Contents

Introduction

If you're trying to get into game development as a Software Engineer, finding learning materials with the right level of context can be challenging. You'll likely face a choice between following materials introducing you to rudimentary C# and OOP concepts while also describing Unity concepts or starting with advanced tutorials and be left to figure out the core concepts deductively.

I started my programming journey around 2003 by picking up Game Maker, a 2D game engine that is still popular today (its incarnation today, *GameMaker: Studio*, uses a more stylized spelling). Countless hours spent coding little games and tools led me to a bigger passion for programming. Eventually, I was at a point where I focused mainly on Software Engineering. From my peers, I know this is quite a common path that many of us took to find programming.

Yet, the game development scene has changed significantly from those days. When I went to pick up Unity after a long absence from game development, I was primarily interested in understanding the basic concepts: what are the fundamental building blocks of a game? What do I need to know about how these building blocks are represented in memory or on disk? How is idiomatic code organized? What patterns are preferred?

Chapter 1 Unity's Basic Concepts

To start with a solid foundation, it is essential to recognize the basic building blocks of Unity. The Unity Editor organizes every game into Scenes, Game Objects, Components, Assets, and Prefabs. Let's dig in!

Scenes

A **Scene** is the largest unit of organizing your objects in-memory. Scenes contain the objects making up your game.

In basic use, one scene represents a **single level** in your game, where one scene is loaded at any given point. In more "advanced" use, you can have two or more active scenes at a time. Scenes can be loaded additively and unloaded[1]. Having multiple scenes loaded during gameplay comes especially handy when building a massive world; keeping far-away areas on-disk rather than in-memory will help you stay within your performance budget.

Every *game object* in Unity needs to be *in* a scene.

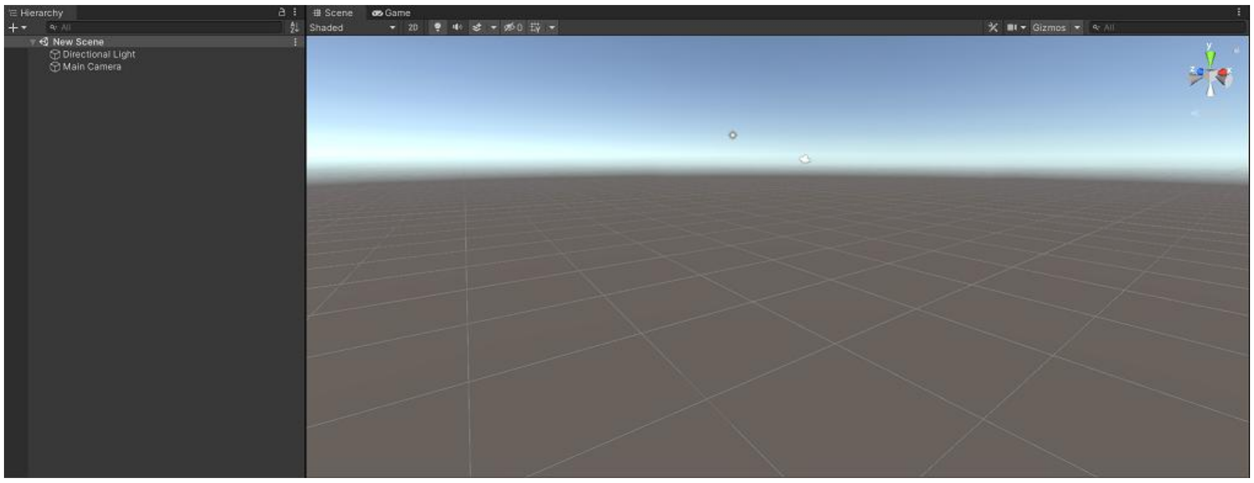


Figure 1 Unity Scene editor opening a default empty scene in 3D mode. Empty Scenes in Unity3D will, by default, include Main Camera *and* Directional light *objects.*

[1] Still, there can only ever be one "main" active scene at a time.

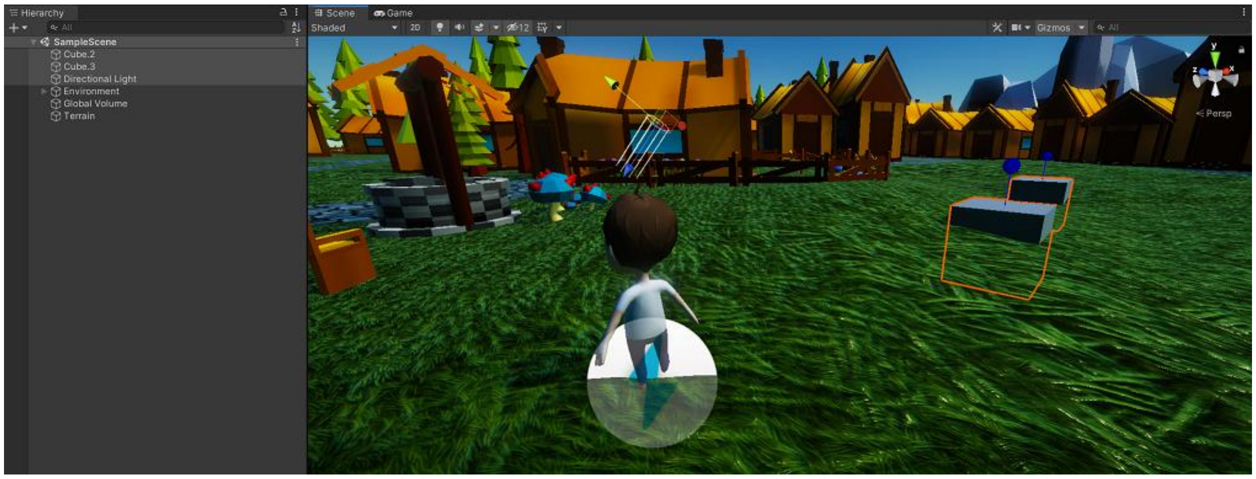


Figure 2 Unity Scene editor showing an example scene, with a few objects selected. You can use the scene view to edit levels in your game.

Game Objects

A **Game Object** (in code, `GameObject`) is one of the basic building blocks of a game.

Game Objects can represent both *physical* things you see in the game (e.g., a player, the ground, a tree, a terrain, lights, a weapon, a bullet, an explosion) *as well as metaphysical* things (e.g., an inventory manager, a multiplayer controller, etc.) in your game.

Every Game Object has a position and rotation. For metaphysical objects, this doesn't matter.

Game Objects can nest under each other. Each object's position and rotation are relative to its parent object. An object directly in the scene is relative to "world space" coordinates.

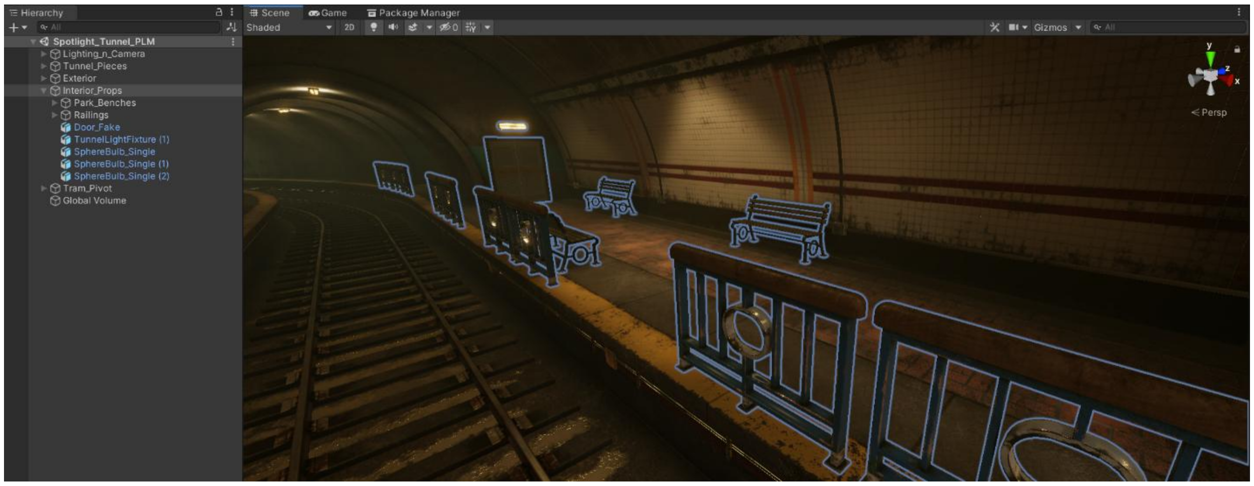


Figure 3 A group of objects nested together in a scene under an empty "Interior_Props" object for organizational purposes.

You might choose to nest your objects for many reasons. For example, you might decide it organizationally makes sense to put all your “environment” (e.g., individual pieces that make up a city or village) objects under an empty parent object. This way, it can be collapsed in the scene view and easily moved together when building your game.

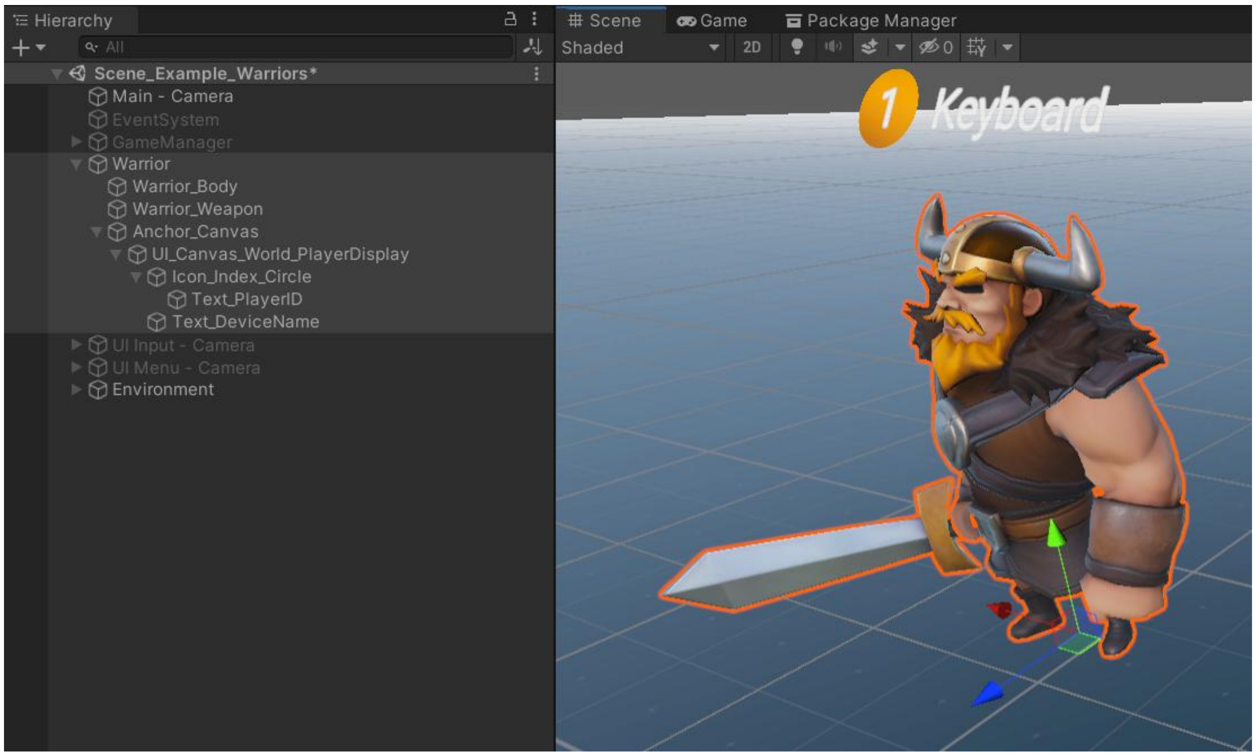
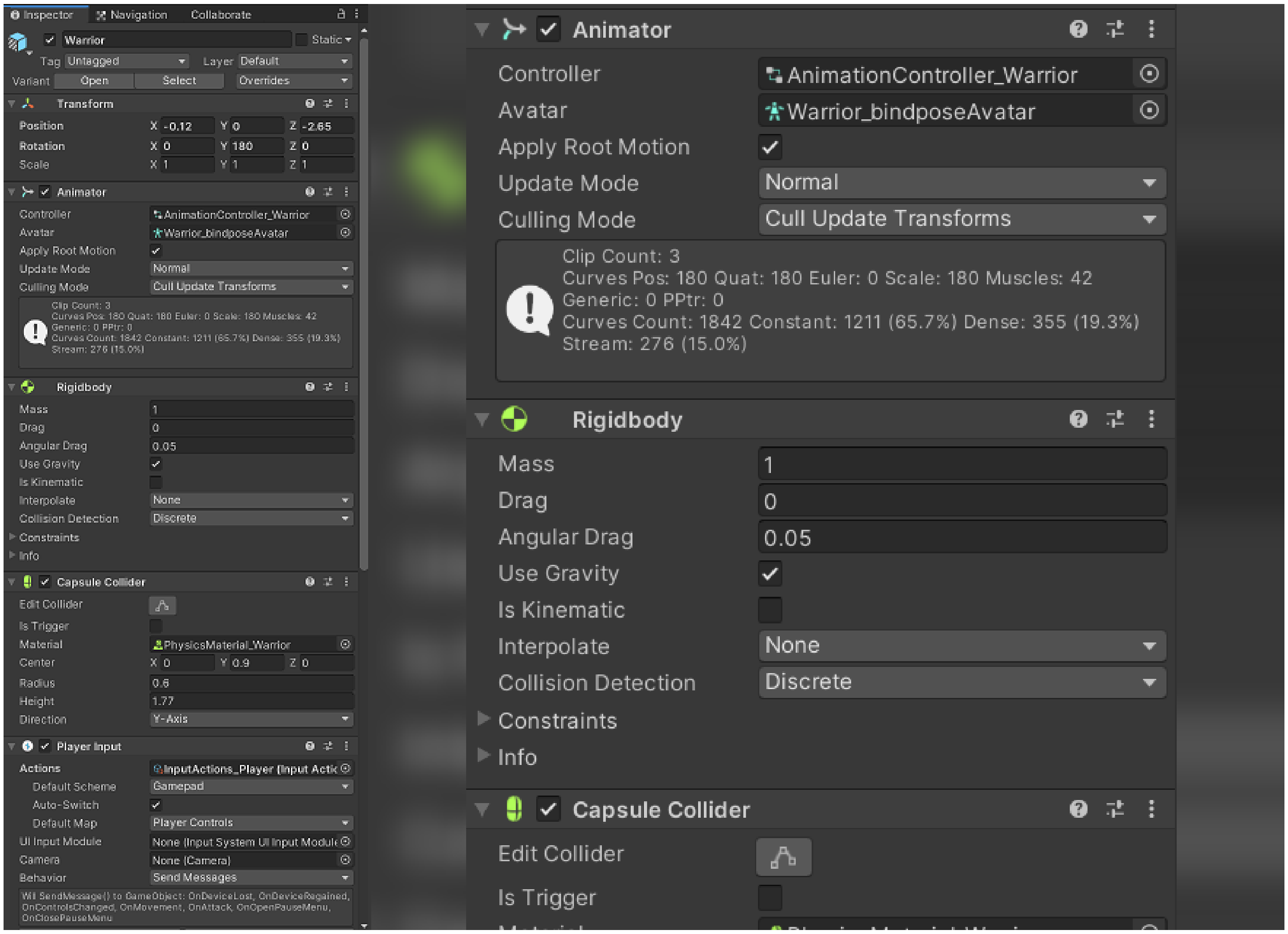


Figure 4 A group of objects nested under the player. These include the player’s weapon, avatar, and various UI elements rendered around the player.

Game Object nesting can also have *functional significance*. For example, a “Car” can be an object with code that controls the car’s speed and rotation as a whole. But individual child objects might represent the four wheels (these would spin independently), the car body, windows, etc. Moving the parent car object would move all the child objects, keeping their relative orientation to the parent (and each other). We might want the player to interact with a door separately from the rest of the car, for instance.

Components (& MonoBehaviours)



```
Inspector   Navigation   Collaborate
✓ Warrior                                   Static
Tag Untagged            Layer Default
Variant  Open  Select   Overrides

Transform
Position          X -0.12   Y 0     Z -2.65
Rotation          X 0       Y 180   Z 0
Scale             X 1       Y 1     Z 1

✓ Animator
Controller          AnimationController_Warrior
Avatar              Warrior_bindposeAvatar
Apply Root Motion   ✓
Update Mode         Normal
Culling Mode        Cull Update Transforms
  Clip Count: 3
  Curves Pos: 180 Quat: 180 Euler: 0 Scale: 180 Muscles: 42
  Generic: 0 PPtr: 0
  Curves Count: 1842 Constant: 1211 (65.7%) Dense: 355 (19.3%)
  Stream: 276 (15.0%)

Rigidbody
Mass                  1
Drag                  0
Angular Drag          0.05
Use Gravity           ✓
Is Kinematic
Interpolate           None
Collision Detection   Discrete
▶ Constraints
▶ Info

✓ Capsule Collider
Edit Collider
Is Trigger
Material      PhysicsMaterial_Warrior
Center        X 0    Y 0.9    Z 0
Radius        0.6
Height        1.77
Direction     Y-Axis

✓ Player Input
Actions            InputActions_Player (Input Actic
  Default Scheme   Gamepad
  Auto-Switch      ✓
  Default Map      Player Controls
UI Input Module    None (Input System UI Input Modul
Camera             None (Camera)
Behavior           Send Messages
  Will SendMessage() to GameObject: OnDeviceLost, OnDeviceRegained,
  OnControlsChanged, OnMovement, OnAttack, OnOpenPauseMenu,
  OnClosePauseMenu
```

Figure 5 The Warrior object from the previous screenshot is shown above in Unity's "Inspector" window. Each illustrated section (e.g., Animator, Rigidbody, Collider) are Components *making up this object.*

Every Game Object is comprised of **Components**.

A Component implements a well-defined set of behaviors for a GameObject to execute. Everything that makes an object what it is would come from the components that make it up:

- A single "visible" piece of a car will have a *Renderer* component that paints it, and
- It's also likely to have a *Collider* component that sets up its collision bounds.
- If a car represents the player, the top-level car object might have a *Player Input Controller* that takes key input events and translates these to code moving the car around.

While you can write large and complex Components that correspond 1:1 to an object (e.g., a player component codes the entire player, while an enemy component codes the enemy as a whole), it's typically common to factor out your logic into streamlined pieces corresponding to individual *traits*. For example:

Note
In code, a `MonoBehaviour` is the ubiquitous parent class representing Components. Most non-built-in components in the wild inherit from `MonoBehaviour`, which itself inherits from `Behavior` and `Component`, respectively.

- Whether a Player or an Enemy, all objects with health might have a `LivingObject` component that sets an initial health value, takes damage, and triggers a death event once it dies.
- A player might additionally have an input component controlling its movement, while the enemy might have an AI component that controls its movement instead.

Components receive various callbacks throughout their lifetime, known in Unity as *Messages*. Examples of Messages include `OnEnable`/`OnDisable`, `Start`, `OnDestroy`, `Update`, and others. If an object implements an `Update()` method, this method will automagically be called by Unity in every frame of the game loop while the object is active and the given component is enabled. These methods can be marked `private`; the Unity engine will still call them.

Components can also expose public methods as you'd expect. Other components can take a reference to a component and call these public methods.

Assets

Assets are the on-disk resources that make up your game project. These include meshes (models), textures, sprites, sounds, and other resources.

When serialized to the disk, your scenes are represented as assets made up of the Game Objects inside of them. The following section will discuss how you can make Game Objects you often reuse into an asset known as a Prefab (p. 13).[2]

[2] The term asset is a bit overloaded. Scenes and Prefabs are *serialized* and *organized* in your projects like other assets. You can browse these in the Asset view alongside code and other assets. If you're deep in Editor code manipulating an asset, Prefabs and Scenes cannot usually be treated as assets.

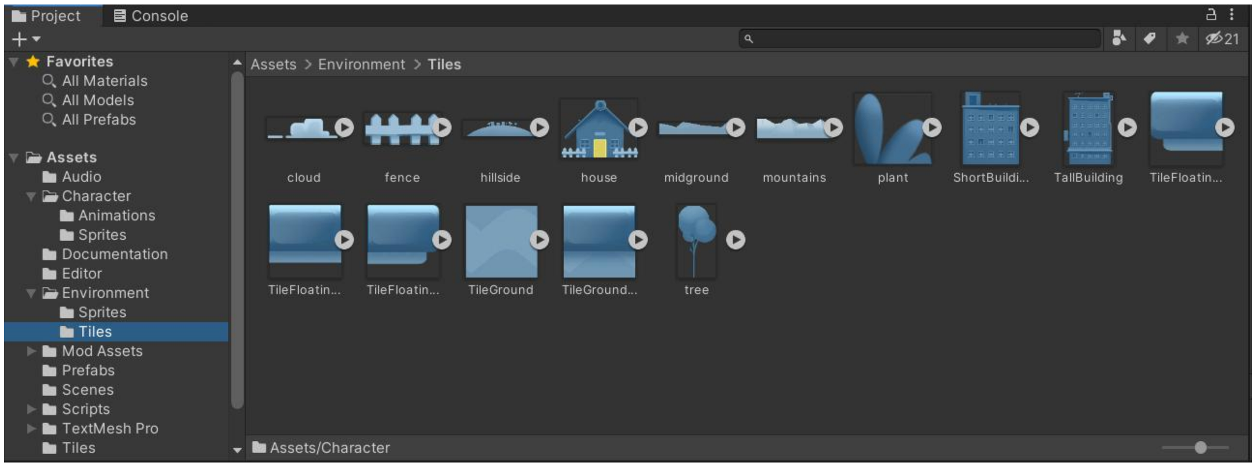


Figure 6 Unity Asset view showing visual assets in a game project.

Assets can also represent less tangible things, such as Input Control Maps, Graphics Settings, i18n string databases, and more. You can also create your own custom asset types using ScriptableObjects (discussed in Chapter 6, p. 45).

For your in-development project, assets form the key representation of your project's codebase alongside your code.

Your built-and-bundled game will include *most*[3] of your assets. These assets will be saved on disk on the device where the game is installed.

Prefabs

Game Objects, their Components, and their input parameters exist as individual *instances* in a scene. But what if a particular class of objects is commonly repeated? Such objects can be made into a **Prefab**, which is effectively the object in asset form.

Instances of a prefab in a scene can have local modifications that distinguish it (e.g., if a *tree* object is a prefab, you can have tree instances of different heights). Instances of a prefab all inherit and override data from their prefab.

Nested Prefabs

Starting with Unity 2018.3, Prefabs can be nested just as you expect:

[3] You can further optimize your game by loading some assets over the network or employing other asset-management techniques.

1. A parent object with prefab child objects can be a prefab itself. In the parent prefab, the child prefab instance can have its own modifications. In the scene, the entire prefab hierarchy is instantiated, and scene-specific modifications can layer on top.
2. A prefab instance in a scene with its own local modifications can be saved as its own "Prefab Variant" asset. A variant is a prefab asset that inherits from another prefab, applying additional modifications on top.

These concepts compose; a prefab variant of a nested prefab, or a prefab variant of a prefab variant, for instance.

Serialization & Deserialization

Your project's assets, scenes, and objects are all persisted on-disk. When editing your game, these objects are loaded in memory and saved back to disk using Unity's serialization system. When playtesting your game, the objects and scenes in-memory are loaded through the same serialization system. This system also maps between assets in your compiled bundle and the loaded/unloaded scene objects in-memory.

The Unity Engine's Serialization/Deserialization flow loads on-disk assets into memory (in your project: for editing/playtesting; in-game, when loading a scene) and is responsible for saving the state of your edited objects and components back into their scenes and prefabs.

Therefore, the serialization system is also at the core of the Unity Editor experience itself. For a `MonoBehaviour` to take some input on construction when instantiated in a scene, those fields must be *serialized*.

Most core Unity types such as `GameObject`s, `MonoBehaviour`s, and asset resources are Serializable and can receive initial values on creation from within the Unity Editor. Public fields on your `MonoBehaviour` are serialized by default (if they're of a serializable type), and private fields need to be marked with Unity's `[SerializeField]` attribute to be serialized as well.

Where Next?

These six concepts cover essential structural pieces for architecting games in Unity. Knowing more about these and how assets on-disk map to in-memory representation should give you the intuition needed to follow some of the more advanced tutorials.

There are still significant areas to wrap your head around in Unity. Understanding the Editor and mapping your Software Engineering best practices to game development best practices (Chapter 2) will help hone your skill. Even more, understanding broad areas such as lighting (Chapter 6), animation controllers (Chapter 12), navigation meshes (Chapter 13), input handling (Chapter 3) will help you go a long way as well. Later on, we’ll cover some game architecture best practices (Chapter 7 & Chapter 8), go over performance considerations (Chapter 17), and also dive deep into Unity’s ultra-high performance DOTS stack, including Unity ECS and the Jobs System in Chapter 18.

By understanding Unity's basic concepts, I hope you are armed with the knowledge needed to have a more *intuitive* understanding of the Engine and its workflows as you learn.

Chapter 8 Architecting your game with Dependency Injection

If you've read this far and feel that my advice to take advantage of the Editor and embrace Scriptable Objects resonates, then hopefully something feels magical when you add a `[SerializeField] private PlayerConfig _config;`, drag an asset into the slot, and suddenly your code has data.

In software engineering terms, the Inspector is a **Dependency Injection (DI) framework**. It is a GUI-based configuration file that resolves dependencies at scene load.

Many advanced Unity developers take this further, using **ScriptableObjects** to replace Singletons entirely. This is a valid, powerful pattern—but as we scale, we need to ensure it doesn't prevent us from writing testable code.

The Unity-Native Approach: ScriptableObjects as "Injected" Globals

The superior "Unity-Native" alternative to *Singleton Hell* is the **ScriptableObject Architecture** we discussed in Chapter 7. Instead of a `PlayerHealth` Singleton, you create a `FloatVariable` ScriptableObject.

- **The Player** writes to this asset.
- **The UI** reads from this asset.
- **The Save System** serializes this asset.

Neither the Player nor the UI knows about the other. They are decoupled, connected only by a shared "bucket" of data that lives in the project files, not the scene.

This has some advantages and disadvantages:

1. **The Good:** It effectively implements the **Liskov Substitution Principle**; you can swap a `PlayerHP` asset with a `TestHP` asset for debugging without changing a line of code. It is "Dependency Injection" configured via drag-and-drop.
2. **The Bad:** Any new scene or new test case requires these scriptable object references to be wired up in the editor by hand, miswiring can be a problem (unless we strongly-type our Scriptable Objects, i.e. a `PlayerHealthVariable` that extends `FloatVariable`), and missed values are an issue when there are many of these objects to inject

3. **The Ugly:** If you put *logic* inside these ScriptableObjects, you create a dependency on the Unity Engine. You cannot `new PlayerHealth()` in a unit test because `ScriptableObject.CreateInstance` requires the Unity engine to be running.

We'll be discussing how you can inject Scriptable Objects for testing in Chapter 16 on p. 122.

Dependency Injection Frameworks as an Alternative

In the next section, I'll describe how a *hybrid architecture* that uses DI frameworks to inject these assets can be combined with Scriptable Object -driven development to give us a "best of both worlds" approach. To get there, though, let's briefly describe what a pure DI-driven approach with pure C# objects would look like.

You will see a few Unity DI frameworks mentioned in the world:

1. **Reflex** (github.com/gustavopsantos/Reflex) – Lightweight framework that is currently the *state of the art* in terms of performance and allocations
2. **VContainer** (vcontainer.hadashikick.jp/) – Fast framework 5-10x faster than the "gold standard", Zenject
3. **Zenject** (github.com/modesttree/Zenject) – The legacy gold standard for DI, still widely used, but no longer actively maintained

Feature	**Zenject**	**VContainer**	**Reflex**
Performance	Slow (Reflection heavy)	Fast (IL Emit / Expression Trees)	Fastest (~4x vs VContainer)
Allocations	High	Low	Lowest (~28% less vs VContainer)
Configuration	Installers (MonoBehaviour) + Contexts	`LifetimeScope` (MonoBehaviour)	`ProjectScope` / `SceneScope` (MonoBehaviour)
Entry Points	`IInitializable`, `ITickable`	`IStartable`, `ITickable` (Pure C# Loop)	`IInstaller`, Standard Awake/Start
Best For	Legacy projects	Architectural Rigor	Performance

You should consider either VContainer or Reflex as your DI framework of choice.

With a DI framework, you can register a specific ScriptableObject instance as a "Singleton" within the container. This means any class that asks for GameConfig will receive that specific asset, without you needing to drag it into the Inspector.

For this chapter, we will use VContainer as its configuration concepts are not too different from Reflex.

In a pure DI architecture, the IDE knows about our scopes; finding references can help us know where objects are provided. On the other hand, we lose the ability to interact with the wiring in the editor and non-technical designers will have trouble reasoning about or extending the structure.

	ScriptableObject-Driven Development	**VContainer**	**Reflex**
Logic Container	ScriptableObject Assets	Plain C# Objects	Plain C# Objects
Dependency Resolution	Visual (Inspector drag-and-drop)	Programmatic	Programmatic
Testing	Asset-based Mocking + `CreateInstance`	Plain C# or Automated Mocking	Plain C# or Automated Mocking
Designer Accessibility	High (visual workflow in Inspector)	Low (C# code)	Low (C# code)
IDE Navigability	Hard (assets are data files)	Easy	Easy
Performance Impact	Native (no container overhead)	Small for container declaration (no post-resolve GC)	Minimal
Lifecycle	Project-wide	Scoped	Scoped

We'll show examples of testing VContainer and Reflect in Chapter 16 on p. 123.

The Hybrid Architecture: Injecting SOs into Pure C#

To maintain engineering rigor (testability and separation of concerns) while keeping the workflow benefits of ScriptableObjects, you may prefer a Hybrid Approach. In this approach, most *injectables* can still be Scriptable Objects. Since our scopes are themselves MonoBehaviours living in prefabs and managed via the Editor as regular assets, those scopes can take in any `[SerializedField]` it wants to and can be hooked up by designers that way.

This way:

- Editor-based dependency resolution happens visually in one (or very few) asset defining the project scope (and optionally a few scene scopes)
- DI system handles propagating those *visually injected* fields into all the objects (Pure C# classes *or* Unity Engine objects) that need them

In this case, a designer:

- can inspect, rewire, and understand how global assets and Scriptable Object ‘variables’ are scoped
- can’t see or rewire how each injectable object makes it to each consumer (this is probably good)

Compared to DI, an engineer loses full IDE-based resolution. You can still tell that a variable comes from some scope, but will need to look into the editor to see what that scope is defined as. The cost of this is much lower than in pure Scriptable Object-driven design, because there is a very small number of scopes that are hierarchically arranged and easy to view and reason about.

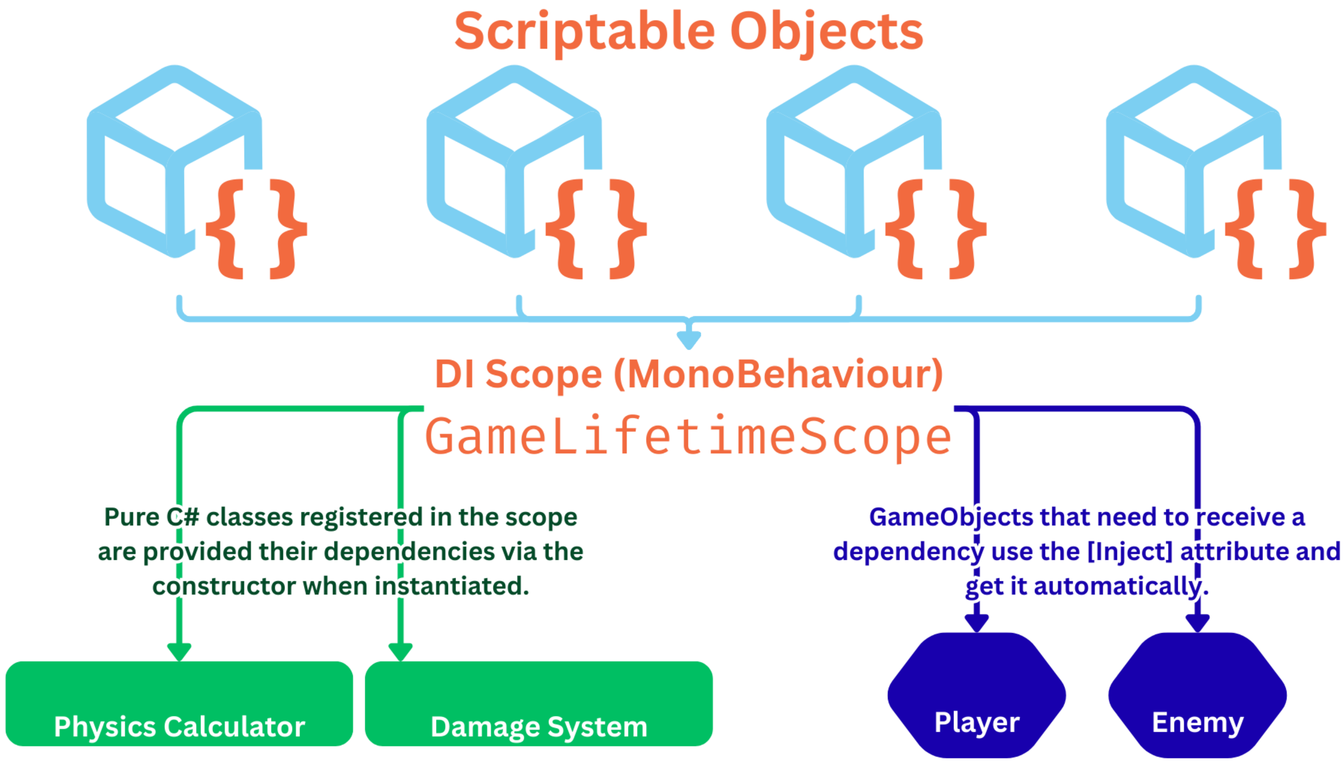

Scriptable Objects
DI Scope (MonoBehaviour)
GameLifetimeScope
Pure C# classes registered in the scope are provided their dependencies via the constructor when instantiated.
GameObjects that need to receive a dependency use the [Inject] attribute and get it automatically.
Physics Calculator
Damage System
Player
Enemy

Figure 25 Demonstrating the Hybrid Approach collecting scriptable objects into GameLifetimeScope then passing that on to classes that need it.

How it works

We treat ScriptableObjects as **Configuration** and **State Containers**, but we use **Pure C# Classes** for logic. Unity Engine objects may also receive injectables. We then use a DI Framework to wire them together.

We can use a DI container to treat our ScriptableObjects as "Singletons" that are automatically injected into our logic classes.

```
// 1. The Data (ScriptableObject)
[CreateAssetMenu]
public class GameConfig : ScriptableObject {
    public float GlobalGravity = 9.8f;
    public int MaxPlayers = 4;
}
```

```
// 2. The Logic (Pure C# - No Unity Dependency*)
// *Except for the Config object, which is just data.
public class PhysicsCalculator {
    private readonly GameConfig _config;

    // We INJECT the ScriptableObject here.
    // We can inject a real config, or a dummy config for testing.
    public PhysicsCalculator(GameConfig config) {
        _config = config;
    }

    public float GetFallSpeed(float time) {
        return _config.GlobalGravity * time; // Logic uses the data
    }
}
```

Classes like PhysicsCalculator can be used with any DI framework for Unity.

Step 1: Setting up the Scope

Create a `LifetimeScope` to manage your dependencies (this would be `ProjectScope` or `SceneScope` in Reflex). Since these "scope" classes are Unity Objects (see p. 40), they can have serializable fields on them and can be configured from within the editor. Designers

```
public class GameLifetimeScope : LifetimeScope
{
    // We expose the ScriptableObject slot here instead of on every
    // individual enemy/player.
    [SerializeField] private GameConfig _sharedConfig;
    [SerializeField] private PlayerStats _playerStats;
    protected override void Configure(IContainerBuilder builder)
    {
        // Register the ScriptableObjects as instances
        // Now, any class requesting 'GameConfig' gets this asset.
        builder.RegisterInstance(_sharedConfig);
        builder.RegisterInstance(_playerStats);

        // Register Pure C# Logic classes
        // VContainer sees they need 'GameConfig' in the constructor
        // and injects it automatically.
        builder.Register<PhysicsCalculator>(Lifetime.Singleton);
        builder.Register<DamageSystem>(Lifetime.Singleton);
    }
}
```

Step 2: The Consumer (Logic)

Now your logic is clean. It defines what it needs in the constructor.

```
public class DamageSystem
{
    private readonly PlayerStats _stats;
    // VContainer automatically passes the ScriptableObject here
    public DamageSystem(PlayerStats stats)
    {
        _stats = stats;
    }
    public void ApplyDamage(int amount)
    {
        _stats.CurrentHP -= amount; // Modifying the SO state
    }
}
```

Step 3: The Consumer (MonoBehaviour)

For MonoBehaviours (which can't have constructors), we use **Method Injection**.

```
public class PlayerHUD : MonoBehaviour
{
    private PlayerStats _stats;

    [Inject] // VContainer calls this automatically
    public void Construct(PlayerStats stats)
    {
        _stats = stats;
    }

    private void Update()
    {
        // We are reading from the ScriptableObject "Singleton"
        // without having to drag-and-drop it in the Inspector.
        DisplayHP(_stats.CurrentHP);
    }
}
```