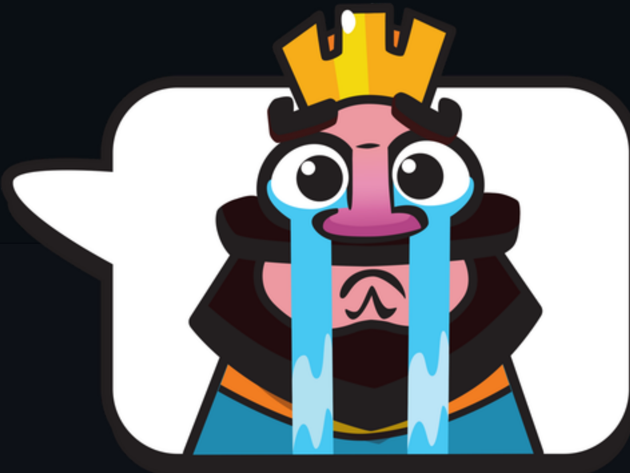
UNITY HACKS PART - 1/100

You're Using GetComponent<T>() Wrong - Here's a Faster Way!

```
private void OnCollisionEnter(Collision collision)
{
    Rigidbody rb = collision.gameObject.GetComponent<Rigidbody>();
    if (rb != null)
    {
        rb.AddForce(Vector3.up * 5f);
    }
}
```



BAD CODE

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.TryGetComponent<Rigidbody>(out Rigidbody rb))
    {
        rb.AddForce(Vector3.up * forceMultiplayer);
    }
}
```

GOOD CODE

Stop Using Strings for Scene Changes – Use Enums Instead!

```
public void LoadGameScene()
{
    SceneManager.LoadScene("GameScene");
}
```

BAD CODE

```
public enum SceneType { MainMenu, GameScene, Credits }

public void LoadScene(SceneType scene)
{
    SceneManager.LoadScene(scene.ToString());
}
```

GOOD CODE

UNITY HACKS PART - 3/100

HOW TO KILL YOUR GAME'S PERFORMANCE

```
public class BadStringComparison : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            // Do stuff to the Player
        }
    }
}
```

BAD CODE

```
public class GoodCompareTag : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            // Do stuff to the Player
        }
    }
}
```

GOOD CODE

MOVEMENT - Part 1

Transform Manipulation

1 - transform.position

```
transform.position += Vector3.forward * speed * Time.deltaTime;
```

- directly sets an object's position in world space.
- It overwrites any previous movement or forces acting on the object.

2 - transform.Translate()

```
transform.Translate(Vector3.forward * speed * Time.deltaTime, Space.Self);
```

- moves the object relative to its current position or a specified coordinate space.

3 - Vector3.Lerp()

```
transform.position = Vector3.Lerp(transform.position, targetPosition, speed * Time.deltaTime);
```

- Moves the object at a decreasing speed, slowing down as it approaches the target.

4 - Vector3.MoveTowards()

```
transform.position = Vector3.MoveTowards(transform.position, targetPosition, speed * Time.deltaTime);
```

- Moves the object at a constant speed until it reaches the target.

⚠ All these methods ignore physics and collisions, which can cause clipping through colliders.

You're Using Coroutines Wrong

Here's what you should do!

```
private void CallCoroutine => StartCoroutine(SlowAction());

private IEnumerator SlowAction()
{
    yield return new WaitForSeconds(5f);
    Debug.Log("Finished slow action!");
}
```

BAD CODE

```
private Coroutine currentRoutine;

private void CallCoroutine => currentRoutine = StartCoroutine(SlowAction());

private IEnumerator SlowAction()
{
    yield return new WaitForSeconds(coroutine_duration);
    Debug.Log("Finished slow action!");
}
```

GOOD CODE

You're Casts are BAD!

Use Layer Masks

```
private void BadRayCast()
{
    if (Physics.Raycast(transform.position, transform.forward, out RaycastHit hit, 100f))
    {
        Debug.Log("Hit: " + hit.collider.name);
    }
}
```

BAD CODE

```
[SerializeField] private LayerMask targetLayers;

private void GoodRayCast()
{
    // Use the Inspector-selected layers for all casts
    if (Physics.Raycast(transform.position, transform.forward, out RaycastHit hit, 100f, targetLayers))
    {
        Debug.Log("Hit: " + hit.collider.name);
    }
}
```

GOOD CODE

You are using are DontDestroyOnLoad() WRONG!

Use SINGLETON

```
public class BadPersistentClass : MonoBehaviour
{
    void Awake()
    {
        DontDestroyOnLoad(gameObject);
    }
}
```

BAD CODE

```
public class GoodPersistentClass : MonoBehaviour
{
    private static GoodPersistentClass instance;
    void Awake()
    {
        if (instance != null && instance != this)
        {
            // We already have one, destroy duplicates
            Destroy(gameObject);
            return;
        }
        instance = this;
        DontDestroyOnLoad(gameObject);
    }
}
```

GOOD CODE

Only BAD Game devs make this mistake

```
void Update()
{
    rigidbody.AddForce(Vector3.forward * 10f, ForceMode.Force);
}
```

BAD CODE

```
void FixedUpdate()
{
    rigidbody.AddForce(Vector3.forward * 10f, ForceMode.Force);
}
```

GOOD CODE

Don't make this mistake with Coroutines

Check Hack in Description

DO NOT use Find() & FindObjectOfType() at every frame

```
void Update()
{
    GameObject player = GameObject.Find("Player");
    transform.LookAt(player.transform);
}
```

BAD CODE

```
private PlayerController player;

void Awake()
{
    player = FindObjectOfType<PlayerController>();

    if (player == null)
        Debug.LogError("PlayerController not found!");
}

void Update() => transform.LookAt(player.transform);
```

GOOD CODE

```
[SerializeField] private PlayerController player; // Assigned via Inspector

void Update()
{
    transform.LookAt(player.transform);
}
```

GREAT CODE

Stop Checking Input Every Frame

Use EVENTS Instead!

```
if (Input.GetKeyDown(KeyCode.Space))
{
    Jump();
}
```

```
using UnityEngine.InputSystem;


private void OnJump(InputAction.CallbackContext context)
{
    if (context.performed)
    {
        Jump();
    }
}
```

Stop Using Time.deltaTime Incorrectly
Use fixedDeltaTime for Physics!

```
void FixedUpdate()
{
    rb.AddForce(Vector3.forward * speed * Time.deltaTime);
}
```

BAD CODE

```
void FixedUpdate()
{
    rb.AddForce(Vector3.forward * speed * Time.fixedDeltaTime);
}
```

GOOD CODE

Don't Pause your game like this
Do this instead!

```
void PauseGame()
{
    Time.timeScale = 0f;
}
```

BAD CODE

```
public interface IPauseable
{
    void Pause();
    void Unpause();
}

public class Player : MonoBehaviour, IPauseable
{
    // Code related to Player class

    public void Pause()
    {
        // Pause logic for player
    }

    public void Unpause()
    {
        // Unpause logic for player
    }
}
```

GOOD CODE

Don't Start Coroutines Like This

```
Coroutine damageCoroutine = StartCoroutine("ApplyDelayedDamage", 2f);

IEnumerator ApplyDelayedDamage(float interval){
    yield return new WaitForSeconds(interval);
    playerHealth.TakeDamage(10);
}
```

BAD CODE

```
Coroutine damageCoroutine = StartCoroutine(ApplyDelayedDamage(2f, 10));

IEnumerator ApplyDelayedDamage(float interval, int damageAmount){
    yield return new WaitForSeconds(interval);
    playerHealth.TakeDamage(damageAmount);
}
```

GOOD CODE

Don't make this mistake when Stopping Coroutines

Don't use different arguments

```
StartCoroutine(MyCoroutine()); // Using IEnumerator
StopCoroutine("MyCoroutine"); // ❌ This will not stop it!
```

BAD CODE

```
Coroutine myCoroutine; // Store the coroutine reference
myCoroutine = StartCoroutine(MyCoroutine());
StopCoroutine(myCoroutine); // ✅ Correct way to stop it
myCoroutine = null;
```

GOOD CODE

PlayerPrefs Is NOT a Save System

Do This Instead!

```
public void SaveGame()
{
    PlayerPrefs.SetFloat("PlayerHealth", playerHealth);
    PlayerPrefs.SetInt("CurrentQuestId", currentQuestId);
    PlayerPrefs.SetInt("DoorUnlocked", doorUnlocked ? 1 : 0);

    // Must be called to persist changes
    PlayerPrefs.Save();
}
```

BAD CODE

```
using System.Runtime.Serialization.Formatters.Binary;

[System.Serializable]
public class SaveData
{
    public float playerHealth;
    public int currentQuestId;
    public bool doorUnlocked;
}

public void SaveGame()
{
    SaveData data = new SaveData
    {
        playerHealth = playerHealth,
        currentQuestId = currentQuestId,
        doorUnlocked = doorUnlocked
    };

    BinaryFormatter bf = new BinaryFormatter();

    // Create or overwrite "savegame.dat"
    using (FileStream fs = new FileStream(SavePath, FileMode.Create))
        bf.Serialize(fs, data);
}
```

GOOD CODE

Don't make game's Enemy AI with Boolean!

```
public class Enemy_BadExample : MonoBehaviour
{
    private bool isPatrolling = true;
    private bool isChasing    = false;
    private bool isAttacking  = false;

    //rest of the code
]
```

BAD CODE

Use State Machine

```
public enum EnemyState
{
    Patrol,
    Chase,
    Attack
}
```

```
public interface IEnemyState
{
    void OnEnter();
    void OnUpdate();
    void OnExit();
}
```

```
/*Classes for each state
that implement IEnemyState*/
public class PatrolState : IEnemyState {}
public class ChaseState : IEnemyState {}
public class AttackState : IEnemyState {}
```

GOOD CODE

```
public class Enemy_GoodExample : MonoBehaviour
{
    private IEnemyState currentState;

    public void ChangeState(IEnemyState newState)
    {/*Logic to change State*/}

    //rest of the code
}
```

PlayerPrefs Is NOT a Save System

```
public void SaveGame()
{
    PlayerPrefs.SetFloat("PlayerHealth", playerHealth);
    PlayerPrefs.SetInt("CurrentQuestId", currentQuestId);
    PlayerPrefs.SetInt("DoorUnlocked", doorUnlocked ? 1 : 0);

    // Must be called to persist changes
    PlayerPrefs.Save();
}
```

BAD CODE

Use Encrypted JSON instead!

```
public void SaveGame(SaveData data)
{
    string json = JsonUtility.ToJson(data);

    //Encrypt JSON using AES or other methods
    string encryptedJson = Encrypt(json, encryptionKey);
    File.WriteAllText(savePath, encryptedJson);
}
```

GOOD CODE

Tag comparison SLOWS down collision detection

```
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Enemy")
    {
        Debug.Log("Hit an enemy!");
    }
}
```

BAD CODE

Use LayerMask Bitwise Operations!

```
void OnCollisionEnter(Collision collision)
{
    if ((enemyLayer.value & (1 << collision.gameObject.layer)) != 0)
    {
        Debug.Log("Hit an enemy!");
    }
}
```

GOOD CODE

Don't use MonoBehaviour classes to store data

```
public class Weapon : MonoBehaviour
{
    public WeaponType weaponType;
    public float damage = 20f;
}
```

BAD CODE

Use ScriptableObject Instead!

```
[CreateAssetMenu(fileName = "WeaponData", menuName = "Game/WeaponData")]
public class WeaponData : ScriptableObject
{
    [SerializeField] private WeaponType weaponType; //WeaponType enum
    [SerializeField] private float damage = 20f;

    public WeaponType WeaponType => weaponType;
    public float Damage => damage;
}
```

GOOD CODE

String Concatenation Is Killing Your Game's Performance

```
public class PlayerHUD : MonoBehaviour
{
    void Update()
    {
            string message = "Player: " + playerName + " Score: " + score + " Level: " + level;
        playerStatsText.text = message;
    }
}
```

BAD CODE

Use StringBuilder Instead!

```
public class PlayerHUD : MonoBehaviour
{
    private readonly StringBuilder stringBuilder = new StringBuilder(100);

    void Update()
    {
        stringBuilder.Clear();
        stringBuilder
            .Append("Player: ")
            .Append(playerName)
            .Append(" Score: ")
            .Append(score)
            .Append(" Level: ")
            .Append(level);

        playerStatsText.text = stringBuilder.ToString();
    }
}
```

GOOD CODE

DO NOT USE try-catch AND WASTE TIME

```
string json = File.ReadAllText(path);
SaveData data = JsonUtility.FromJson<SaveData>(json);
```

BAD CODE

```
try {
    string json = File.ReadAllText(path);
    SaveData data = JsonUtility.FromJson<SaveData>(json);
} catch (Exception ex) {
    Debug.LogError($"Load failed: {ex.Message}");
}
```

GOOD CODE

DO NOT PUT EVERYTHING in a SINGLE SCRIPT

```
public class PlayerGodScript : MonoBehaviour
{
    void Update()
    {
        HandleMovement();
        HandleShooting();
        HandleHealth();
        UpdateUI();
        CheckEnemyCollisions();
        PlaySoundEffects();
        UpdateAnimations();
        HandleInventory();
        ManageQuests();
        ApplyPowerups();
        ShowDialogue();
        CheckAchievements();
        // And dozens more systems...
    }

    // Hundreds of lines more code here...
}
```

BAD CODE

```
public class PlayerMovement : MonoBehaviour {}   // Handles Movement
public class PlayerHealth : MonoBehaviour {}     // Handles Health
public class PlayerCombat : MonoBehaviour {}     // Handles Shooting
public class PlayerAnimator : MonoBehaviour {}   // Handles Animations
public class PlayerInventory : MonoBehaviour {} // Handles Items
public class PlayerUI : MonoBehaviour {}         // Handles Interface
```

GOOD CODE