ดนัย เจษฎาฐิติกุล, จารุต บุศราทิจ DcG.IT@PBRU

Presented by Ares Grau

Agenda

1. Syntax พื้นฐาน
2. Control flow (if/else, loops)
3. Functions & methods
4. Class และ Object
5. inheritance/polymorphism
6. GameObject & Components
7. MonoBehaviour class และ Lifecycle methods
8. Coroutines
9. Transform
10. Physics
11. Scenes & Scene Management
12. Q&A

Syntax พื้นฐาน

1. Variable
2. data types
3. operators

“สามารถใช้ตัวแปรเพื่อเก็บค่า และข้อมูลที่เรียกค้นได้ง่าย ๆ โดยตัวแปรของ C# ประกอบด้วยตัวชี้ (pointer) และตัวอ้างอิง”

Variable Declaration

```
int number = 10;
string str = "Hello, World!";
```

“C# มีการสนับสนุนข้อมูลพื้นฐานทั้งหมด เช่น int, float, string, boolean เป็นต้น นอกจากนี้ยังสามารถจัดเก็บค่าจำนวนนับไม่จำกัดด้วยตัวแปร long”

Data Types

```
int x = 2147483647; // จำนวนเต็ม
float y = 1234.56; // ทศนิยม
bool z = false;
char a = 'C';
```

"ภาษา C# มีความเป็นไปได้สองประการที่สามารถใช้กับชนิดข้อมูล int และ float นั่นคือ สามารถดำเนินการทางคณิตศาสตร์ เช่น การบวก การลบ การคูณ และการหาร เป็นต้น นอกจากนี้ยังสามารถดำเนินการทางตรรกะโดยใช้ตัวดำเนินการที่เทียบเท่ากันได้อีกด้วย"

Operators

```
int a = 5;
a *= 2; // การคูณ

// การเปรียบเทียบที่แตกต่างกัน
bool x = (a < b); // ไม่มีสัญลักษณ์เครื่องหมาย
```

พยายามทำความคุ้นเคยกับภาษาโปรแกรมโดยการเขียนโค้ดมากขึ้น

นอกจากนี้ยังเป็นสิ่งสำคัญที่จะทราบว่าตัวอย่างสคริปต์ Unity ที่เรียบง่ายมักเกี่ยวข้องกับ C# พื้นฐาน

Control flow

1. if/else
2. loops

"ในภาษา C# คำสั่ง 'if' เป็นคำสั่งควบคุมที่ใช้ตรวจสอบเงื่อนไขและดำเนินการตามลำดับ"

If Statement

```
if (condition)
{
// เมื่อเงื่อนไขเป็นจริง ให้รันบล็อกนี้ของโค้ด
}
```

สิ่งที่ทำ

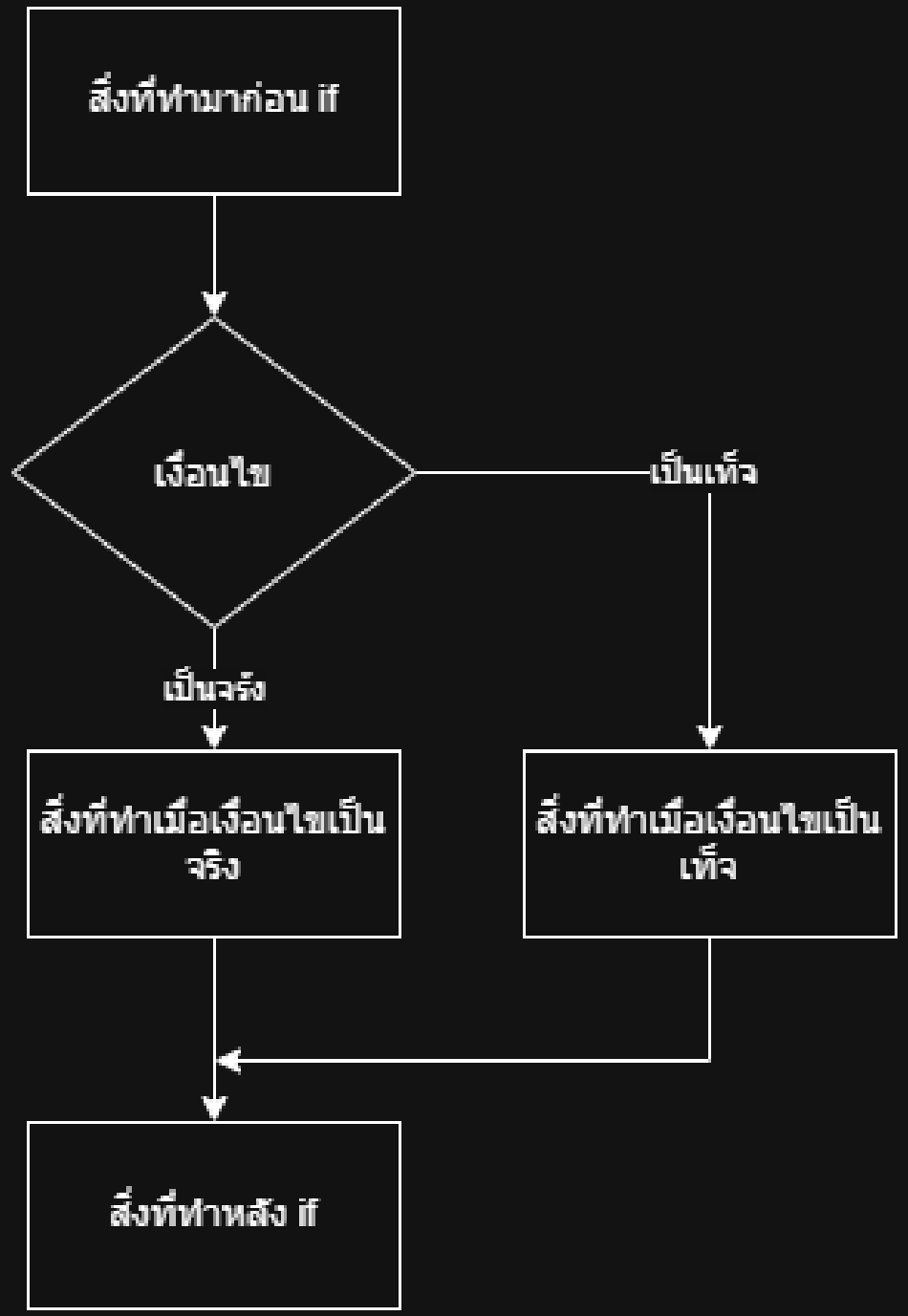
```
if (condition)
{
// เมื่อเงื่อนไขเป็นจริง ให้รันบล็อกนี้ของโค้ด
}
```

สิ่งที่ทำ

"else เป็นคำสั่งควบคุมที่ใช้ระบุว่าควรเรียกใช้โปรแกรมใด หากไม่ได้ตรวจสอบเงื่อนไขในคำสั่ง 'if' ไว้ก่อนหน้านี้"

Else Statement

```
else
{
// เมื่อเงื่อนไขเป็นเท็จ ให้รันบล็อกนี้ของโค้ด
}
```




```
สิ่งที่ทำ

if (condition)
{
// เมื่อเงื่อนไขเป็นจริง ให้รันบล็อกนี้ของโค้ด
}
else
{
// เมื่อเงื่อนไขเป็นเท็จ ให้รันบล็อกนี้ของโค้ด
}

สิ่งที่ทำ
```

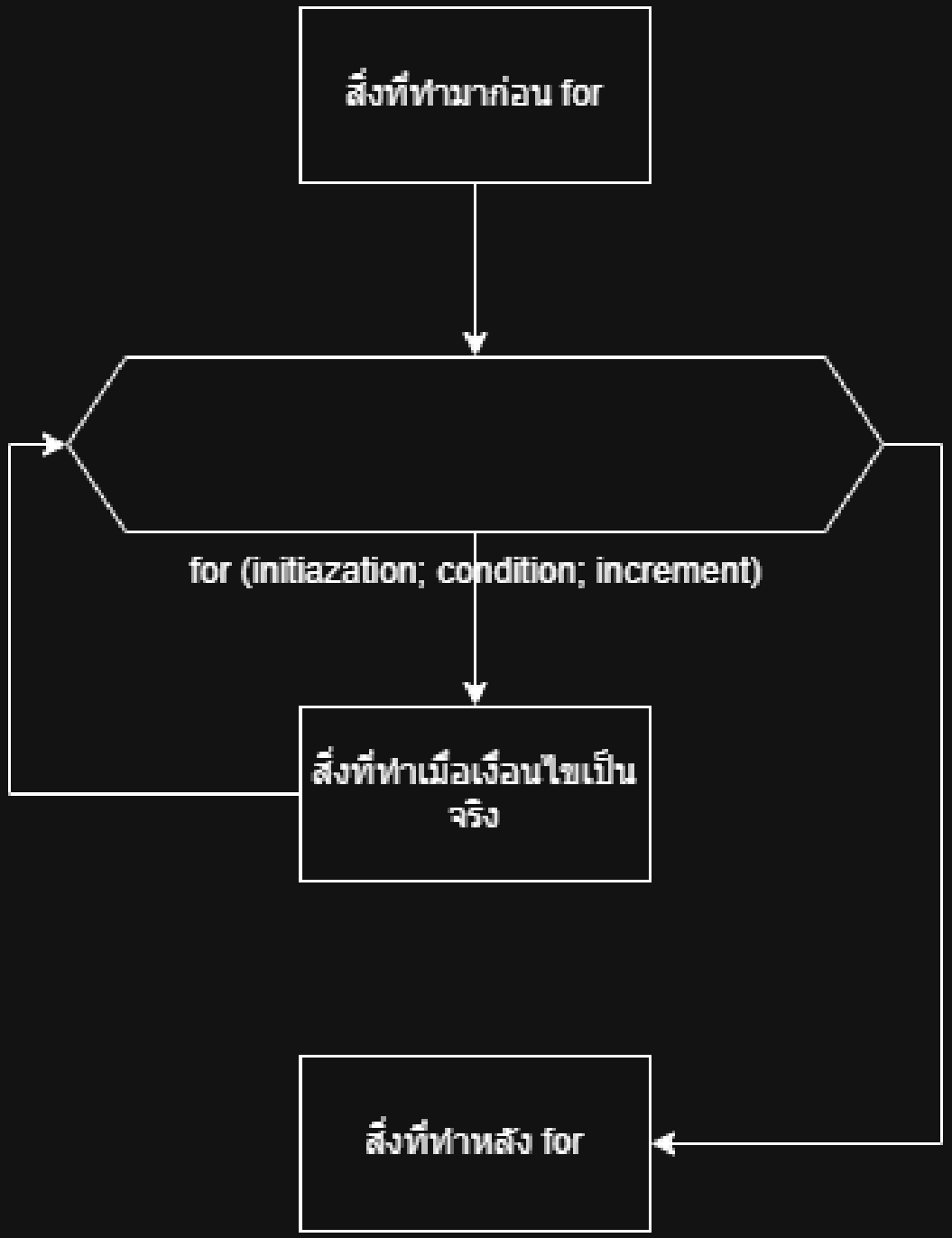
"การทำซ้ำช่วยให้สามารถดำเนินการซ้ำ ๆ ได้อย่างมีประสิทธิภาพโดยไม่ต้องทำด้วยตนเองทุกครั้ง"

Loops

"นี่คือการวนซ้ำที่พบได้บ่อยที่สุด โดยมีสถานะหลักสามสถานะ ได้แก่ สถานะเริ่มต้น สถานะเงื่อนไข และสถานะเพิ่มขึ้น"

For Loop

```
for (initiazation; condition; increment)
{
// ทำงาน
}
```



สิ่งที่ทำ

```
for (initiazation; condition; increment)
{
// ทำงาน
}
```

สิ่งที่ทำ

หลักการทำงานของ for

Initiazation

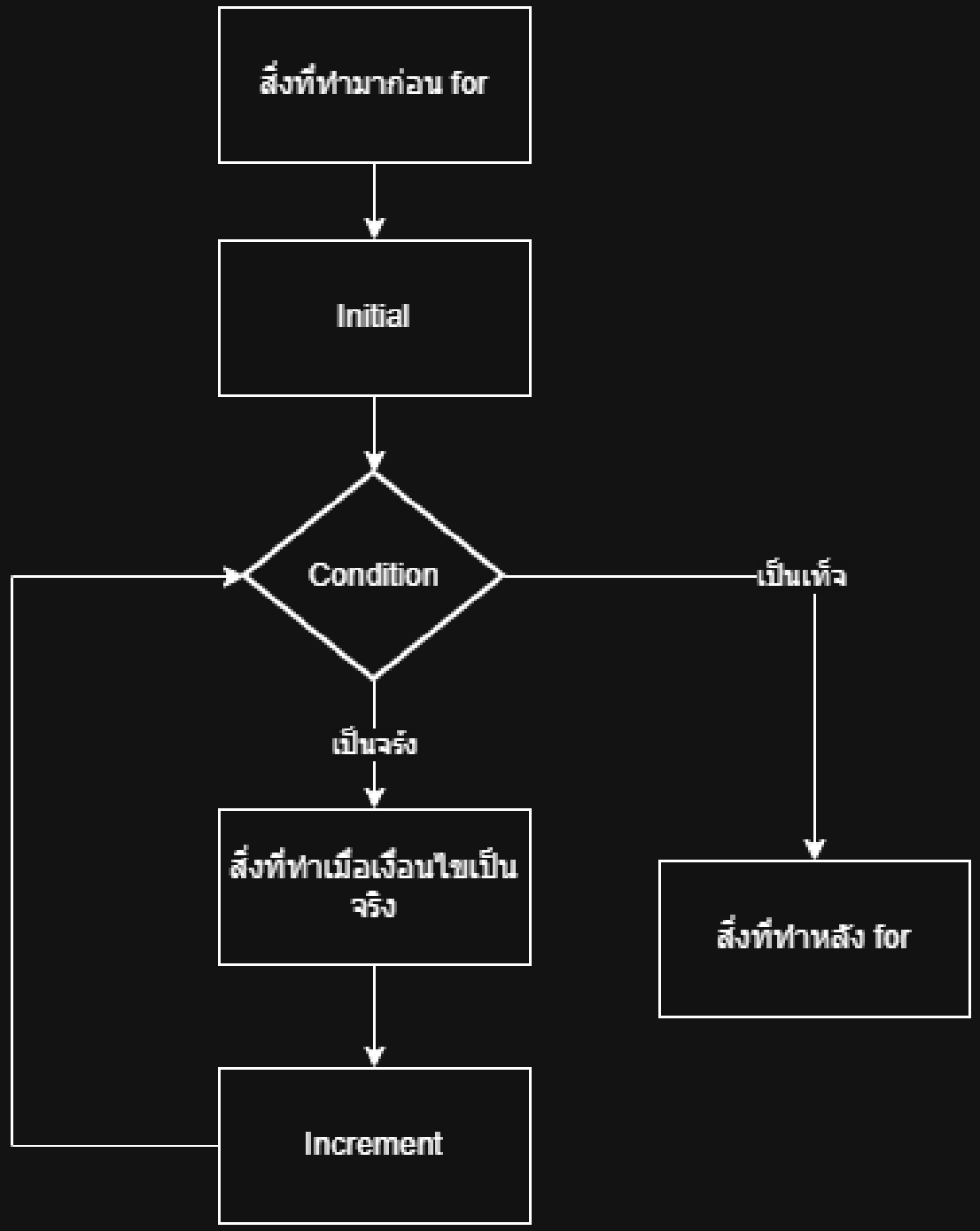
ช่วงเริ่มต้นก่อนตรวจสอบเงื่อนไขเพื่อทำการทำซ้ำ (ทำเพียงครั้งเดียว)

Condition

เงื่อนไขสำหรับตรวจสอบก่อนทำคำสั่งภายใน for แต่ละรอบ (ถ้าเป็นจริงจะทำ แต่ถ้าเป็นเท็จจะจบการทำซ้ำ)

Increment

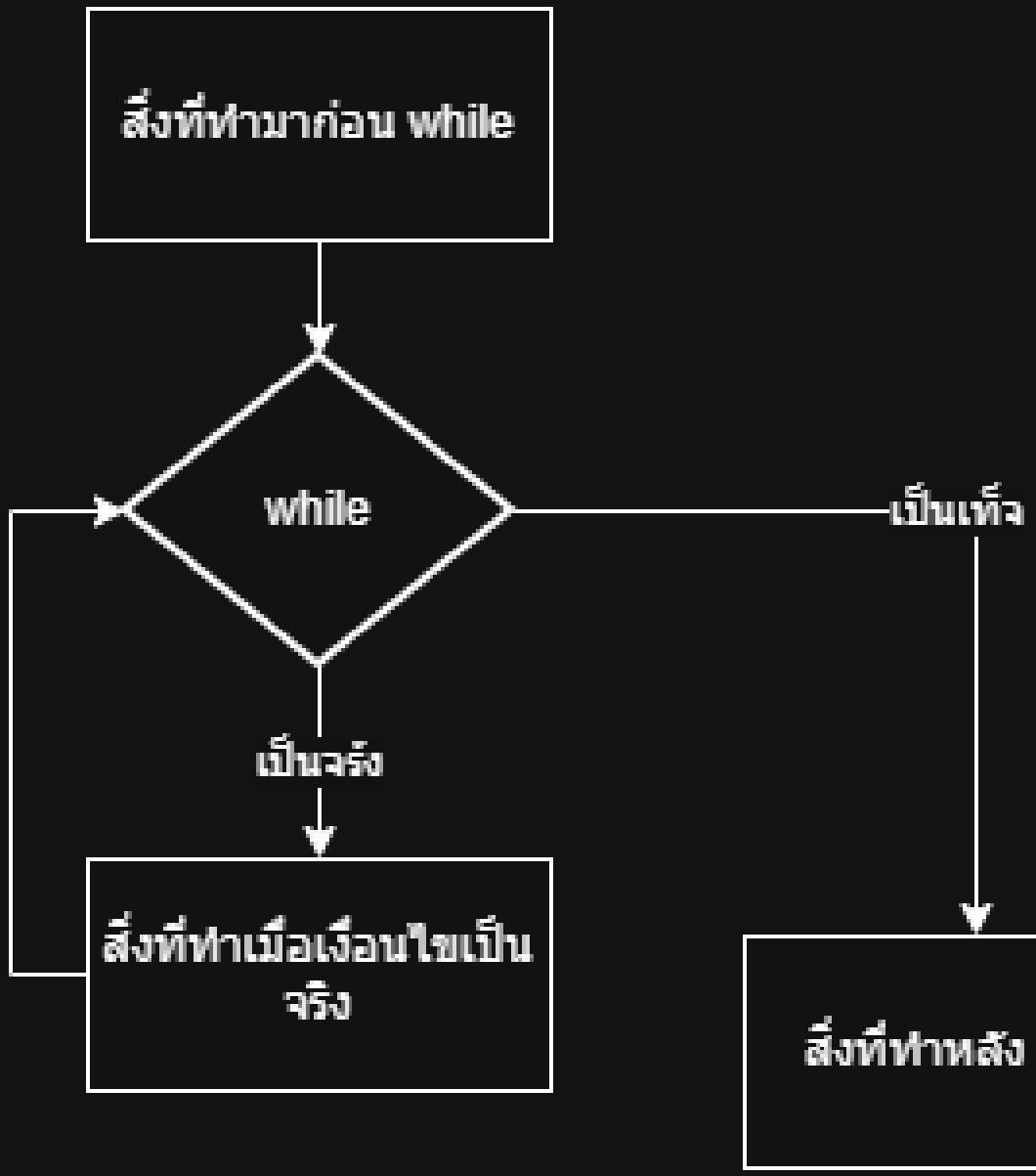
ส่วนของการเพิ่ม/ลดค่า (หรืออื่น ๆ ที่ต้องการ) เป็นส่วนที่ทำงานหลังจากทำคำสั่งต่าง ๆ ใน for เสร็จแล้ว และเมื่อทำงานของ Increment เสร็จจะกลับไปทำ Condition


สิ่งที่ทำมาก่อน for
Initial
Condition
เป็นเท็จ
เป็นจริง
สิ่งที่ทำเมื่อเงื่อนไขเป็นจริง
สิ่งที่ทำหลัง for
Increment

"ลูป while จะทำงานซ้ำไปเรื่อย ๆ ตราบใดที่เงื่อนไขยังเป็นจริง"

While Loop

```
while(condition)
{
    // ทำงาน
}
```

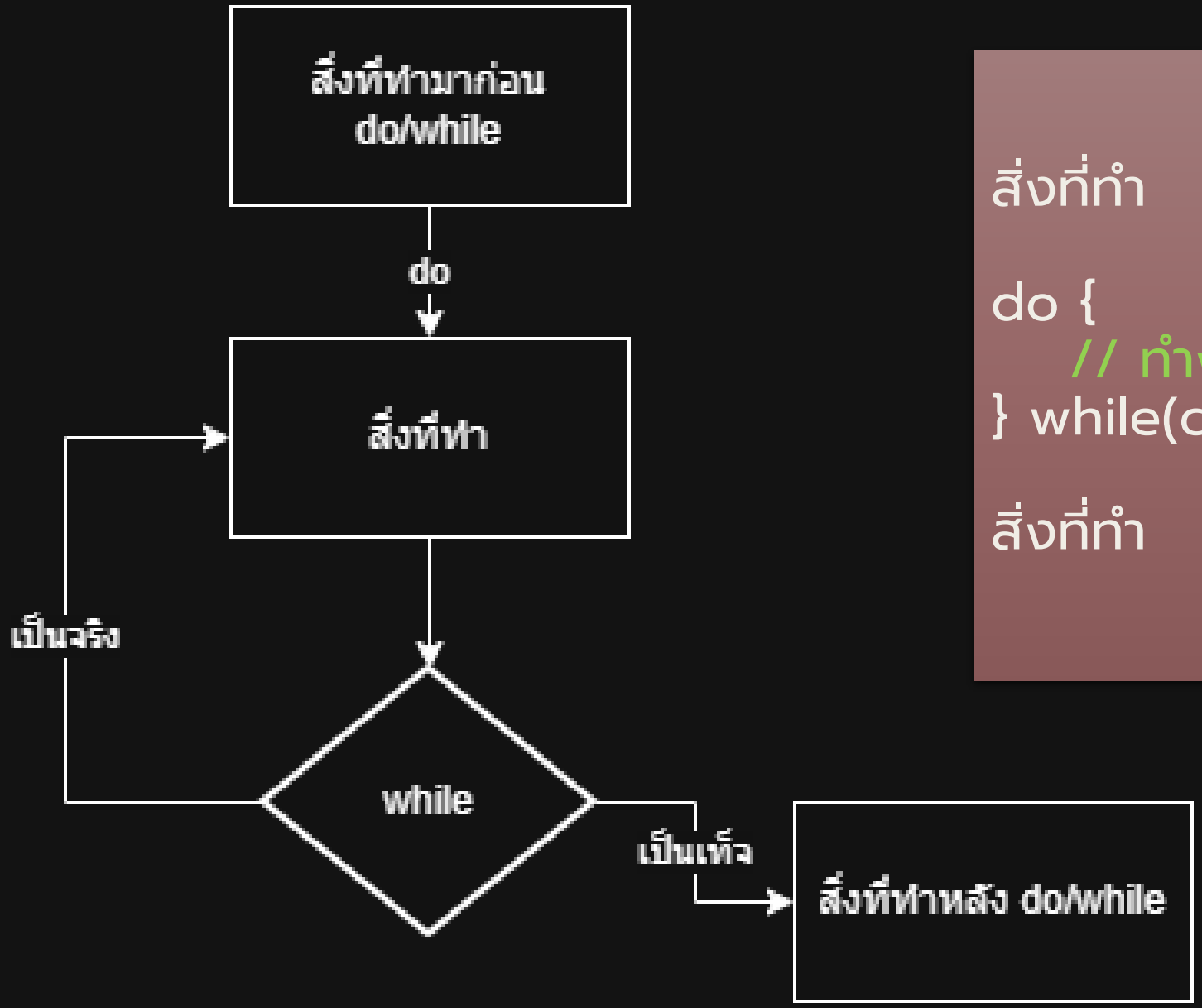



```
สิ่งที่ทำ

while(condition)
{
    // ทำงาน
}

สิ่งที่ทำ
```

"ในบางสถานการณ์ วิธีนี้มีประสิทธิภาพมากกว่า เพราะจะดำเนินการตามบล็อกก่อน จากนั้น ตรวจสอบเงื่อนไขและทำซ้ำกระบวนการตามนั้น"

Do-While Loop

```
do {
    // ทำงาน
} while(condition);
```




```
สิ่งที่ทำ

do {
    // ทำงาน
} while(condition);

สิ่งที่ทำ
```

"ลูปจะทำงานในลักษณะเดียวกันเมื่ออยู่ภายในบล็อกลูปเพิ่มเติม เช่น `for` ภายใน `for` หรือ `while` ภายใน `while` เป็นต้น"

Nested Loops

ข้อควรทราบ

ใช้ break;

เพื่อออกจากการวนซ้ำ

ใช้ continue;

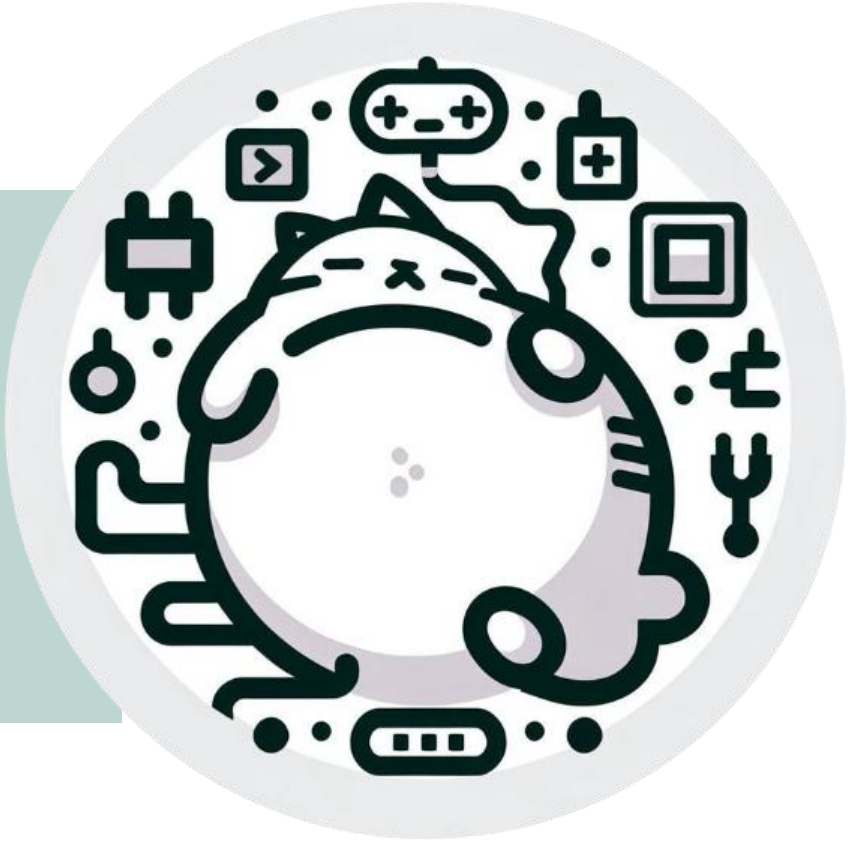
เพื่อผ่านรอบการวนซ้ำปัจจุบัน

แนะนำ

หากมีหลายบล็อก ควรใช้คำสั่งควบคุม เช่น if, else และ switch

อย่าลืมว่า**การวนซ้ำ**ใน Unity นั้น จำเป็นสำหรับการทำงานในเกม เช่น การทำให้ตัวละครเดินไปรอบ ๆ ฉากทั้งหมด เป็นต้น

Functions & methods



"Methods ใน C# มีวิธีหลายประเภท"

Methods

วิธีการแบบสแตติก: ไม่ส่งค่ากลับ

```
public static int Sum(int a, int b)
{
    return a + b;
}
```

วิธีการที่ไม่ใช่สแตติก: ส่งค่ากลับ

```
public virtual string MyString { get; set; }
```

วิธีการที่มีพารามิเตอร์: ส่งค่ากลับคืน

```
public int MyInt { get; set; } = 0;
```

“คีย์เวิร์ด `return` ใช้สำหรับส่งค่ากลับจากเมธอด"

Return

ระดับการเข้าถึง (Access Level)
สาธารณะ (public) อนุญาตให้ผู้ใช้เข้าถึงได้
ป้องกัน (protected) อนุญาตให้ผู้ใช้เข้าถึงได้
ส่วนตัว (private) อนุญาตให้เข้าถึงเฉพาะภายในคลาสเท่านั้น

Access Level

ใช้ **private** และ **public** กับประเภทเมธอดเพื่อจำกัดขอบเขตของการมองเห็น

หลีกเลี่ยงการใช้ชุดคำสั่งแบบ

สแตติก (static)

และ**ไม่ใช่สแตติก**

Class และ Object ของ C# สำหรับ Unity

ใช้ class เพื่อสร้างสิ่งที่เรียกว่าอ็อบเจ็กต์หรือประเภท

C# Class

ในภาษา C# คลาสเปรียบเสมือนแบบแผน ในขณะที่อ็อบเจ็กต์คือสิ่งที่มีอยู่จริง

C# Class

```
public class MyClass
{
    public string MyString { get; set; }
}
```

```
public class GameObject
{
    public string Name { get; set; }
    public int Health { get; set; }
}
```

```
// การสร้าง Object
var player = new GameObject();
player.Name = "Player";
player.Health = 100;
```

สามารถสร้างอ็อบเจ็กต์หรือประเภท
ได้โดยใช้คำสั่ง `class`

MonoBehaviour คือคลาสที่ทำงานภายใน Unity และมีฟังก์ชันสำหรับสร้างส่วนประกอบเกม

```
using UnityEngine;

public class Player : MonoBehaviour
{
    public int health;

    void Start() {
        Debug.Log("Player started!");
    }

    void Update() {
        // ทำงานทุกเฟรม (60Hz)
    }
}
```

Unity ใช้ Component-based architecture

Component System

Unity มีคลาสและฟังก์ชันที่รองรับ Component เช่น Transform, Rigidbody และ Rigidbody2D เป็นต้น

```
// GameObject มีหลาย components: Transform, Rigidbody, etc.
void OnCollisionEnter(Collision collision)
{
    Debug.Log("Hit " + collision.gameObject.name);
}
```

MonoBehaviour เป็น class หลักของ Unity

ทุก GameObject ต้องมี Component (เช่น Transform)

ใช้ [SerializeField] เพื่อ expose variable ใน Inspector

[SerializeField] เป็น Attribute ที่บอกให้ Unity แสดงตัวแปรใน Inspector โดยไม่ต้องทำให้ทั้งคลาสเป็น Serializable

[SerializeField]

```
using UnityEngine;

public class Player : MonoBehaviour
{
    // จะปรากฏใน Inspector
    [SerializeField] private int health = 100;

    // ไม่ปรากฏ (private เฉยๆ)
    private string debugMode = "normal";
}
```

Attribute	ใช้กับอะไร?	ผลลัพธ์
[SerializeField]	ตัวแปร (fields/properties)	แสดงใน Inspector, serializable เฉพาะตัวแปรนั้น
[Serializable]	Class/Struct/Enum ทั้งคลาส	ทำให้ทั้งคลาส + ทุกสมาชิก serializable

```
// ใช้ SerializeField สำหรับ variables
[Serializable]
public class Player : MonoBehaviour
{
    [SerializeField] private int health = 100;  // แสดงใน Inspector
    [SerializeField] private float speed = 5f;   // แสดงใน Inspector
    private string tempData = "temporary";       // ไม่แสดง
}

// ใช้ Serializable สำหรับ custom classes
[System.Serializable]
public class SaveData
{
    public int level;
    public List<string> achievements;
}
```

1. ไม่ต้อง serializable ทั้งคลาส ทำให้คลาสเล็กกว่า
2. เลือก expose ได้เฉพาะที่ต้องการ
3. ปลอดภัยกว่า เนื่องจาก private fields ไม่รั่วไหลใน Inspector
3. ทำงานได้กับ Unity ทุกรุ่น

ข้อดี

1. ใช้ได้กับ fields และ properties เท่านั้น
2. ไม่มี effect กับ
    2.1 methods/constructors
    2.2 static variables
    2.3 Unity's internal types

ข้อจำกัด

```
[Serializable]
public class Example : MonoBehaviour
{
    [SerializeField]
    private int publicVar = 10;       // OK - field

    //  ไม่ทำงาน เนื่องจากใช้กับ methods
    [SerializeField]
    private void Update() { }

    // ไม่ทำงาน เนื่องจากใช้กับ properties (ควรใช้ SerializeProperty)
    public float Speed
    {
        get => _speed;
        set => _speed = value;
    }
}
```

Best Practices

```
public class GameConfig : MonoBehaviour
{
    [SerializeField] private int maxPlayerLives = 3;
    [SerializeField] private float gameTime = 60f;

    // ใช้ในโค้ดได้ ไม่ต้อง expose เป็น public
}
```

ใช้ private + SerializeField เป็นมาตรฐาน

```
public class PlayerSettings : MonoBehaviour
{
    [Header("Movement")]
    [SerializeField] private float moveSpeed = 5f;
    [SerializeField] private bool autoRun = false;

    [Header("Combat")]
    [SerializeField] private int damage = 10;
    [SerializeField] private float attackCooldown = 1f;
}
```

Group with Labels

```
public class PlayerController : MonoBehaviour
{
    [Range(0f, 20f)]
    [SerializeField] private float speed = 5f;

    [Range(-180f, 180f)]
    [SerializeField] private float rotation = 0f;
}
```

ใช้ Min/Max สำหรับ Range Slider

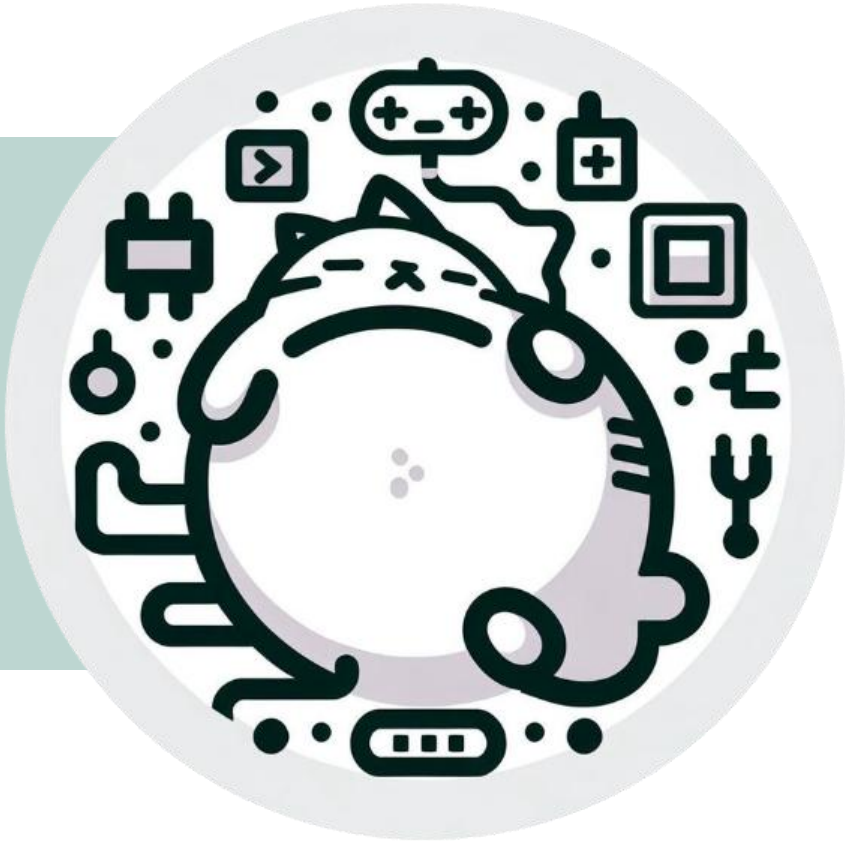
ตัวอย่าง Save System

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

public class PlayerSave : MonoBehaviour
{
    // Expose ใน Inspector แต่เก็บเป็น private ในโค้ด
    [SerializeField]
    [Min(1)] private int saveSlot = 1;

    [SerializeField]
    private string playerName = "Player";

    // ไม่ expose → เก็บในโค้ดเท่านั้น
    private Dictionary<int, int> inventory = new();
}
```

1. ใช้ [SerializeField] กับ variables ที่ต้องการ edit ใน Inspector
2. ใช้ [Serializable] กับ custom classes/structs
3. ใช้ [Min]/[Max]/[Range] สำหรับ input validation
4. ใช้ [Header] สำหรับจัดกลุ่มใน Inspector

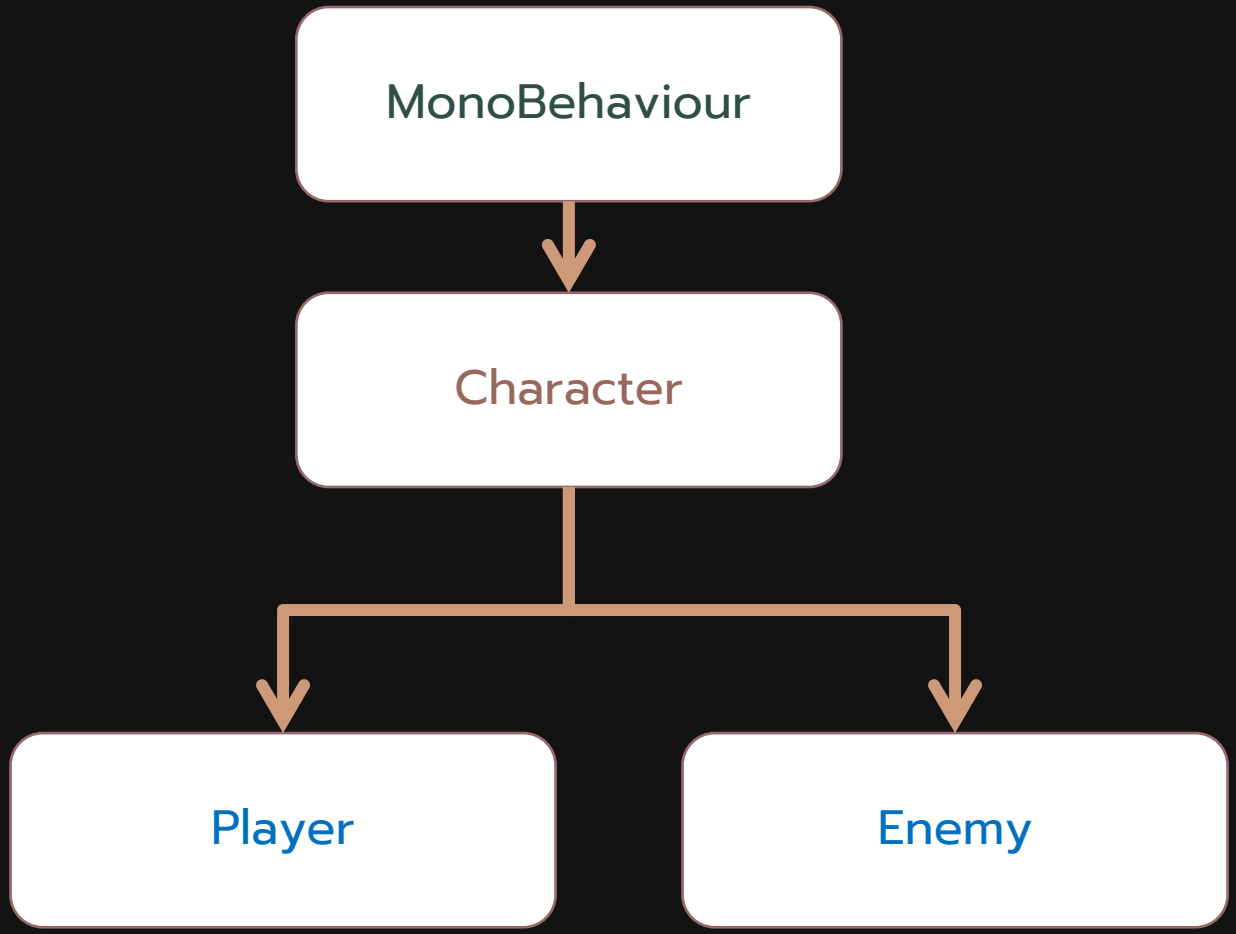
สรุป: ใช้เมื่อไหร่?

inheritance/ polymorphism



คลาสลูกสามารถรับสมาชิกจากคลาสแม่ได้

Inheritance (การสืบทอด)


MonoBehaviour
Character
Player
Enemy

```
// Base class
public class Character : MonoBehaviour
{
    public int health = 100;

    public void TakeDamage(int damage)
    {
        health -= damage;
    }
}
```

```
// Derived classes
public class Player : Character
{
    public new int maxHealth = 200; // override

    public void Heals()
    {
        /* heal logic */
    }
}
```

```
public class Enemy : Character
{
    public bool isAggressive = true;

    public void Attack()
    {
        /* attack logic */
    }
}
```

ส่งผ่าน object ของ base class แต่เรียก method ของ derived

Polymorphism (ความหลากหลาย)

```
void UpdateEnemies(List<Character> enemies)
{
    foreach (var enemy in enemies)
    {
        enemy.TakeDamage(10); // polymorphic call
    }
}
```

1. ใช้ BaseComponent สำหรับระบบทั่วไป
2. ใช้ [Serializable] สำหรับ serialization
3. ใช้ MonoBehaviour สำหรับทุก GameObject

Unity-specific Patterns

[Serializable] คือคำสั่งพิเศษ (Attribute) ที่บอกให้โปรแกรมรู้ว่าเราต้องการแปลงข้อมูลในคลาส (Class) หรือโครงสร้าง (Struct) ที่เราสร้างขึ้นเอง ให้กลายเป็นรูปแบบที่ Unity สามารถบันทึกเก็บไว้ในไฟล์ฉาก (Scene), ไฟล์พรีฟาบ (Prefab) หรือนำไปแสดงผลบนหน้าต่าง Inspector ได้

[Serializable] Attribute ใน C# สำหรับ Unity

```
using UnityEngine;

[Serializable]
public class PlayerData
{
    public string name;
    public int health;
    public float positionX;
}
```

Expose Variables ใน Inspector

```
[Serializable]
public class GameConfig : ScriptableObject
{
    [Header("Player Settings")]
    [SerializeField]
    private int playerHealth = 100;
    [SerializeField]
    private float maxSpeed = 5f;

    [Header("World Settings")]
    [SerializeField]
    private Vector3 worldSize = new Vector3(100, 200, 100);
}
```

Custom Serialization

```
[Serializable]
public class SaveData
{
    public List<string> items;
    public Dictionary<int, int> inventory;
    public PlayerData playerState;

    public void SetDefaults()
    {
        items = new List<string>();
        inventory = new Dictionary<int, int>();
        playerState = new PlayerData();
    }
}
```

1. บันทึกตัวแปรใน Unity Inspector ได้โดยตรง
2. ใช้กับ ScriptableObject สำหรับระบบ Save/Load
3. รองรับ nested types และ collections (List, Dictionary)

ข้อดี

1. ไม่รองรับ private fields โดยตรง ถ้าจะเป็นต้องกำหนดเป็น private ให้เปลี่ยนไปใช้ [SerializeField]
2. ไม่รองรับ custom structs ที่ซับซ้อนเกินไป
3. Memory overhead เล็กน้อยสำหรับ variables ที่ serialize

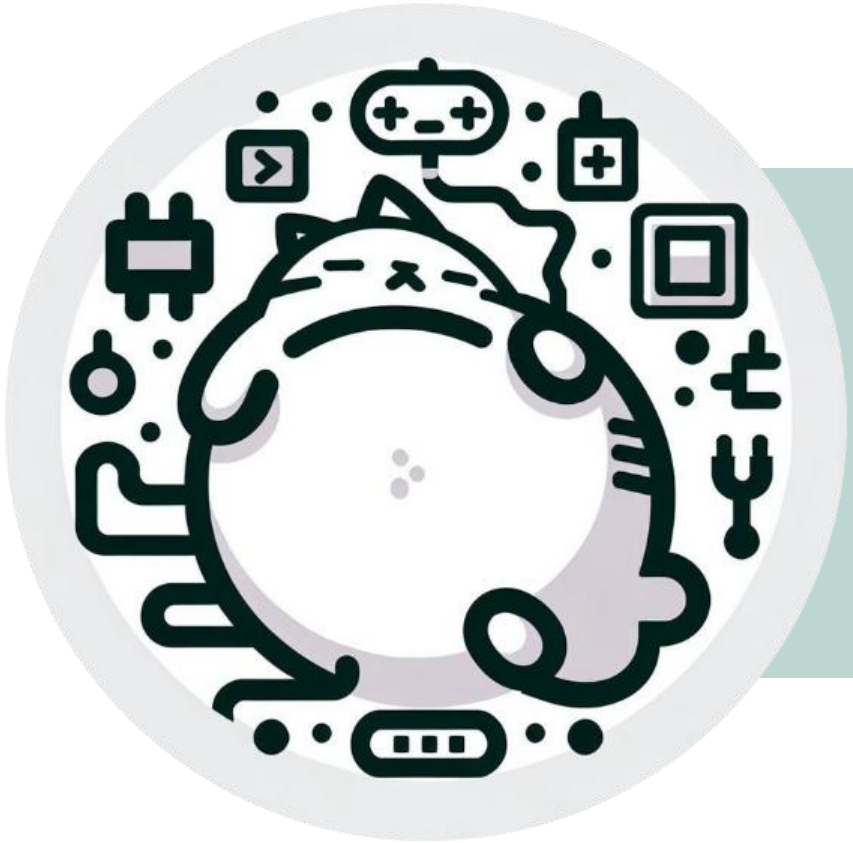
ข้อจำกัด

ตัวอย่าง Save System

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

[System.Serializable]
public class PlayerSave
{
    [SerializeField] private string playerName = "Player1";
    [SerializeField] private int currentHealth = 100;
    [SerializeField] private List<Item> inventory = new();

    public void Save()
    {
        // บันทึกข้อมูล
    }
}

[System.Serializable]
public class Item
{
    public string name;
    public int quantity;
}
```

1. ใช้ [SerializeField] แทน [Serializable] สำหรับ variables
2. ใช้ [System.Serializable] เฉพาะสำหรับ custom classes
3. หลีกเลี่ยง serialization ของ private fields ที่ซับซ้อน
4. ใช้ ScriptableObject สำหรับ config ที่ต้อง persist

Best Practices



GameObject & Components

GameObject และ Component เป็นแนวคิดพื้นฐานของสถาปัตยกรรม Unity โดย Unity ใช้แนวทาง Component-Based Architecture ในการประกอบวัตถุภายในเกม

GameObject และ Component

GameObject เป็น Entity ที่อยู่ใน Scene ส่วน **Component** เป็นหน่วยที่กำหนดคุณสมบัติและพฤติกรรมของ GameObject

GameObject และ Component

GameObject หนึ่งตัวสามารถประกอบด้วย Component หลายชนิด และแต่ละ Component รับผิดชอบหน้าที่เฉพาะด้าน

GameObject และ Component

GameObject **เป็น Object พื้นฐาน**ที่ Unity ใช้แทน Entity **ภายใน Scene** โดยที่GameObject เองไม่ได้มีความสามารถเฉพาะด้านมากนัก **แต่สามารถเพิ่มความสามารถผ่าน Components**

GameObject

ดังนั้น GameObject สามารถมองในเชิงสถาปัตยกรรมได้ว่า**เป็น Container ของ Components**

GameObject

Component คือ**โมดูลที่เพิ่มคุณสมบัติ**หรือ**พฤติกรรม**ให้กับ GameObject

Component

ตัวอย่าง Component มาตรฐานของ Unity

1. Transform
2. MeshFilter
3. MeshRenderer
4. Collider
5. Rigidbody
6. Animator
7. AudioSource
8. Camera
9. Light
10. ParticleSystem

Component	หน้าที่
Transform	Position, Rotation, Scale
MeshFilter	กำหนด Mesh
MeshRenderer	แสดงผล Mesh
Collider	Collision Geometry
Rigidbody	Physics Simulation
Animator	Animation
PlayerController	Game Logic

Component

Prefab (Prefabricated GameObject) คือ Asset ที่ใช้จัดเก็บ Template ของ GameObject พร้อม Components, Properties และโครงสร้างของ Child GameObjects เพื่อให้สามารถนำกลับมาใช้ซ้ำในหลายตำแหน่งหรือหลาย Scene ได้

Prefab

```
Instantiate(
        Prefab,
        position,
        rotation
);
```

```
using UnityEngine;

public class BulletController : MonoBehaviour
{
    [SerializeField]
    private float speed = 20f;

    [SerializeField]
    private float lifetime = 5f;

    private void Start()
    {
        Destroy(gameObject, lifetime);
    }

    private void Update()
    {
        transform.position +=
            transform.forward *
            speed *
            Time.deltaTime;
    }
}
```

โค้ดควบคุมกระสุนที่แนบเข้ากับ bulletPrefab และสั่งสร้างด้วย

```
Instantiate(
        bulletPrefab,
        muzzle.position,
        muzzle.rotation
);
```

```
using UnityEngine;

public class ProjectileSpawner : MonoBehaviour
{
    // อ้างอิงไปยัง Prefab Asset ในระบบของ Unity
    [SerializeField] private GameObject bulletPrefab;

    public void FireBullet()
    {
        // การเรียกใช้ Instantiate เพื่อสร้างวัตถุจริงจากแม่แบบ (Prefab)
        // โดยกำหนดตำแหน่งและทิศทางตามวัตถุต้นกำเนิด (this.transform)
        GameObject bullet = Instantiate(
                    bulletPrefab,
                    transform.position,
                    transform.rotation
          );

        // สามารถกำหนดค่าเฉพาะให้กับ Instance ใหม่ได้ทันทีหลังสร้าง
        bullet.name = "Bullet_" + System.DateTime.Now.Ticks;
    }
}
```

Prefab =

Reusable GameObject Template

เมื่อนำ prefab จาก Project Window ไปวางใน Scene จะเกิด Prefab Instance

Prefab Instance

Prefab Variant คือ Prefab ที่สร้างขึ้นจาก Prefab อื่น และสามารถปรับแต่งเฉพาะบางส่วนได้

Prefab Variant

ไม่ควรสร้าง Prefab ทุกอย่างโดยไม่มีเหตุผลเนื่องจาก Prefab เหมาะกับ Object ที่

1. ใช้ซ้ำ
2. Spawn Runtime
3. มีโครงสร้างซับซ้อน
4. ต้องการ Template
5. ต้องการ Instance หลายตัว

หาก Object มีเพียงหนึ่งตัวและเป็นองค์ประกอบเฉพาะของ Scene อาจไม่จำเป็นต้องแยกเป็น Prefab

ข้อควรระวังในการใช้ Prefab

ไม่ควรเก็บ Runtime State สำคัญไว้ใน Prefab Asset **ตัวอย่างที่ไม่เหมาะสม** เช่น Enemy.prefab มีกำหนด Current HP = 73 เพราะ Current HP เป็น Runtime State **ควรแยก** EnemyData.asset เก็บ Max HP = 100 กับ Enemy Instance เก็บ Current HP = 73

ข้อควรระวังในการใช้ Prefab

ใช้ **ScriptableObject** สำหรับ Shared Configuration เช่น EnemyData, WeaponData, SkillData, ItemData
ส่วน **Prefab** เก็บ Model, Collider, Animator, Controller, Audio, VFX

ข้อควรระวังในการใช้ Prefab

ตัวอย่าง

หากเกมมีศัตรูชนิดเดียวกัน 100 ตัว ไม่จำเป็นต้องสร้าง GameObject และกำหนด Components ทั้ง 100 ครั้ง แต่สามารถสร้าง Enemy.prefab เพียงครั้งเดียว แล้วสร้าง Instance จาก Prefab ดังกล่าว

ตัวอย่าง Prefab Variant สำหรับ RPG

Character.prefab
- Animator
- CharacterController
- HealthComponent
- MovementComponent
- CombatComponent

Character.prefab
- Warrior.prefab
- Mage.prefab
- Archer.prefab

Warrior.prefab
- Character
- Sword
- Warrior-specific configuration

Mage.prefab
- Character
- Staff
- Mage-specific configuration

Prefab = Structure+Components+Behavior

ScriptableObject = Configuration/Data

GameObject = RuntimeEntity

GameObject

คือตัวตนของวัตถุในขณะที่ปรากฏในฉาก (Instance)

Component

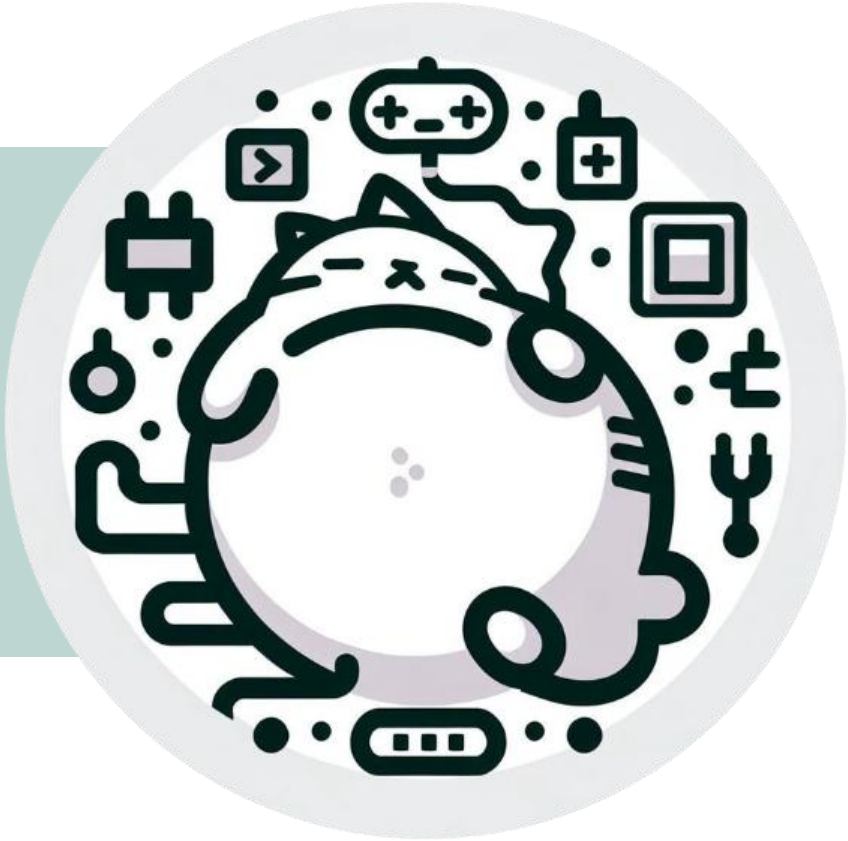
คือพฤติกรรมหรือคุณสมบัติที่ติดอยู่บน GameObject

Prefab

คือพิมพ์เขียว (Blueprint) ที่ใช้เก็บข้อมูลของ GameObject และ Component เพื่อการนำกลับมาใช้ซ้ำและการจัดการที่เป็นระบบ

MonoBehaviour

class



MonoBehaviour เป็นคลาสพื้นฐานของ Unity Engine ที่ทำหน้าที่เป็นหน่วยประมวลผลหลักในการจัดการวงจรชีวิต เหตุการณ์ และระบบส่วนประกอบของ GameObject ทั้งหมดใน Scene

MonoBehaviour

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    private float currentHealth;

    // Constructor และ Initialization
    void Start()
    {
        Initialize();
    }
}
```

Lifecycle Methods

Method	ความถี่	วัตถุประสงค์	ตัวอย่างการใช้งาน
Awake()	ครั้งเดียว	Initial setup ก่อน Start	การเตรียมข้อมูลเริ่มต้น
Start()	ครั้งเดียว	หลัง GameObject ถูก Instantiate	การเริ่มกระบวนการเกม
OnEnable()	หลายครั้ง	เมื่อ GameObject เปิดใช้งาน	Reactivate logic

Initialization Sequence

```
public class Example : MonoBehaviour
{
    void Awake()
    {
        // Initial setup (เรียกก่อน Start)
        Debug.Log("Awake - Object created");
    }

    void Start()
    {
        // Initialization after dependencies ready
        Debug.Log("Start - Ready to begin");
    }
}
```

Method	ความถี่	วัตถุประสงค์	ตัวอย่างการใช้งาน
Update()	ทุกเฟรม (~60Hz)	Logic ที่ต้องรันทุก frame	Movement, input handling
FixedUpdate()	Fixed timestep	Physics simulation	Rigidbody movement

Update Loop

```
Using UnityEngine;
Using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
    [SerializeField] private float speed = 5f;

    void Update()
    {
        if (Keyboard.current.wKey.isPressed))
        {
            transform.Translate(Vector3.forward * speed * Time.deltaTime);
        }
    }

    void FixedUpdate()
    {
        // Physics-based calculations
        Rigidbody rb = GetComponent<Rigidbody>();
        rb.velocity = new Vector3(0, rb.velocity.y, speed);
    }
}
```

Method	วัตถุประสงค์	ตัวอย่างการใช้งาน
OnDisable()	เมื่อ GameObject ถูก disable	Pause logic
OnDestroy()	เมื่อ Object ถูกลบ	Cleanup resources
OnApplicationPause()	Game paused/resumed	Save/Load handling

Cleanup Methods

```
public class GameManager : MonoBehaviour
{
    void OnEnable()
    {
        // Resume game logic
        Debug.Log("Game resumed");
    }

    void OnDestroy()
    {
        // Cleanup references, unsubscribe events
        Debug.Log("Game destroyed - cleanup complete");
    }
}
```

Component System Architecture

```
public class ComponentExample : MonoBehaviour
{
    // Get component ของ GameObject ปัจจุบัน
    void GetComponentExample()
    {
        Rigidbody rb = GetComponent<Rigidbody>();
        AudioSource audio = GetComponent<AudioSource>();
    }

    // Get component ของ GameObject อื่น
    void GetComponentOnOtherGameObjectExample(GameObject target)
    {
        Transform transform = target.GetComponent<Transform>();
        Collider collider = target.GetComponent<Collider>();
    }
}
```

Method	วัตถุประสงค์	ตัวอย่าง
GetComponent<T>()	Get component ของ GameObject ปัจจุบัน	transform.GetComponent<Rigidbody>()
GetComponentInChildren<T>()	Get component ของทุก child	Camera mainCam = GetComponentInChildren<Camera>()
GetComponentInParent<T>()	Get component ของ parent	AudioListener listener = GetComponentInParent<AudioListener>()

Component Search Methods

```
public class CameraController : MonoBehaviour
{
    void FindComponents()
    {
        // Current GameObject components
        Camera mainCamera = GetComponent<Camera>();

        // Child components
        Transform[] allTransforms = GetComponentsInChildren<Transform>();

        // Parent components
        AudioManager audioManager = GetComponentInParent<AudioManager>();
    }
}
```

Unity Event Methods

Collision Events

```
public class CollisionExample : MonoBehaviour
{
    void OnCollisionEnter(Collision collision)
    {
        Debug.Log("Collision with: " + collision.gameObject.name);
    }

    void OnCollisionStay(Collision collision)
    {
        // During collision
    }

    void OnCollisionExit(Collision collision)
    {
        Debug.Log("Lost contact with: " + collision.gameObject.name);
    }
}
```

Physics Events

```
public class RigidbodyExample : MonoBehaviour
{
    private Rigidbody rb;

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

    void OnTriggerEnter(Collider other)
    {
        Debug.Log("Trigger entered: " + other.gameObject.name);
    }
}
```

ScriptableObject และ Data Container มีความสำคัญต่อการออกแบบสถาปัตยกรรมเกม โดยเฉพาะเมื่อเกมมีข้อมูลจำนวนมากที่ต้องใช้ร่วมกันระหว่างหลายระบบ เช่น ข้อมูลตัวละคร อาวุธ ไอเทม Skill ศัตรู Dialogue Quest และค่าการตั้งค่าของเกม

ScriptableObjects และ Data Containers

แนวคิดสำคัญ คือ

แยกข้อมูล (Data)

ออกจาก

พฤติกรรม (Behavior)

โครงสร้างหรือวัตถุที่มีหน้าที่หลักในการ จัดเก็บข้อมูล โดยไม่รับผิดชอบต่อ Logic ของระบบ เช่นตัวอย่างตัวละคร (Character) ประกอบด้วย Name, HP, MP, Attack, Defense, Speed และ CriticalRate เป็นต้น

Data Container

```
[System.Serializable]
public class CharacterData
{
    public string characterName;
    public int maxHP;
    public int maxMP;
    public int attack;
    public int defense;
    public float speed;
}
```

คลาส CharacterData ทำหน้าที่เป็น Data Container แต่ CharacterData ไม่ได้ทำหน้าที่ เช่น โจมตี, เคลื่อนที่, รับ Damage หรือ เล่น Animation เป็นต้น เนื่องจากหน้าที่เหล่านี้ควรอยู่ในระบบ Behavior

ปัญหาของการเก็บข้อมูลไว้ใน MonoBehaviour

พิจารณาเกม RPG ที่มีอาวุธจำนวนมาก

```
public class Weapon : MonoBehaviou
{
    public string weaponName;
    public int damage;
    public float attackSpeed;
    public int price;
}
```

หากมี Sword, Axe, Bow, Staff, Dagger, ... นักพัฒนาอาจต้องสร้าง Prefab หลายตัวเพียงเพื่อเก็บข้อมูล Sword.prefab Axe.prefab Bow.prefab Staff.prefab Dagger.prefab **ปัญหาคือ** ข้อมูลถูกผูกติดกับ GameObject **ทั้งที่**ข้อมูลอาวุธอาจไม่จำเป็นต้องเป็น GameObject

Sword

Damage = 20

Price = 100

Attack Speed = 1.2

เป็นเพียงข้อมูลเชิงสถิตศาสตร์ ไม่จำเป็นต้องมี Transform, GameObject หรือ Component จึงเกิดแนวคิดในการแยก

GameObject

จัดการเรื่องพฤติกรรม

ScriptableObject

จัดการเรื่องข้อมูล

ScriptableObject เป็นคลาสพื้นฐานของ Unity สำหรับสร้าง Data Asset ที่สามารถจัดเก็บข้อมูลเป็น Asset ภายใน Project

ScriptableObject

```
using UnityEngine;

[CreateAssetMenu(
    fileName = "New Weapon",
    menuName = "Game/Weapon Data“
)]

public class WeaponData : ScriptableObject
{
    public string weaponName;
    public int damage;
    public float attackSpeed;
    public int price;
}
```

จากนั้นสามารถสร้าง Asset ใน Unity Editor ให้มีโครงสร้างดังนี้

Project
Assets
Data
Weapons
Sword.asset
Axe.asset
Bow.asset
Staff.asset

โดยแต่ละ .asset จะเป็น Instance ของ WeaponData

GameObject

1. Transform
2. Component
3. Position
4. Rotation
5. Scene

ScriptableObject

1. Data
2. Asset
3. Shared Reference

ดังนั้น ScriptableObject ไม่จำเป็นต้องอยู่ใน Scene

Scene

1. Player
2. Enemy
3. Camera
4. UI

Project Assets

WeaponData

1. Sword.asset
2. Axe.asset
3. Bow.asset

ตัวอย่าง Weapon Data

สร้าง ScriptableObject

```
using UnityEngine;

[CreateAssetMenu(
    fileName = "WeaponData",
    menuName = "Game Data/Weapon"
)]
public class WeaponData : ScriptableObject
{
    [Header("Basic Information")]
    public string weaponName;

    public Sprite icon;

    [Header("Combat")]
    public int damage;

    public float attackSpeed;

    public float range;

    [Header("Economy")]
    public int price;
}
```

ขั้นตอนในการสร้าง

1. Right Click
2. Create
3. Game Data
4. Weapon

แล้วกำหนดค่า เช่น

Sword.asset

1. weaponName = Iron Sword
2. damage = 25
3. attackSpeed = 1.2
4. range = 1.5
5. price = 100

Behavior สามารถ Reference Data Container

```
using UnityEngine;

public class WeaponController : MonoBehaviour
{
    [SerializeField]
    private WeaponData weaponData;

    public void Attack()
    {
        Debug.Log(
            $"Attack Damage: {weaponData.damage}"
        );
    }
}
```

โครงสร้างจึงเป็น WeaponController อ้างอิง (references) ไปยัง WeaponData ที่ประกอบไปด้วย damage, attackSpeed, range และ price

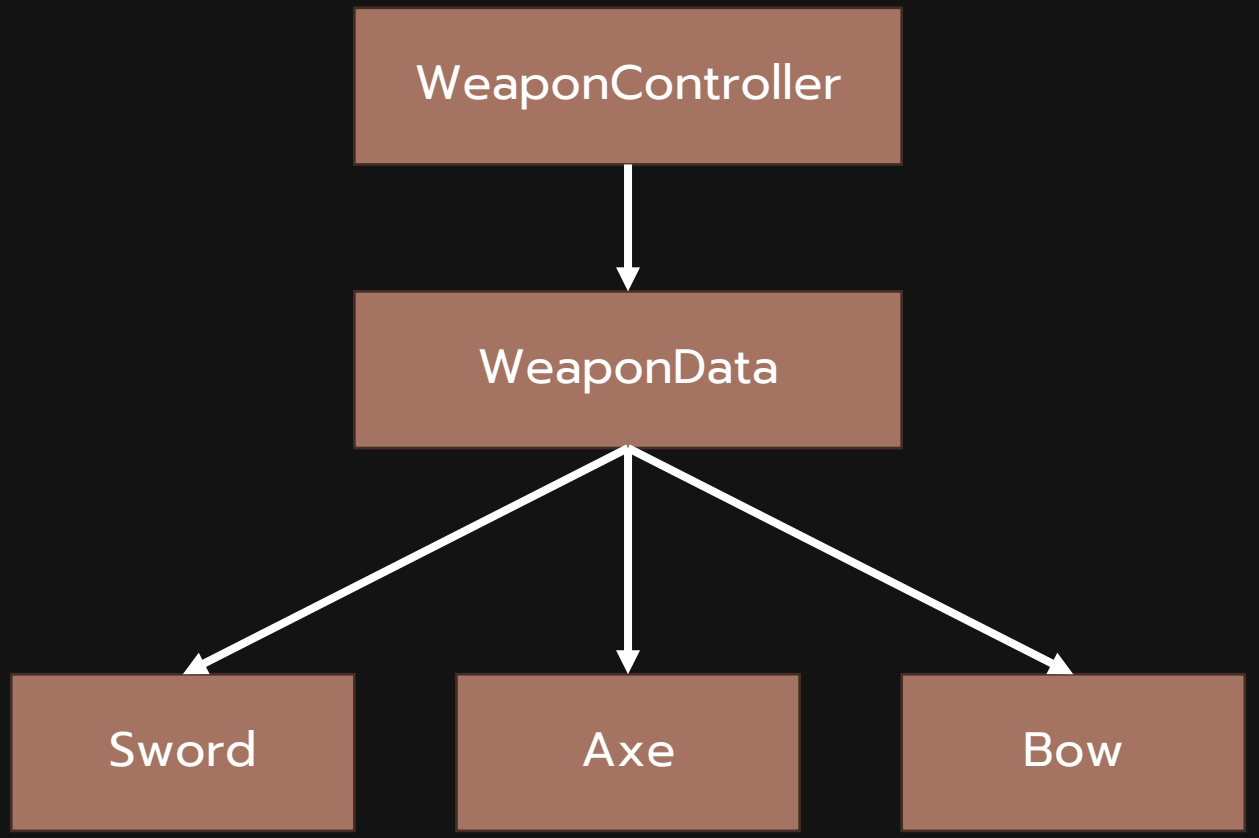
Data-Driven Design

แทนที่จะเขียนค่าต่าง ๆ ลงใน Logic

```
if (weaponType == "Sword")
{
    damage = 25;
}
else if (weaponType == "Axe")
{
    damage = 40;
}
```

สามารถใช้ Data Asset

```
public class WeaponController : MonoBehaviour
{
    [SerializeField]
    private WeaponData data;

    public void Attack()
    {
        int damage = data.damage;

        // Combat Logic
    }
}
```



ทำให้ Logic ไม่จำเป็นต้องรู้ว่าเป็น Sword, Axe หรือ Bow นี่เป็นแนวทางที่มีประสิทธิภาพมากสำหรับเกมที่มีข้อมูลจำนวนมาก

ตัวอย่าง EnemyData

```
using UnityEngine;

[CreateAssetMenu(
    fileName = "EnemyData",
    menuName = "Game Data/Enemy"
)]
public class EnemyData : ScriptableObject
{
    public string enemyName;

    public int maxHP;

    public int attack;

    public int defense;

    public float moveSpeed;

    public float attackRange;

    public int experienceReward;
}
```

จากนั้นสามารถสร้าง:

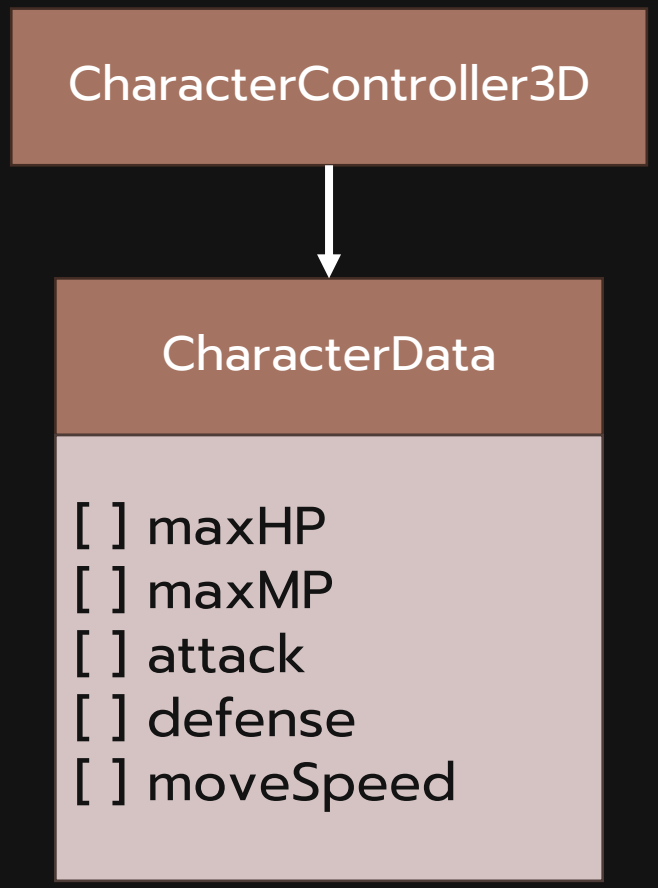
```
EnemyData
│
├── Goblin.asset
├── Orc.asset
├── Skeleton.asset
├── Slime.asset
└── Dragon.asset
```

โดยไม่ต้องสร้าง Component ที่แตกต่างกันสำหรับศัตรูแต่ละชนิด

ตัวอย่างระบบ Character

```
using UnityEngine;

[CreateAssetMenu(
    fileName = "CharacterData",
    menuName = "Game Data/Character"
)]
public class CharacterData : ScriptableObject
{
    public string characterName;

    public int maxHP;
    public int maxMP;

    public int attack;
    public int defense;

    public float moveSpeed;

    public Sprite portrait;
}
```

```
using UnityEngine;

public class CharacterController3D : MonoBehaviour
{
    [SerializeField]
    private CharacterData data;

    private int currentHP;

    private void Start()
    {
        currentHP = data.maxHP;
    }

    public void TakeDamage(int damage)
    {
        int finalDamage =
            Mathf.Max(0, damage - data.defense);

        currentHP -= finalDamage;
    }
}
```


CharacterController3D
CharacterData
[ ] maxHP
[ ] maxMP
[ ] attack
[ ] defense
[ ] moveSpeed

ใช้กับ Ability/Skill

```
using UnityEngine;

[CreateAssetMenu(
    fileName = "SkillData",
    menuName = "Game Data/Skill"
)]
public class SkillData : ScriptableObject
{
    public string skillName;

    public Sprite icon;

    public int manaCost;

    public float cooldown;

    public int damage;

    public float range;

    [TextArea]
    public string description;
}
```

สามารถสร้าง

```
Skills
├── Fireball.asset
├── IceBolt.asset
├── Heal.asset
├── Thunder.asset
└── Meteor.asset
```

```
public class SkillController : MonoBehaviour
{
    [SerializeField]
    private SkillData skill;

    public void Cast()
    {
        Debug.Log(
            $"Cast {skill.skillName}"
        );

        // ใช้ skill.damage
        // ใช้ skill.manaCost
        // ใช้ skill.cooldown
    }
}
```

SkillController สามารถใช้ข้อมูล

สามารถสร้าง

```
Skills
├── Fireball.asset
├── IceBolt.asset
├── Heal.asset
├── Thunder.asset
└── Meteor.asset
```

ระบบ Inventory

```
ItemData
    ├── ID
    ├── Name
    ├── Icon
    ├── Description
    ├── Price
    ├── Weight
    └── ItemType
```

จากนั้น Inventory ไม่จำเป็นต้องเก็บข้อมูลทุกอย่างเอง

```
Inventory
    ├── ItemData → Potion
    ├── ItemData → Sword
    ├── ItemData → Herb
    └── ItemData → Armor
```

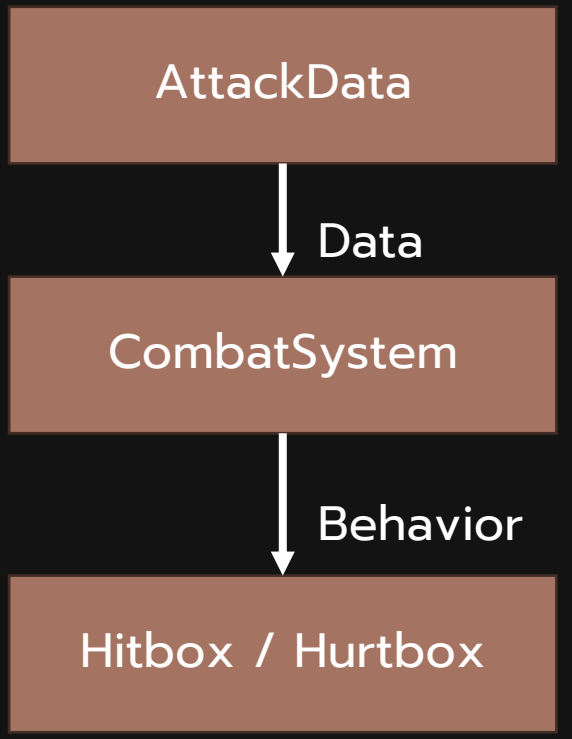
```
using UnityEngine;
public enum ItemType
{
    Consumable,
    Weapon,
    Armor,
    Quest,
    Material
}
[CreateAssetMenu(
    fileName = "ItemData",
    menuName = "Game Data/Item"
)]
public class ItemData : ScriptableObject
{
    public string itemID;
    public string itemName;
    public Sprite icon;

    [TextArea]
    public string description;
    public ItemType itemType;
    public int price;
}
```

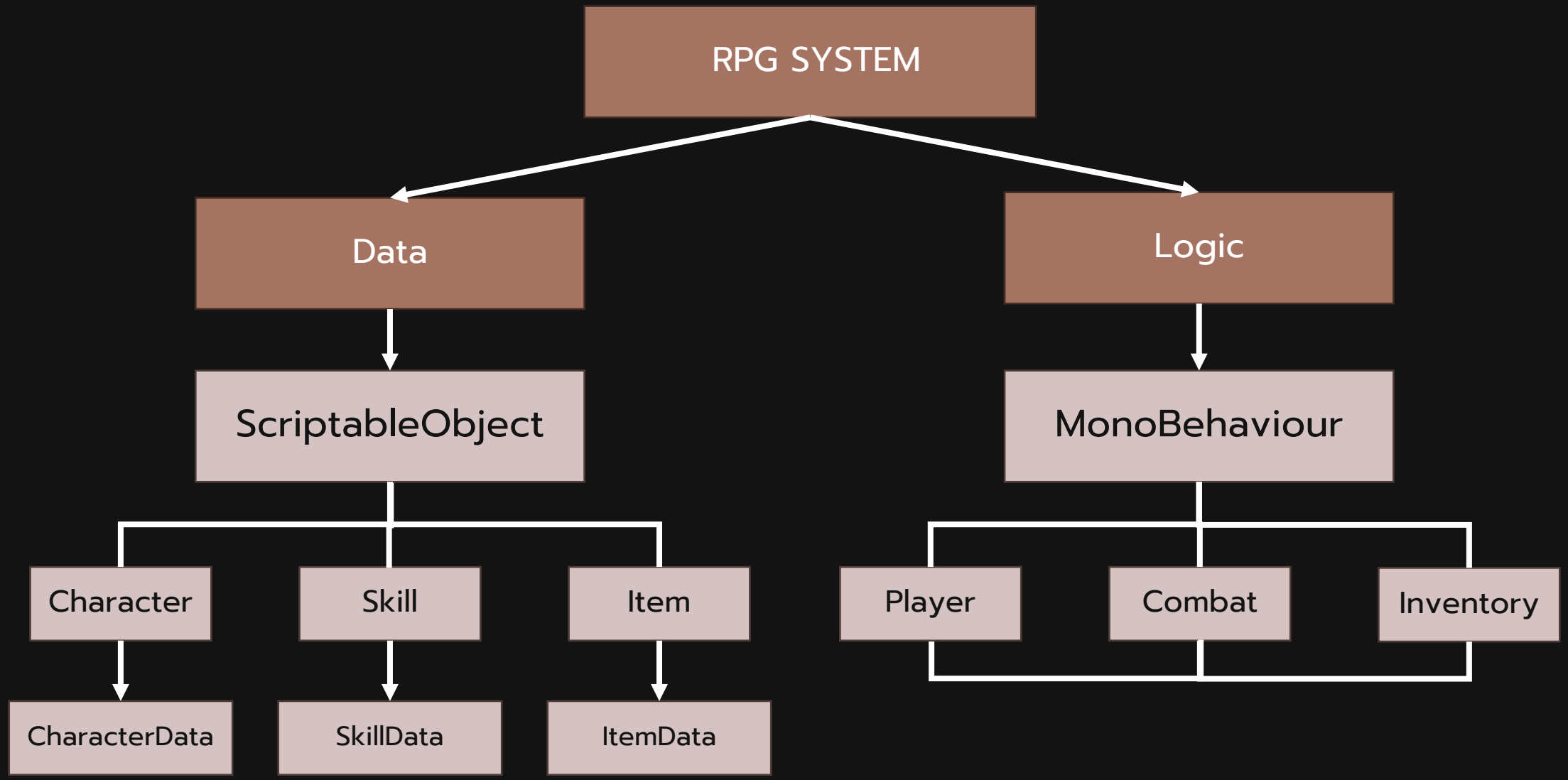
```cs
using UnityEngine;

[CreateAssetMenu(
    fileName = "AttackData",
    menuName = "Combat/Attack"
)]
public class AttackData : ScriptableObject
{
    public string attackName;

    public int damage;

    public float staminaCost;

    public float startupTime;

    public float activeTime;

    public float recoveryTime;

    public float knockbackForce;
}
```

ระบบ CombatSystem

จากนั้น CombatSystem ทำหน้าที่ประมวลผล:

```
public class CombatSystem : MonoBehaviour
{
    [SerializeField]
    private AttackData attackData;


    public void PerformAttack()
    {
        // อ่านข้อมูลจาก attackData
        // แล้วดำเนิน Combat Logic
    }
}
```




RPG SYSTEM
Data
Logic
ScriptableObject
MonoBehaviour
Character
Skill
Item
Player
Combat
Inventory
CharacterData
SkillData
ItemData

Component Separation Pattern

```
// Data Component - เก็บข้อมูลและ configuration
public class PlayerData : MonoBehaviour
{
    [SerializeField] private float maxHealth;
    [SerializeField] private int currentLevel;
}

// Logic Component - เก็บพฤติกรรมและ methods
public class PlayerController : MonoBehaviour
{
    private void Update() { }
}

// Event Handler - จัดการ events และ delegates
public class PlayerEvents : MonoBehaviour
{
    private void OnEvent() { }
}
```

Singleton Pattern สำหรับ Global State

```
using UnityEngine;

public sealed class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }

    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
            Destroy(gameObject);
    }
}
```

ข้อควรระวัง

Memory Management

```
public class MemoryExample : MonoBehaviour
{
    private void Update()
    {
        // ไม่ดี - เก็บ reference ไว้จนเกิด memory leak
        if (someCondition)
        {
            Destroy(someGameObject); // ไม่จำเป็นต้องใช้ Destroy
        }

        // ดี - ใช้ ref เสร็จแล้ว set เป็น null
        someReference = null;
    }
}
```

Null Safety

```
public class NullSafety : MonoBehaviour
{
    private void Update()
    {
        // ไม่ดี - อาจเกิด NullReferenceException
        Transform target = someParent.FindChild("Target");
        target.Translate(...); // ถ้า Target ไม่มี children จะ error

        // ดี - ตรวจสอบก่อนใช้งาน
        if (target != null)
        {
            target.Translate(...);
        }
    }
}
```

Coroutine

Coroutine เป็นกลไกสำหรับควบคุมการทำงานของเมธอดให้สามารถ หยุดการทำงานชั่วคราว (suspend) ณ จุดที่กำหนด และกลับมาทำงานต่อในภายหลัง โดยไม่จำเป็นต้องสร้าง Thread ใหม่ กลไกดังกล่าวมีประโยชน์อย่างมากสำหรับการพัฒนาเกมที่ต้องจัดการงานซึ่งมีระยะเวลา เช่น การหน่วงเวลา การเปลี่ยนสถานะของเกม การทำ Animation แบบเป็นลำดับ การสร้างศัตรูเป็นช่วงเวลา การ Fade UI และการดำเนินกระบวนการหลายขั้นตอน

Coroutine

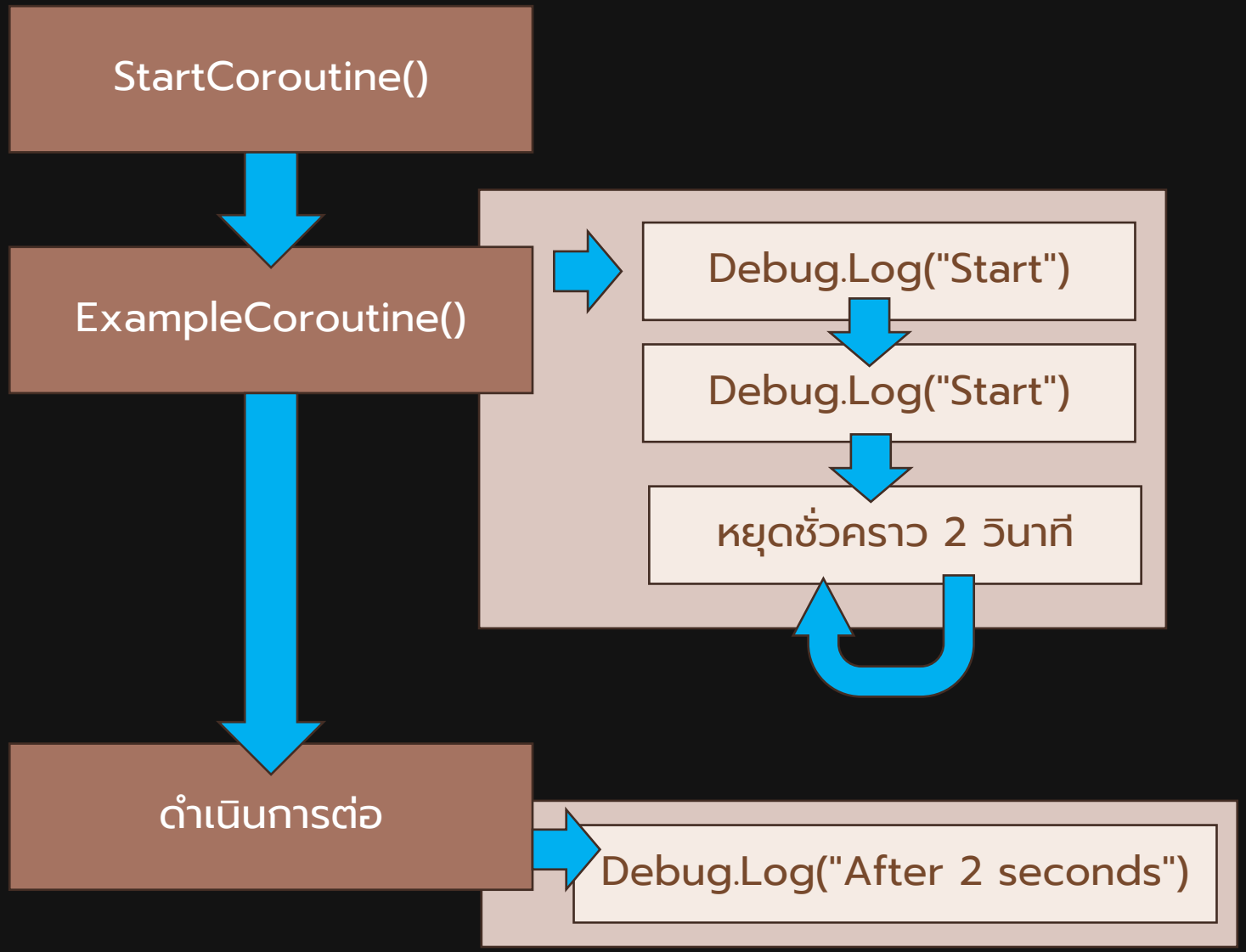
ใน Unity 6.3 Coroutine มักถูกเขียนในรูปของเมธอดที่คืนค่า IEnumerator และใช้คำสั่ง yield return เพื่อระบุจุดที่ Coroutine จะหยุดชั่วคราว

ตัวอย่างโครงสร้างพื้นฐาน

```
IEnumerator ExampleCoroutine()
{
    Debug.Log("Start");

    yield return new WaitForSeconds(2.0f);

    Debug.Log("After 2 seconds");
}
```

การเรียกใช้งาน

```
StartCoroutine(ExampleCoroutine());
```



ประเด็นสำคัญคือ yield return **ไม่ได้ทำให้ Thread หยุดทำงาน** แต่เป็นการแจ้ง Unity ว่า Coroutine ควรหยุดดำเนินการชั่วคราว และ Unity จะกลับมาเรียกดำเนินการต่อเมื่อเงื่อนไขของ yield นั้นเป็นจริง

ใน Unity 6.3 Coroutine มักถูกเขียนในรูปของเมธอดที่คืนค่า IEnumerator และใช้คำสั่ง yield return เพื่อระบุจุดที่ Coroutine จะหยุดชั่วคราว

ในเกมมีงานจำนวนมากที่ไม่ควรดำเนินการเสร็จภายในเฟรมเดียว เช่น

1. รอเวลา 3 วินาที
2. แสดงข้อความทีละตัวอักษร
3. Fade หน้าจอ
4. Spawn ศัตรูทุก ๆ 2 วินาที
5. รอให้ Animation สิ้นสุด
6. รอให้เงื่อนไขบางอย่างเป็นจริง
7. ดำเนิน Sequence ของเหตุการณ์
8. แสดง Dialogue
9. เปลี่ยนค่า UI อย่างต่อเนื่อง

หากดำเนินการด้วยโค้ดแบบต่อเนื่องในเมธอดเดียว อาจทำให้เกิดการวนลูปที่ใช้เวลานานและส่งผลต่อ Frame Rate

```
void BadExample()
{
    float startTime = Time.time;

    while (Time.time - startTime < 3.0f)
    {
        // รอ 3 วินาที
    }

    Debug.Log("Complete");
}
```

ตัวอย่างที่ ไม่เหมาะสม

โค้ดเป็น **Busy Waiting** ซึ่งทำให้ Main Thread ทำงานวนซ้ำโดยไม่เปิดโอกาสให้ Unity ประมวลผลส่วนอื่นตามปกติ

Coroutine

```
IEnumerator GoodExample()
{
    yield return new WaitForSeconds(3.0f);

    Debug.Log("Complete");
}
```

Coroutine โดยทั่วไปมีองค์ประกอบ 3 ส่วน

```
IEnumerator CoroutineName()
{
    // ขั้นตอนที่ 1
    yield return ...;
    // ขั้นตอนที่ 2
    yield return ...;
    // ขั้นตอนที่ 3
}
```

โครงสร้างของ Coroutine

```
IEnumerator GameSequence()
{
    Debug.Log("Prepare");
    yield return new WaitForSeconds(2.0f);

    Debug.Log("Start");
    yield return new WaitForSeconds(3.0f);

    Debug.Log("Finish");
}
```

เรียกใช้งาน

```
StartCoroutine(GameSequence());
```

Coroutine ใน Unity ใช้ IEnumerator เนื่องจาก Coroutine ต้องสามารถคืนการควบคุมกลับไปยัง Unity และกลับมาดำเนินการต่อในภายหลัง

IEnumerator

การใช้ IEnumerator ไม่ได้หมายความ
ว่า Coroutine เป็น Thread

Coroutine โดยทั่วไปทำงานร่วมกับ Unity Main Thread / Player Loop ไม่ได้สร้าง CPU Thread แยกสำหรับ Coroutine

yield return null เป็นรูปแบบที่ใช้บ่อยมากสำหรับการหยุด Coroutine จนถึงเฟรมถัดไป จึงเหมาะกับงานที่ต้องการดำเนินการแบบ Frame-by-Frame

yield return null

```
IEnumerator Example()
{
    Debug.Log("Frame A");

    yield return null;

    Debug.Log("Frame B");

    yield return null;

    Debug.Log("Frame C");
}
```

WaitForSeconds()
ใช้สำหรับหยุด Coroutine เป็นระยะเวลาที่กำหนด

WaitForSeconds

ใช้สร้างระบบ Spawn

```
IEnumerator SpawnEnemies()
{
    while (true)
    {
        SpawnEnemy();
        yield return new WaitForSeconds(2.0f);
    }
}
```

เริ่มทำงาน

```
StartCoroutine(SpawnEnemies());
```

โค้ดนี้ทำให้มีการเรียกใช้ SpawnEnemy() ทุก 2 วินาที (เพราะใช้ while (true))

1. WaitForSeconds ขึ้นอยู่กับ Time.timescale ดังนั้น หากเกมหยุดด้วย Time.timeScale = 0f;
2. Coroutine ที่ใช้ WaitForSeconds จะไม่ดำเนินเวลาต่อ ในแบบปกติหากต้องการเวลาจริงของระบบ ให้ใช้ yield return new WaitForSecondsRealtime(2.0f);

WaitForSecondsRealtime

ใช้สร้างระบบ Spawn

```
IEnumerator ShowNotification()
{
    notification.SetActive(true);

    yield return new WaitForSecondsRealtime(3.0f);

    notification.SetActive(false);
}
```

สั่งเปิด UI สำหรับ Notify แล้วรอ 3 วินาที หลังจากนั้นสั่งปิด UI

WaitUntil ใช้รอจนกว่าเงื่อนไขจะเป็น true

WaitUntil

```
bool isReady = false;

IEnumerator WaitForReady()
{
    yield return new WaitUntil(() => isReady);
    Debug.Log("System is ready");
}
```

เมื่อระบบอื่นกำหนด isReady = true;
Coroutine จะดำเนินการต่อ

ใช้กับระบบ Game State

```
IEnumerator WaitForGameStart()
{
    yield return new WaitUntil(() => GameManager.IsGameStarted);
    StartGameplay();
}
```

WaitWhile ตรงกันข้ามกับ WaitUntil คือ

WaitWhileyield return new WaitWhile(() => isLoading);

หมายความว่า Coroutine จะหยุดรอ ตราบใดที่ isLoading == true

WaitWhile

```
IEnumerator WaitForLoading()
{
    yield return new WaitWhile(() => isLoading);

    Debug.Log("Loading Complete");
}
```

เมื่อ isLoading = false;Coroutine จึงดำเนินการต่อ

Coroutine เหมาะสำหรับการเปลี่ยนค่าต่อเนื่อง เช่น การเคลื่อนที่ของวัตถุ

```
IEnumerator MoveObject(    Transform target,    Vector3 destination,
float duration)
{
    Vector3 start = target.position;
    float elapsed = 0f;
    while (elapsed < duration)
    {
        elapsed += Time.deltaTime;
        float t = elapsed / duration;
        target.position = Vector3.Lerp(start, destination, t);
        yield return null;
    }
    target.position = destination;
}
```

```
StartCoroutine(
    MoveObject(
        transform,
        new Vector3(10f, 0f, 0f),
        2f
    )
);
```

```
IEnumerator FadeIn(
    CanvasGroup canvasGroup,
    float duration)
{
    float elapsed = 0f;
    canvasGroup.alpha = 0f;
    while (elapsed < duration)
    {
        elapsed += Time.deltaTime;
        canvasGroup.alpha = Mathf.Clamp01(elapsed / duration);

        yield return null;
    }
    canvasGroup.alpha = 1f;
}
```

```
StartCoroutine(     FadeIn(canvasGroup, 1.0f));
```

```
IEnumerator FadeOut(
    CanvasGroup canvasGroup,
    float duration)
{
    float elapsed = 0f;
    while (elapsed < duration)
    {
        elapsed += Time.deltaTime;
        canvasGroup.alpha =  1f - Mathf.Clamp01(elapsed / duration);
        yield return null;
    }
    canvasGroup.alpha = 0f;
}
```

```
StartCoroutine( FadeOut(canvasGroup, 1.0f));
```

Coroutine มีประโยชน์อย่างมากกับระบบ Dialogue
โดยเฉพาะการแสดงข้อความทีละตัวอักษร

Coroutine สำหรับ Dialogue

```
IEnumerator TypeText(
    string text,
    float interval)
{
    dialogueText.text = "";

    foreach (char c in text)
    {
        dialogueText.text += c;
        yield return new WaitForSeconds(interval);
    }
}
```

```
StartCoroutine(
    TypeText(
        "Welcome to the game.",
        0.05f
    )
);
```

Unity สามารถหยุด Coroutine ได้ด้วย

StopCoroutine()

หรือ

StopAllCoroutines();

การหยุด Coroutine

```
Coroutine currentCoroutine;void Start()
{
    currentCoroutine = StartCoroutine(MyCoroutine());
}
```

จากนั้น

```
void StopMyCoroutine()
{
    if (currentCoroutine != null)
    {
        StopCoroutine(currentCoroutine);
        currentCoroutine = null;
    }
}
```

รูปแบบนี้มีข้อดีมากกว่าการใช้ชื่อเมธอดโดยตรงในกรณีที่ระบบมี Coroutine หลายตัว

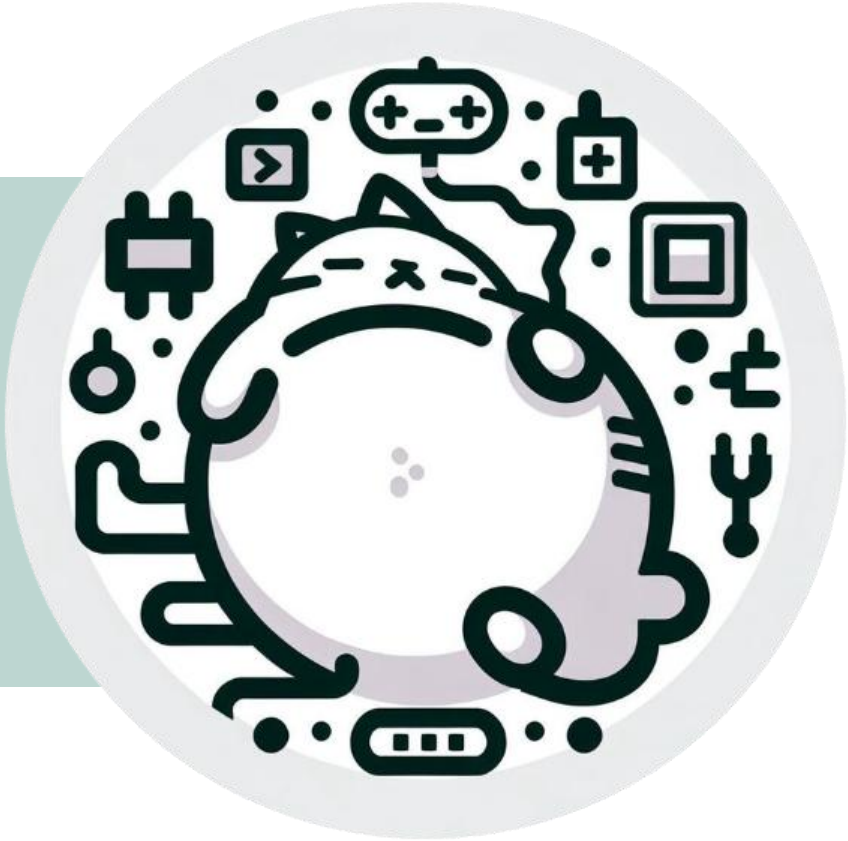
อย่างไรก็ตาม ควรใช้ด้วยความระมัดระวัง เนื่องจากคำสั่งนี้จะหยุด ทุก Coroutine ที่ทำงานโดย MonoBehaviour ตัวนั้น ตัวอย่างเช่น หากมี

```
EnemyController
  ├── AttackCoroutine
  ├── PatrolCoroutine
  └── AnimationCoroutine
```

การเรียก StopAllCoroutines(); จะหยุดทั้งหมดดังนั้น ในระบบขนาดใหญ่ควรจัดการ Coroutine เป็นรายตัวเมื่อมีความจำเป็น

Transform

position, rotation, scale



Transform =

Position

+ Rotation

+ Scale

Transform คือ Component พื้นฐานของ GameObject ที่ทำหน้าที่จัดการ spatial positioning, orientation และ scale ใน 3D space

Transform

```
using UnityEngine;

public class TransformExample : MonoBehaviour
{
    // Transform มีตำแหน่ง (position) มุม (rotation) และการขยาย (scale)
    void Update()
    {
        Debug.Log("Position: " + transform.position);
        Debug.Log("Rotation: " + transform.rotation);
        Debug.Log("Scale: " + transform.localScale);
    }
}
```

Transform Position
(ตำแหน่ง)

Coordinate System	คำอธิบาย	ตัวอย่างค่าเริ่มต้น
World Space	ตำแหน่งสัมพัทธ์กับ Scene Origin (0, 0, 0)	new Vector3(0f, 5f, 0f)
Local Space	ตำแหน่งสัมพัทธ์กับ Parent GameObject	new Vector3(1f, 0f, 0f)

Coordinate Systems

การเข้าถึง Position

```
public class PositionExample : MonoBehaviour
{
    // อ่านค่าตำแหน่งปัจจุบัน (World Space)
    public float GetXPosition() { return transform.position.x; }
    public float GetYPosition() { return transform.position.y; }
    public float GetZPosition() { return transform.position.z; }

    // เขียนค่าตำแหน่งใหม่
    public void SetPositionToZero()
    {
        transform.position = Vector3.zero;
    }
    public void MoveToVector3(Vector3 target)
    {
        transform.position = target;
    }
}
```

Local Space Movement

```
public class LocalMovement : MonoBehaviour
{
    [SerializeField]
    private float moveSpeed = 5f;

    void Update()
    {
        // เคลื่อนที่ใน Local Space
        transform.position += transform.right * moveSpeed * Time.deltaTime;

        // หรือใช้ Translate
        transform.Translate(
                Vector3.right * moveSpeed * Time.deltaTime,
                Space.Self
        );
    }
}
```

World Space Movement

```
public class WorldMovement : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 5f;

    void Update()
    {
        // เคลื่อนที่ใน World Space
        transform.position += Vector3.right * moveSpeed * Time.deltaTime;

        // หรือใช้ Translate
        transform.Translate(
                Vector3.right * moveSpeed * Time.deltaTime,
                Space.World
        );
    }
}
```

Relative Movement
(Local to World)

```
public class RelativeMovement : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 5f;

    void Update()
    {
        // เคลื่อนที่ใน Local Space แต่แสดงผลใน World Space
        transform.Translate(
                Vector3.forward * moveSpeed * Time.deltaTime
        );

        // หรือใช้ Translate แบบ Default (Local)
        transform.Translate(
                Vector3.up * moveSpeed * Time.deltaTime
        );
    }
}
```

Transform Rotation

(มุม)

Quaternion Rotation

```
public class RotationExample : MonoBehaviour
{
    void Update()
    {
        // อ่านค่า rotation ปัจจุบัน (Quaternion)
        Quaternion currentRotation = transform.rotation;

        // เขียนค่า rotation ใหม่
        transform.rotation = Quaternion.Euler(45f, 90f, 180f);

        // หรือใช้ Quaternion.LookRotation สำหรับทิศทาง
        Vector3 targetDirection = new Vector3(0f, 1f, 0f);
        transform.rotation = Quaternion.LookRotation(targetDirection);
    }
}
```

Euler Angles

```
public class EulerExample : MonoBehaviour
{
    void Update()
    {
        // อ่านค่าเป็น degrees
        float x = transform.eulerAngles.x;
        float y = transform.eulerAngles.y;
        float z = transform.eulerAngles.z;

        // เขียนค่าเป็น degrees
        transform.rotation = Quaternion.Euler(0f, 180f, 0f);
    }
}
```

การหมุนแบบ Smooth

```
public class SmoothRotation : MonoBehaviour
{
    [SerializeField] private float rotationSpeed = 90f;

    void Update()
    {
        // หมุนใน Local Space
        transform.Rotate(Vector3.up, rotationSpeed * Time.deltaTime);

        // หรือหมุนใน World Space
        transform.RotateAround(
                transform.position,
                Vector3.up,
                rotationSpeed * Time.deltaTime
        );
    }
}
```

การคำนวณการหมุน
(Rotation Math)

```
public class RotationMath : MonoBehaviour
{
    void CalculateRotation()
    {
        // Get target direction
        Transform target = GameObject.Find("Target").transform;

        // Calculate rotation to face target
        Quaternion targetRotation = Quaternion.LookRotation(
                target.position - transform.position
        );

        // Smoothly interpolate rotation
        transform.rotation = Quaternion.Slerp(
                transform.rotation, targetRotation, Time.deltaTime * 5f
        );
    }
}
```

Transform Scale

(ขนาด)

การอ่าน/เขียน Scale

```
public class ScaleExample : MonoBehaviour
{
    void Update()
    {
        // อ่านค่า scale ปัจจุบัน
        Vector3 currentScale = transform.localScale;

        // เขียนค่า scale ใหม่
        transform.localScale = new Vector3(2f, 2f, 2f);

        // หรือใช้ Scale แบบเฉพาะแกน
        transform.localScale = new Vector3(1.5f, 1f, 0.5f);
    }
}
```

การขยายแบบ Smooth

```
public class SmoothScale : MonoBehaviour
{
    [SerializeField] private float targetScale = 5f;
    [SerializeField] private float scaleSpeed = 2f;

    void Update()
    {
        // Smoothly interpolate scale
        transform.localScale = Vector3.Lerp(
                transform.localScale,
                new Vector3(targetScale, targetScale, targetScale),
                Time.deltaTime * scaleSpeed
        );
    }
}
```

การแปลงระหว่าง

Local Space

และ

World Space

Local to World Transformation

```
public class CoordinateConversion : MonoBehaviour
{
    // แปลงจาก Local Position เป็น World Position
    public Vector3 GetWorldPosition()
    {
        return transform.TransformPoint(transform.localPosition);
    }

    // แปลงจาก Local Direction เป็น World Direction
    public Vector3 GetWorldDirection()
    {
        return transform.TransformDirection(transform.localEulerAngles);
    }
}
```

World to Local Transformation

```
public class CoordinateConversion : MonoBehaviour
{
    // แปลงจาก World Position เป็น Local Position
    public Vector3 GetLocalPosition(Vector3 worldPos)
    {
        return transform.InverseTransformPoint(worldPos);
    }

    // แปลงจาก World Direction เป็น Local Direction
    public Vector3 GetLocalDirection(Vector3 worldDir)
    {
        return transform.InverseTransformDirection(worldDir);
    }
}
```

Child Hierarchy ผ่าน Parenting

การเพิ่ม/ลบ Children

```
public class HierarchicalSystem : MonoBehaviour
{
    // เพิ่ม GameObject เป็น child
    public void AttachChild(GameObject child)
    {
        if (child != null)
            transform.GetChild(transform.GetChildrenCount()).AddChild(child);
    }

    // ลบ GameObject จาก hierarchy
    public void DetachChild(GameObject child)
    {
        if (child != null)
        {
            Transform childTransform = child.GetComponent<Transform>();
            childTransform.SetParent(null);
        }
    }
}
```

การเข้าถึง Children

```
public class ChildSystem : MonoBehaviour
{
    // Get single child component
    public Camera GetMainCamera()
    {
        return GetComponentInChildren<Camera>();
    }

    // Get all children transforms
    public Transform[] GetAllChildren()
    {
        return GetComponentsInChildren<Transform>();
    }
}
```

ตัวอย่างการใช้งาน
Player Controller

```
using UnityEngine;
using UnityEngine.InputSystem;

[RequireComponent(typeof(Rigidbody))]
public class PlayerController : MonoBehaviour
{
    [Header("Movement")]
    [SerializeField] private float moveSpeed = 5f;
    [SerializeField] private float rotationSpeed = 180f;

    [Header("Physics")]
    [SerializeField] private Vector3 movementVelocity = Vector3.zero;

    private float horizontal;
    private float vertical;
```

```
void Update()
{
    HandleInput();
    HandleMovement();
    HandleRotation();
}

private void HandleMovement()
{
    // Move ใน Local Space
    Vector3 movement = transform.TransformVector(
                                movementInput.normalized
                            ) * moveSpeed * Time.deltaTime;
    transform.position += movement;
}
```

```
private void HandleInput()
{
    horizontal = 0f;
    vertical = 0f;
    if (Keyboard.current.aKey.isPressed) horizontal = -1f;
    if (Keyboard.current.dKey.isPressed) horizontal = 1f;
    if (Keyboard.current.wKey.isPressed) vertical = 1f;
    if (Keyboard.current.sKey.isPressed) vertical = -1f;

    Vector3 movementInput = new Vector3(horizontal, 0f, vertical);

    // คำนวณทิศทางจาก Camera
    Vector3 cameraDirection = Camera.main.transform.forward;
    cameraDirection.y = 0f;
    cameraDirection.Normalize();
    Vector3 flatCameraDirection = transform.right * -horizontal
                                + cameraDirection.vertical * vertical;
    movementInput.Normalize();
}
```

```
    private void HandleRotation()
    {
        // Rotate to face camera
        float targetYRotation = Mathf.Atan2( horizontal, vertical) *
                                Mathf.Rad2Deg +
                                Camera.main.transform.eulerAngles.y;

        Quaternion targetRotation = Quaternion.Euler(
                                0f,
                                targetYRotation,
                                0f);
        transform.rotation = Quaternion.Slerp(
                                transform.rotation,
                                targetRotation,
                                Time.deltaTime * rotationSpeed);
    }
}
```

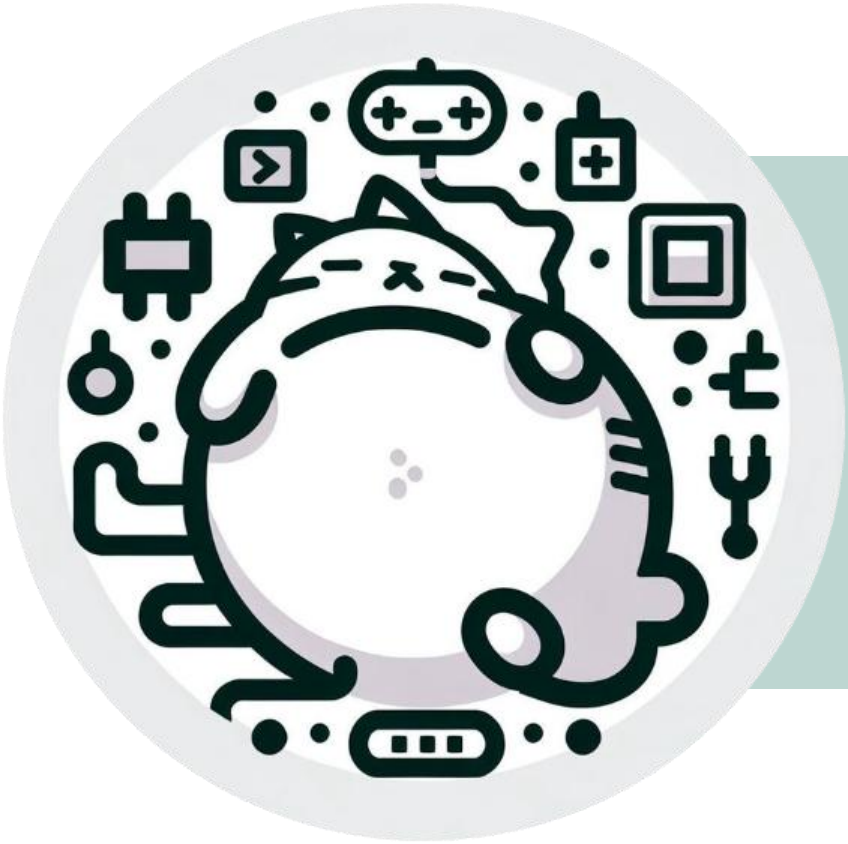
ข้อควรระวัง

Euler Angle
Gimbal Lock

```
// ไม่ดี - ใช้ Euler angles อาจเกิด Gimbal Lock
public class BadRotation : MonoBehaviour
{
    void Update() {
          // จะเสียการควบคุม
        transform.eulerAngles = new Vector3(0f, Input.GetAxis("Y") * 5f, 0f);
    }
}

// ดี - ใช้ Quaternion แทน
public class GoodRotation : MonoBehaviour
{
    void Update()
    {
        Quaternion targetRotation = Quaternion.Euler(0f, Input.GetAxis("Y") * 5f, 0f);
        transform.rotation = Quaternion.Slerp(
                                    transform.rotation,
                                    targetRotation,
                                    Time.deltaTime * 5f);
    }
}
```

Performance Optimization

```
// ไม่ดี - เรียก Update หลายครั้งต่อ frame
public class BadUpdate : MonoBehaviour
{
    void Update() {
        private float horizontal=0f;
        private float vertical=0f;
        private float forward=0f;
        if (Keyboard.current.aKey.isPressed) horizontal = -10f;
        if (Keyboard.current.dKey.isPressed) horizontal = 10f;
        if (Keyboard.current.wKey.isPressed) vertical = 10f;
        if (Keyboard.current.sKey.isPressed) vertical = -10f;
        if (Keyboard.current.qKey.isPressed) forward = 10f;
        if (Keyboard.current.eKey.isPressed) forward = -10f;
        transform.position = new Vector3(horizontal, vertical, forwardf); // คำนวณทุก frame
    }
}
// ดี - Cache Transform reference
public class GoodUpdate : MonoBehaviour
{
    private Vector3 targetPosition;
    void Update() {
        if (targetPosition != transform.position) {
            transform.position = targetPosition;
        }
    }
}
```



Physics

CollisionDetection, Rigidbody

Unity ใช้ PhysX เป็นฟิสิกส์ engine ที่รองรับ 3D physics simulation โดยใช้ Collision Detection และ Rigidbody Components

Physics Engine

Component	หน้าที่	ต้องใช้กับอะไร?
Collider	กำหนดรูปทรง collision volume	ใช้กับทุก GameObject ที่ต้องการ detect collision
Rigidbody	ทำให้ GameObject เป็น physical body และรองรับ physics simulation	ใช้เฉพาะเมื่อต้องการ physics behavior

Collider vs Rigidbody

```
// Collider สำหรับ static objects (ไม่มี movement)
public class StaticWall : MonoBehaviour
{
    void Awake()
    {
        // เพิ่ม Collider component
        Collider wallCollider = GetComponent<Collider>();
        if (wallCollider == null)
        {
            wallCollider = gameObject.AddComponent<BoxCollider>();
        }
    }
}
// Rigidbody สำหรับ dynamic objects
public class MovingObject : MonoBehaviour
{
    void Awake()
    {
        // เพิ่ม Rigidbody component
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb == null)
        {
            rb = gameObject.AddComponent<Rigidbody>();
        }
    }
}
```

Collider Type	ใช้งานกับ	ความแม่นยำ	Performance
BoxCollider	Cubic objects	Low-Medium	High
SphereCollider	Round objects	Low	Highest
CapsuleCollider	Human/Character models	Medium	Medium
MeshCollider	Complex geometry	High	Lowest

Collider Types

```
public class ColliderExample : MonoBehaviour
{
    void Awake()
    {
        // Add BoxCollider (default for most cases)
        BoxCollider box = gameObject.AddComponent<BoxCollider>();

        // Set size ของ collider
        box.size = new Vector3(2f, 4f, 1f);

        // Center offset
        box.center = new Vector3(0f, 0.5f, 0f);
    }

    void OnCollisionEnter(Collision collision)
    {
        Debug.Log("Collision with: " + collision.gameObject.name);
    }
}
```

Property	คำอธิบาย	ค่าเริ่มต้น
Mass	มวลของวัตถุ (kg)	1 kg
Drag	Air resistance	0f
Angular Drag	Rotational resistance	0f
Constraints	Limit movement/rotation	None

Rigidbody Properties

```
public class RigidbodyExample : MonoBehaviour
{
    void Awake()
    {
        Rigidbody rb = GetComponent<Rigidbody>();

        if (rb == null)
        {
            rb = gameObject.AddComponent<Rigidbody>();
        }
        rb.mass = 10f; // ตั้งค่า mass
        rb.drag = 0.5f; // ตั้งค่า drag
        rb.angularDrag = 0.2f;
    }

    void Update()
    {   // Access Rigidbody reference
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb != null)
        {
            Debug.Log("Velocity: " + rb.velocity);
            Debug.Log("Angular Velocity: " + rb.angularVelocity);
        }
    }
}
```

Physics Mode

```
using UnityEngine;
using UnityEngine.Physics;

public class RigidbodyModeExample : MonoBehaviour
{
    void SetupRigidbody()
    {
        Rigidbody rb = gameObject.AddComponent<Rigidbody>();

        // Dynamic (default) - รองรับ physics simulation
        rb.isKinematic = false;

        // Static - ไม่ตอบสนองต่อ gravity, ใช้สำหรับ static objects
        rb.isKinematic = true;
        rb.mass = float.PositiveInfinity;

        // Character Controller - สำหรับ character movement
        rb.isKinematic = true;
        rb.collisionDetectionMode = CollisionDetectionMode.Disable;
    }
}
```

Collision Event Methods

ใช้ตรวจสอบเมื่อมีวัตถุ 2 ชิ้นที่มี Collider (และมีอย่างน้อยหนึ่งชิ้นที่มี Rigidbody) วิ่งมาชนกัน

```
using UnityEngine;
public class PhysicsTest : MonoBehaviour
{
    private void OnCollisionEnter(Collision collision) // ทำงานเมื่อเริ่มมีการชนกัน
    {
        Debug.Log($"[Physics Success] ชนกับวัตถุชื่อ: {collision.gameObject.name}");
    }
    private void OnCollisionStay(Collision collision) // ทำงานค้างไว้ตราบใดที่ยังชนกันอยู่
    {
        // Debug.Log("กำลังชนกันอยู่...");
    }
    private void OnCollisionExit(Collision collision) // ทำงานเมื่อวัตถุหลุดออกจากกัน
    {
        Debug.Log("วัตถุหลุดออกจากกันแล้ว");
    }
}
```

ทดสอบการชน (Collision Test)

ใช้ในกรณีที่วัตถุชิ้นหนึ่งติ๊กถูกที่ช่อง Is Trigger ในคอมโพเนนต์ Collider (วัตถุจะทะลุผ่านกันได้ แต่ระบบยังรับรู้การตัดกันอยู่)

```
using UnityEngine;
public class TriggerTest : MonoBehaviour
{
    // ทำงานเมื่อเริ่มเดินทะลุเข้าโซน
    private void OnTriggerEnter(Collider other)
    {
        Debug.Log($"[Trigger Success] วัตถุ {other.gameObject.name} เดินเข้ามาในโซนตรวจจับ");
    }
}
```

ทดสอบการทะลุผ่าน/โซนตรวจจับ (Trigger Test)

ใช้ในกรณีที่วัตถุชิ้นหนึ่งติ๊กถูกที่ช่อง Is Trigger ในคอมโพเนนต์ Collider (วัตถุจะทะลุผ่านกันได้ แต่ระบบยังรับรู้การตัดกันอยู่)

```
using UnityEngine;
public class TriggerTest : MonoBehaviour
{
    // ทำงานเมื่อเริ่มเดินทะลุเข้าโซน
    private void OnTriggerEnter(Collider other)
    {
        Debug.Log($"[Trigger Success] วัตถุ {other.gameObject.name} เดินเข้ามาในโซนตรวจจับ");
    }
}
```

ทดสอบการทะลุผ่าน/โซนตรวจจับ (Trigger Test)

OnTriggerEnter Events (Trigger-based)

```
public class TriggerExample : MonoBehaviour
{
    void Start()
    {   // ต้องตั้งค่า Collider เป็น Is Trigger ก่อน
        Collider collider = GetComponent<Collider>();
        if (collider != null)
        {
            collider.isTrigger = true;
        }
    }

    void OnTriggerEnter(Collider other)
    {
        Debug.Log("Trigger entered: " + other.gameObject.name);
    }

    void OnTriggerStay(Collider other)
    {
        Debug.Log("Trigger staying");
    }

    void OnTriggerExit(Collider other)
    {
        Debug.Log("Trigger exited: " + other.gameObject.name);
    }
}
```

Collision Detection Methods

```
public class CollisionDetection : MonoBehaviour
{
    [SerializeField] private LayerMask detectLayers;

    void Start()
    {
        // Continuous dynamic - ตรวจสอบทุก frame (precise)
        GetComponent<Rigidbody>().collisionDetectionMode =
            CollisionDetectionMode.ContinuousDynamic;

        // Continuous specicfic - สำหรับการเคลื่อนที่เร็ว
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb != null)
        {
            rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpecicifc;
        }
    }
}
```

Collision Detection Methods

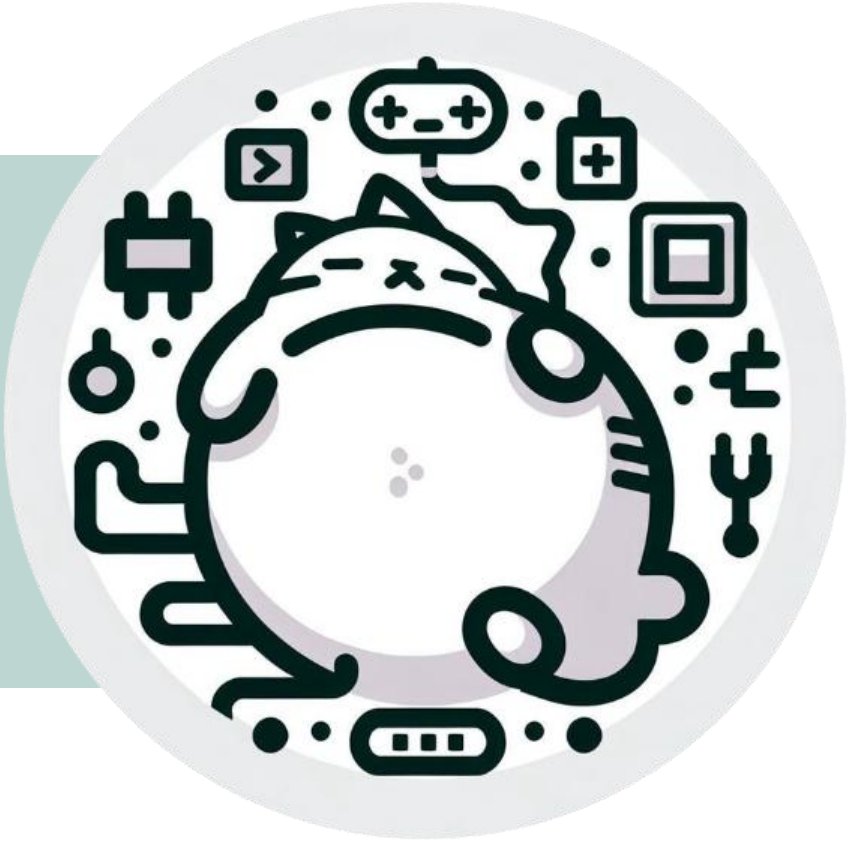
```
public class CollisionDetection : MonoBehaviour
{
    [SerializeField] private LayerMask detectLayers;

    void Start()
    {
        // Continuous dynamic - ตรวจสอบทุก frame (precise)
        GetComponent<Rigidbody>().collisionDetectionMode =
            CollisionDetectionMode.ContinuousDynamic;

        // Continuous specicfic - สำหรับการเคลื่อนที่เร็ว
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb != null)
        {
            rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpecicifc;
        }
    }
}
```

Raycast System

Raycast Basics

```
public class RaycastExample : MonoBehaviour
{
    [SerializeField] private Vector3 rayOrigin = Vector3.zero;
    [SerializeField] private Vector3 rayDirection = Vector3.forward;
    [SerializeField] private float rayDistance = 10f;

    void StartRaycast()
    {    // Raycast จาก transform origin
        Ray ray = new Ray(transform.position, transform.forward);
        // Detect hit
        if (Physics.Raycast(ray, out RaycastHit hit, rayDistance))
        {
            Debug.Log("Hit: " + hit.transform.gameObject.name);
            Debug.Log("Distance: " + hit.distance);
            Debug.Log("Normal: " + hit.normal);
            Debug.Log("Point: " + hit.point);
            // Access collider
            if (hit.collider != null)
            {
                Debug.Log("Collider type: " + hit.collider.GetType().Name);
            }
        }
    }
}
```

Scenes
&
Scene Management



ในระบบนิเวศของการพัฒนาเกมนั้น Scene และ Scene Management คือโครงสร้างพื้นฐานที่ใช้จัดการลำดับการดำเนินเรื่อง (Flow), การบริหารจัดการหน่วยความจำ (Memory Management) และการแบ่งส่วนเนื้อหาของเกมออกเป็นส่วนย่อย ๆ

Scene และ Scene Management

Scene คือหน่วยพื้นฐานในการจัดเก็บและแสดงผลข้อมูลใน Unity โดยหนึ่ง Scene จะประกอบด้วยวัตถุ (GameObjects), ส่วนประกอบ (Components), แสง (Lights), และสภาพแวดล้อมทางกายภาพที่จำเป็นสำหรับการประมวลผลในขณะนั้น

Scene (ฉากหรือพื้นที่ทำงาน)

Scene ทำหน้าที่เป็น **"ภาชนะ"** (Container) ที่รวมรวบทุกอย่างที่ต้องแสดงผลให้ผู้เล่นเห็น ในหนึ่งสถานะของเกม เช่น ฉากเมนูหลัก (Main Menu), พื้นที่การเล่น (Level/Stage), หรือห้องโถงขนาดเล็ก

Scene **ไม่ใช่แค่ "สถานที่"** แต่**เป็น Logical Partition ของข้อมูล** ดังนั้น การแบ่งเนื้อหาออกเป็นหลาย Scene ช่วยให้ระบบจัดการหน่วยความจำได้มีประสิทธิภาพ เนื่องจาก Unity จะทำการโหลดและคืนทรัพยากร (Unload) เฉพาะสิ่งที่จำเป็นในขณะนั้นเท่านั้น

Scene Management คือกระบวนการควบคุมการเปลี่ยนผ่านระหว่างสถานะต่างๆ ของเกม โดยใช้คลาสหลักคือ UnityEngine.SceneManagement เพื่อสั่งการให้ระบบทำการโหลดหรือสลับไปยัง Scene ที่กำหนด

Scene Management (การบริหารจัดการฉาก)

รูปแบบการโหลดฉากพื้นฐาน

Single Loading (การโหลดฉากเดี่ยว) เป็นวิธีการมาตรฐานเมื่อผู้เล่นเปลี่ยนจากหน้าเมนูเข้าสู่เนื้อหาเกม ระบบจะทำการทำลาย (Destroy) ทุกอย่างใน Scene ปัจจุบันและโหลดข้อมูลใหม่เข้าไปแทนที่ทั้งหมด

Single Loading (การโหลดฉากเดี่ยว)

Additive Loading (การโหลดแบบเพิ่มเสริม) เป็นเทคนิคขั้นสูงในการเปิดใช้งานหลาย Scene พร้อมกันในเวลาเดียวกัน เช่น การโหลดฉาก **"ระบบอินเตอร์เฟซ"** หรือ **"พื้นที่ส่วนขยาย"** ซ้อนทับลงบนฉากหลัก วิธีนี้ช่วยให้สามารถรักษาสภาพแวดล้อมเดิมไว้ได้ในขณะที่เพิ่มเนื้อหาใหม่เข้าไป

Additive Loading (การโหลดแบบเพิ่มเสริม)

สำหรับการพัฒนาเกมขนาดใหญ่ การใช้คำสั่ง **SceneManager.LoadScene** แบบปกติอาจทำให้เกิดอาการ **"กระตุก"** หรือหยุดชะงัก (Frame Drop) เนื่องจากการประมวลผลที่หนักเกินไปในเฟรมเดียว Unity จึงนำเสนอระบบ Asynchronous Loading ผ่าน **SceneManager.LoadSceneAsync()**

Asynchronous Loading (การโหลดข้อมูลแบบไม่ประสานเวลา)

ระบบจะทยอยโหลดข้อมูลของฉากใหม่เข้าไปในหน่วยความจำแบบค่อยเป็นค่อยไป (Background Loading) โดยไม่ขัดจังหวะการประมวลผลเฟรมปัจจุบัน

ช่วยให้ผู้พัฒนาสามารถสร้าง **"หน้าจอโหลด"** (Loading Screen) ที่มีความลื่นไหล และแสดงแถบเปอร์เซ็นต์การดาวน์โหลดที่ถูกต้องได้

ในการควบคุมการสลับฉากผ่านระบบ Scene Management นักพัฒนาต้องเรียกใช้ Namespace

UnityEngine.SceneManagement

เพื่อเข้าถึงเครื่องมือจัดการหลัก

```
using UnityEngine;
using UnityEngine.SceneManagement; // จำเป็นสำหรับการใช้งาน SceneManager

public class SceneController : MonoBehaviour
{
    // ตัวอย่างการเปลี่ยนฉากแบบปกติ (Synchronous)
    public void LoadNextLevel()
    {
        // คำสั่งจะหยุดการประมวลผลในปัจจุบันและโหลดฉากชื่อ "GameScene" ทันที
        SceneManager.LoadScene("GameScene");
    }
    // ตัวอย่างการเปลี่ยนฉากแบบไม่ประสานเวลา (Asynchronous) เพื่อความลื่นไหลของเฟรมเรต
    public async void LoadLevelAsync()
    {
        // เริ่มต้นกระบวนการโหลดฉากในเบื้องหลัง
        AsyncOperation operation = SceneManager.LoadSceneAsync("GameScene");
        while (!operation.isDone)
        {
            // รอเวลาเล็กน้อยก่อนตรวจสอบสถานะถัดไป (เช่น เพื่ออัปเดตความคืบหน้าบน UI)
            float progress = operation.progress;
            Debug.Log($"Progress: {progress}");
            await System.Threading.Tasks.Task.Delay(10); // รอ 10 มิลลิวินาทีในแต่ละรอบการตรวจสอบ
        }
    }
    // การเรียกใช้คำสั่งเพื่อกลับไปยังเมนูหลัก (ตัวอย่างการจัดการสถานะ)
    public void GoToMainMenu()
    {
        SceneManager.LoadScene("MainMenuScene");
    }
}
```

Scene

คือหน่วยจัดเก็บข้อมูลและบริบทของเนื้อหา (Data & Context Container)

SceneManagement

คือระบบควบคุมลำดับการทำงานและการจัดการทรัพยากรผ่านการสลับหรือเพิ่มฉาก

Asynchronous Operations

เป็นเทคนิคสำคัญในการบริหารจัดการประสบการณ์ผู้ใช้ (User Experience) และประสิทธิภาพเชิงวิศวกรรม (System Performance) ในเกมขนาดใหญ่

Q & A

How to handle audience questions effectively