Tổng hợp kiến thực một cách lộn xộn

Tổng hợp những mảnh kiến thức nhỏ về lập trình và Unity

Đặc trưng của OOP

Tính kế thừa

- **Định nghĩa:** Thừa hưởng các thuộc tính, hành vi từ một lớp khác.
- **Biểu hiện:** Class có thể kế thừa (extends) lớp cha và triển khai (implements) các interface.
- **Lợi ích:**
  - Giúp tái sử dụng code đã có nhưng vẫn duy trì một hệ thống phân cấp thống nhất.
  - Lớp cha đơn giản và tổng quát hơn lớp con. Lớp con cụ thể và đa dạng hơn lớp cha.
- **Khác:** C# không hỗ trợ đa kế thừa nhưng có hỗ trợ sử dụng Interface. Trong một số trường hợp có thể sử dụng interface để mô phỏng đa kế thừa.
- **Ví dụ:** Lớp cha là lớp động vật có các phương thức di chuyển, kêu, ăn… các lớp con như chó, mèo, chim, cá… kế thừa lớp cha động vật thì đều có các hành vi di chuyển, kêu, ăn…

Tính đóng gói

- **Định nghĩa:** Giữ tất cả các thông tin quan trọng (data, method) bên trong một lớp, chỉ cho phép truy cập nhưng thông tin cần thiết.
- **Biểu hiện:** Xác định mức độ truy cập thông qua các accsess modifiers public, internal, protected, private
- **Lợi ích:**
  - Tăng cường bảo mật, bảo vệ dữ liệu quan trọng khỏi truy cập trái phép.
  - Hạn chế truy cập thông tin không cần giúp dễ đọc, dễ hiểu, dễ bảo trì hơn.
  - Cho phép thay đổi cách hoạt động bên trong lớp mà không ảnh hưởng đến chương trình.
  - Tăng khả năng tái sử dụng của class trong các dự án khác nhau.
- **Ví dụ:** Lớp chó có thuộc tính trạng thái (đói, no) là private để thay đổi trạng thái này người dùng phải sử dụng method public cho ăn (feed) không thể truy cập trực tiếp vào thuộc tính trạng thái.

Tính trừu tượng

- **Định nghĩa:** chỉ tiết lộ những thông tin cần thiết của một đối tượng. Bỏ qua những chi tiết phức tạp bên trong.
- **Biểu hiện:** Thể hiện thông qua abstact class và interface. Thưởng chỉ đưa ra nhiệm vụ mà không nói cụ thể cách triển khai. Cách triển khai được thực hiện ở các class kế thừa nó.
- **Lợi ích:**

    - Đơn giản hoá, người dùng chỉ cần quan tâm đến các thuộc tính, phương thức được công khai không cần quan tâm đến cách thức hoạt động của nó.
    - Dễ dàng thay đổi cách thức hoạt động bên trong mà không ảnh hưởng đến phần còn lại của chương trình.
    - Tăng khả năng tái sử dụng
- **Ví dụ:**
    - Khi cho động vật ăn thì nó thực hiện việc ăn. Người nuôi không cần quan tâm đến bộ não con chó ra lệnh gì, con chó nhai như thế nào...
    - Lái một chiếc ô tô thì người lái chỉ cần quan tâm đến côn, ga, số, phanh, vô lăng không cần quan tâm đến cách động cơ đốt nhiên liệu như nào các bánh răng truyền động ra sao để chuyển hướng...

Tính đa hình

- **Định nghĩa:** Cho phép các đối tượng thuộc các lớp khác nhau có thể phản hồi một message theo nhiều cách khác nhau.
- **Biểu hiện:** Overiding method, overloading method
    - Overiding method (trong runtime): lớp con phản hồi khác lớp cha của nó.
    - Overloading method (compile time): Phương thức có cùng tên nhưng khác số lượng tham số và có thể khác dữ liệu trả về
- **Lợi ích:**
    - Giảm thiểu lặp code, giúp dễ bảo trì hơn.
    - Cho phép chương trình thích ứng với các trường hợp khác nhau mà không cần sửa code nhiều.
    - Giúp tạo ra các mô hình dễ đọc, dễ hiểu.
- **Ví dụ:** Các động vật kế thừa lớp động vật đều có hành động di chuyển, kêu nhưng chó, chim, cá thực hiện các hành động di chuyển, kêu hoàn toàn khác nhau.

Cấu trúc dữ liệu

Array, List và HashSet

Cả 3 đều là cấu trúc dữ liệu được sử dụng để lưu trữ một tập hợp các phần tử có cùng kiểu dữ liệu

Tiêu chí	Array	List<T>	HashSet<T>
Cấu trúc	Mảng cố định kích thước	Mảng động (có Capacity, tự tăng)	Bảng băm (bucket + hash)
Kích thước	Cố định (khởi tạo xong không đổi)	Linh hoạt (tự tăng Capacity)	Linh hoạt
Truy cập theo index	O(1)	O(1)	**Không hỗ trợ**
Duyệt tuần tự	O(n)	O(n)	O(n)
Thêm phần tử	Không (phải cấp phát lại)	Cuối danh sách: trung bình O(1); chèn giữa: O(n)	Trung bình O(1)
Tìm	O(n)	O(n)	**Trung bình O(1)**
Cho phép trùng lặp	Có	Có	**Không** (tập hợp duy nhất)
Duy trì thứ tự	**Thứ tự chỉ số**	**Giữ thứ tự chèn**	**Không đảm bảo thứ tự**
Khi nên dùng	Kích thước biết trước, tối đa hiệu năng/nhớ	Danh sách tổng quát, thêm/xóa không quá nhiều ở giữa	Kiểm tra tồn tại, phép toán tập hợp (union/intersect/diff)
Hạn chế	Cứng nhắc về kích thước	Thêm/xóa giữa tốn kém O(n)	Không truy cập theo index; overhead bộ nhớ lớn hơn

Hashtable và Dictionary

Đều là cấu trúc dữ liệu để lưu trữ dữ liệu dưới dạng cặp key/value.

Tiêu chí	Hashtable	Dictionary<TKey,TValue>
IsGeneric	**Không** (non-generic) → boxing với value types	**Có** (generic) → tránh boxing
Kiểu key/value	`object`/`object`	`TKey`/`TValue`
Null Key	**Không cho phép**	**Không cho phép**
Null Value	Cho phép (kiểu tham chiếu)	Cho phép (kiểu tham chiếu)
Hiệu năng trung bình	Thấp hơn do boxing + cast	**Tốt hơn**, đặc biệt với value types

An toàn kiểu (type-safety)	Thấp (dễ lỗi cast runtime) Trả về null	**Cao** (kiểm tra compile-time) Trả về exception
Thứ tự	Không đảm bảo	Không đảm bảo (thường là theo thứ tự chèn trên .NET hiện đại, **đừng phụ thuộc**)
Thread-safety	Không an toàn luồng. Có thể bọc `Hashtable.Synchronized` (overhead cao)	Không an toàn luồng. Dùng `ConcurrentDictionary<,>` cho đa luồng.
API	Cũ, chủ yếu vì tương thích legacy	**Khuyến nghị mặc định** trong hầu hết trường hợp
Khi nên dùng	Chỉ khi phải tương thích code cũ	**Lựa chọn mặc định** cho map/tra cứu key/value.

So sánh

Equals và '=='

Đặc điểm	Toán tử ==	Phương thức Equals()
Loại so sánh mặc định	**Kiểu tham chiếu (class):** mặc định so sánh **tham chiếu** (trừ khi kiểu đó *overload* `==` (Ví dụ: `string`, `record`). **Kiểu giá trị (struct):** chỉ **một số kiểu dựng sẵn** (int, double, bool, enum, char, …) có `==` sẵn và so sánh **giá trị**; với *struct tự định nghĩa*, **không có** `==` **mặc định** — muốn dùng phải tự overload.	**Kiểu tham chiếu (class):** mặc định là **tham chiếu** như `==` (trừ khi kiểu override `Equals`). `string.Equals` mặc định so sánh **theo nội dung** (giá trị). **Kiểu giá trị (struct):** mặc định so sánh **giá trị** (field-by-field) thông qua `ValueType.Equals`, trừ khi bạn override/IEquatable.
Ghi đè / tuỳ biến	**Có thể** *overload* (không phải *override*) bằng cách khai báo `public static bool operator ==(...)`. Khi overload `==` **phải overload luôn** `!=`, và thường **nên** override `Equals` + `GetHashCode` cho nhất quán.	**Có thể** *override* (virtual method `System.Object`). Với generics/hiệu năng, **nên** triển khai `IEquatable<T>.Equals(T other)` và đảm bảo tương thích với `Equals(object)` + `GetHashCode()`.
Tính linh hoạt	Ít linh hoạt hơn (chỉ 1 toán tử; phải cẩn thận tương thích với `Equals`/`GetHashCode`).	Linh hoạt hơn: có thể override/triển khai `IEquatable<T>`, và có thêm `object.Equals(a,b)` (static) **an toàn null**. Với `string.Equals` có overload chọn `StringComparison`.
Tốc độ	Không có quy tắc “nhanh hơn” nói chung. Với `string`, `==` thực chất gọi `string.Equals(a,b)`. Chênh lệch thường **không đáng kể** sau JIT inline.	Tương tự: phụ thuộc kiểu cụ thể và triển khai `Equals`. Dùng `EqualityComparer<T>.Default` để nhận cách so sánh tối ưu cho T.
Sử dụng với `null`	**An toàn** (không ném NRE); `a == null` hợp lệ. Nếu tự overload, nhớ xử lý null đúng cách.	Gọi **instance** `a.Equals(b)` sẽ **ném NRE** nếu `a` là `null`. Dùng `object.Equals(a,b)` hoặc `a?.Equals(b) ?? false` để **an toàn null**.

Generic và Non-Generic

Đặc điểm	Non-generic	Generic

Khái niệm	Là kiểu dữ liệu truyền thống. Không khai báo cụ thể khi định nghĩa class, method, interface. Kiểu dữ liệu thực tế được xác định khi sử dụng.	Là kiểu dữ liệu được tham số hoá bởi một hoặc nhiều kiểu khác nhau. Cho phép khai báo class, method, interface với kiểu cụ thể. Kiểu này sẽ được định nghĩa và sử dụng xuyên suốt.
Ưu điểm	Đơn giản, dễ hiểu Hiệu quả cao hơn với các kiểu dữ liệu nguyên thuỷ	Tăng khả năng tái sử dụng code Đảm bảo an toàn dữ liệu vào và ra khi sử dụng Tốt hơn do được tối ưu hoá cho kiểu dữ liệu cụ thể
Nhược điểm	Kém linh hoạt, khó tái sử dụng. Thiếu an toàn do có thể sử dụng sai kiểu dữ liệu	Phức tạp hơn so với non-generic. Có thể ảnh hưởng hiệu suất với những kiểu dữ liệu phức tạp.
Sử dụng	Khi cần code đơn giản dễ hiểu, không yêu cầu tái sử dụng cao, không yêu cầu an toàn dữ liệu cao.	Khi cần code có khả năng tái sử dụng cao và yêu cầu tính an toàn dữ liệu cao.

Delegate và Event

Delegate	Event
- Là một tham chiếu đại diện cho phương thức. - Có thể gọi trực tiếp từ bất kỳ đâu có quyền truy cập. - Có thể gọi trực tiếp mà không có cơ chế bảo vệ nào nên có thể dẫn đến lỗi. - Ưu tiên tính linh hoạt.	- Là thành phần của lớp sử dụng delegate để quản lý danh sách các phương thức. - Chỉ có thể kích hoạt bên trong lớp mà nó được khai báo. - Có cơ chế bảo vệ ngăn chặn gọi từ bên ngoài. Chỉ cho phép đăng ký bên trong class thông quan += hoặc huỷ đăng ký qua -= - Ưu tiên an toàn và kiểm soát.

Struct và Class

Tiêu chí	struct	class
Bản chất	Kiểu **giá trị** (value type)	Kiểu **tham chiếu** (reference type)
Cấp phát & bộ nhớ	Nằm trên **stack hoặc heap** tùy vào nơi nó sống. **Sao chép theo giá trị**	Luôn cấp phát trên **heap**, biến giữ **tham chiếu** **Sao chép tham chiếu**
Vòng đời & GC	Không tạo đối tượng riêng trên heap ⇒ **ít áp lực GC** (trừ khi bị boxing)	Đối tượng do **GC** quản lý; cần cân nhắc phân bổ/thu hồi.

Kế thừa	**Không** kế thừa class; chỉ **implement interface**	**Có** inheritance + implement interface
Constructor mặc định	Luôn có `default` (zero-init); từ **C# 10** có thể **tự định nghĩa** constructor không tham số	Tự do định nghĩa constructor; không có zero-init bắt buộc
Nullability	Không thể `null` (trừ **nullable struct** `T?`)	Có thể `null` (nullable reference types)
Equality mặc định	So sánh **theo giá trị** (field-by-field)	So sánh **tham chiếu** (trừ khi override/record)
Boxing	Bị **boxing** khi chuyển sang `object/interface` (nếu không dùng generics ràng buộc) → phát sinh cấp phát	Không có khái niệm boxing
Bất biến/biến đổi	Nên thiết kế **nhỏ & bất biến** (immutable) để tránh chi phí copy	Linh hoạt cho **đối tượng phức tạp**, nhiều trạng thái
Hiệu năng (quy tắc thực tế)	**Tốt** cho dữ liệu **nhỏ** (≈ ≤ 16–32 byte), sống ngắn, dùng thường xuyên; tránh struct lớn/mutable vì **tốn copy**	Tốt cho mô hình phức tạp, cần chia sẻ qua tham chiếu, vòng đời dài
Generics	Tránh boxing với `where T : struct`; hỗ trợ `Nullable<T>`	Ràng buộc `where T : class` cho API dùng tham chiếu
Khi nên dùng	Để lưu những dữ liệu đơn giản, bất biến.	Đa phần các đối tượng có đối tượng có dữ liệu, hành vi phức tạp..
Unity lưu ý	Dùng **struct** (hoặc `readonly struct`) cho dữ liệu để **giảm GC** Cẩn thận copy trong Update	Dùng **class** cho `MonoBehaviour`, managers, asset refs; quản lý GC bằng pooling

Abstract class và Interface

Tiêu chí	Abstract class	Interface
Định nghĩa	Là lớp nền tảng có **biến chung** và **một phần code sẵn**. Lớp con kế thừa và hoàn thiện.	Là **bản hợp đồng** liệt kê chức năng. Lớp/struct thực hiện (implement) các chức năng đó.
Access Modifiers	**Có**	**Không**. Mặc định là public
Field, Method dùng chung	**Có**	**Có thể** có method mặc định (từ C# 8 trở lên), nhưng không có field, constructor
Kế thừa / triển khai	Mỗi lớp chỉ kế thừa **1 abstract class**.	Một lớp/struct có thể **implement nhiều interface**.
Dùng cho struct	**Không dùng** cho struct.	**Dùng được** cho struct.

Khi nên dùng	Dùng để tạo một **bản thiết kế** chung cho các đối tượng liên quan đến nhau. Cung cấp các chi tiết mà các đối tượng đó phải tuân theo. Khi cần constructor.	Tạo ra một **vai trò/chức năng** mà các đối tượng triển khai nó sẽ đảm nhiệm.
Ưu điểm	Tái sử dụng code mạnh, quản lý chung dễ, thêm thành viên thường ít phá vỡ lớp con.	Linh hoạt, áp dụng rộng (kể cả struct), hỗ trợ nhiều vai trò, rất hợp cho test/DI.
Nhược điểm	Ràng buộc thừa kế (chỉ 1 lớp cha), dễ tạo cây thừa kế sâu nếu lạm dụng.	Không có dữ liệu chung; nếu thêm phương thức mới có thể ảnh hưởng kiểu đã implement **(giảm rủi ro nhờ "method mặc định" từ C# 8).**
Ví dụ Unity	`BaseController`, `BaseState` chứa biến/logic chung cho nhiều state.	`IService`, `IRepository`, `ISaveSystem` để tiêm phụ thuộc và viết unit test.

Stack và Heap

Tiêu chí	Stack	Heap
Bản chất	Vùng nhớ tạm cho **biến cục bộ** & **method**	Vùng nhớ động cho **đối tượng sống lâu**
Lưu gì	Giá trị của biến cục bộ (bao gồm **struct nhỏ**), con trỏ return address	**Object của class**, mảng, **struct nằm trong object/mảng**, dữ liệu boxed
Cấp phát/giải phóng	Rất nhanh theo kiểu **LIFO**, trong method	**Linh hoạt**: cấp phát tự do, thu gom bởi **GC**
Quản lý	Tự động khi vào/ra hàm (không có GC)	**Managed heap** do GC quản lý (Gen0/1/2, LOH)
Tốc độ	Rất nhanh, liên tục, cache-friendly	Chậm hơn stack; có thể bị **tạm dừng (GC pause)**
Kích thước	Giới hạn (stack size)	Rộng hơn, tùy RAM/GC
Thời gian sống	**Ngắn**, theo phạm vi hàm	**Dài/linh hoạt**, theo tham chiếu còn sống
Khi nào xảy ra	Biến cục bộ, tham số, **struct** cục bộ	`new` class, mảng; **struct** là field trong class/mảng; **boxing**
Lưu ý quan trọng	Dữ liệu **biến mất** khi ra khỏi hàm	Tránh rác nhiều/đối tượng lớn gây áp lực **GC/LOH**

For và Foreach

Tiêu chí	for	foreach

Khái niệm	Vòng lặp theo **index** (`i = 0..n-1`)	Vòng lặp **duyệt từng phần tử** qua *enumerator.*
Cách hoạt động	Truy cập `collection[i]` mỗi bước	Gọi `GetEnumerator()` → `MoveNext()` → `Current`
Truy cập theo index	**Có**	**Không có**
Sửa phần tử	**Có thể** Sửa phần tử chỉ định qua index	**Không thể sửa** trong vòng lặp
Thêm/Xóa khi lặp	Có thể làm được (cẩn thận cập nhật chỉ số; thường lặp **ngược** khi xóa)	**Không được** thay đổi collection đang duyệt → dễ ném `InvalidOperationException`
Hiệu năng (thực tế)	Rất nhanh cho **mảng** và khi cần truy cập ngẫu nhiên	Rất nhanh cho **List** (enumerator là struct, thường **không cấp phát**); nhìn chung chênh lệch nhỏ, chọn theo **độ rõ ràng**
Độ rõ ràng mã	Tốt khi cần index/điều kiện bước nhảy tùy biến	Ngắn gọn, dễ đọc khi **chỉ duyệt**
Khi nên dùng	Cần **index**, cần **chèn/xóa** trong lúc lặp, hoặc cần nhảy bước	Duyệt đọc, không chỉnh sửa cấu trúc collection, code gọn
Lỗi thường gặp	Quên cập nhật `i`, truy cập vượt biên	Thêm/xóa trong vòng lặp.

Static và non-Static

class

Tiêu chí	Static	Non-static
Khởi tạo	Không thể `new`; không có instance	Có thể `new`; nhiều instance
Thành viên	**Tất cả phải là static**	Có cả instance và static
Kế thừa	Không kế thừa/không implement interface; ngầm `sealed`	Có thể kế thừa/implement interface
Dùng khi	Nhóm **hàm tiện ích/hằng số** không cần state riêng	Mô hình đối tượng có **state/đa hình**

field/method/property

Tiêu chí	Static	Non-static
Thuộc về	Class	Instance
Truy cập	`Type.Member`	`obj.Member`
Vòng đời	Tồn tại suốt thời gian chạy	Sống theo vòng đời instance (GC)
Trạng thái	**Chung toàn cục** → cần chú ý **thread-safety**	Tách biệt theo từng đối tượng

Dùng khi	Cấu hình bất biến, cache dùng chung, hằng số	Dữ liệu/hành vi gắn với **một** đối tượng

Others

Lập trình bất đồng bộ

Lập trình bất đồng bộ là kỹ thuật lập trình cho phép dừng tiến trình mà không chặn luồng chính.

- **async** là từ khóa dùng để khai báo phương thức bất đồng bộ. Phương thức bất đồng bộ có thể trả về giá trị hoặc không.
- **await** cho phép nhường quyền điều khiển và tiếp tục thực thi khi tác vụ hoàn tất.

Tác dụng:

- Giữ UI/Gameplay mượt khi chờ I/O (HTTP, file, DB).
- Dễ chạy **đồng thời** nhiều I/O với `Task.WhenAll(...)`.
- Đẩy việc **CPU-bound** ra nền bằng `Task.Run(...)`.

Ưu điểm:

- Code tuyến tính, dễ đọc.
- Không block thread → scalable.
- Hỗ trợ hủy (`CancellationToken`) và gom lỗi chuẩn.

Nhược điểm / Lỗi hay gặp:

- `.Result`/`.Wait()` trên main/UI → deadlock.
- `async void` (trừ event) khó bắt lỗi.
- Đồng bộ hoá context phức tạp (cẩn trọng `ConfigureAwait(false)` trong lib).

UniTask

UniTask là thư viện async tối ưu GC cho Unity. Cung cấp awaiter cho `AsyncOperation`, `UnityWebRequest`, Addressables… và tích hợp PlayerLoop.

Tác dụng:

- `await` trực tiếp Unity API (load scene, asset, web request).
- Giảm cấp phát so với `Task`.
- Điều khiển thời điểm tiếp tục (Update/LateUpdate/FixedUpdate).
- Hủy theo vòng đời object: `GetCancellationTokenOnDestroy()`.

Ưu điểm:

- Hiệu năng/GC tốt, sát Unity main thread.
- API phong phú: `UniTask.WhenAll`, `Delay`, `Yield`, `.Forget(handler)`.

Nhược điểm / Lưu ý:

- Thêm phụ thuộc vào bên thứ ba.
- Cần quản lý PlayerLoop & hủy đúng cách để tránh leak.
- Fire-and-forget phải **log/bắt lỗi** (`.Forget(ex => LogException)`).

Gợi ý nhanh:

- I/O: tạo nhiều tác vụ → `WhenAll`.
- CPU: `Task.Run`.
- Tránh `.Result`/`.Wait()`; tránh `async void`.
- Unity: ưu tiên **UniTask**, luôn dùng `CancellationToken` gắn vòng đời object.

Garbage Collection (GC)

GC là bộ thu gom rác tự động của .NET quản lý **vòng đời đối tượng** trên **managed heap. GC** tự phát hiện đối tượng **không còn tham chiếu**, thu hồi bộ nhớ, và (thường) **nén** vùng heap để giảm phân mảnh.

Cách ho?t ??ng

- **Phân tầng**:
  - Gen0: ngắn hạn, được dọn dẹp thường xuyên và rất nhanh
  - Gen1, Gen2: trung hạn và dài hạn, dữ liệu không bị dọn dẹp ở Gen0 sẽ được chuyển lên Gen1 và Gen2. Những tầng này ít bị dọn dẹp hơn. Khi dọn dẹp sẽ tốn thời gian hơn.
- **LOH (Large Object Heap)**:
  - Dữ liệu ≥ ~85KB cấp phát ở LOH;
  - Dữ liệu này thường **không nén** mỗi lần GC. Được sắp xếp lại mỗi lần dọn dẹp.
  - Có thể gây phân mảnh
- **Tạm dừng (STW)**: Để đảm bảo dọn dẹp hiệu quả GC sẽ yêu cầu chương trình tạm dừng để dọn dẹp.
- **Thời điểm kích hoạt**:
  - Khi thiếu bộ nhớ để cấp phát, áp lực bộ nhớ cao.
  - Khi gọi `GC.Collect()` (không khuyến khích).

Destructor & `IDisposable`

- **destructor** (`~Type()`): chạy **không xác định thời điểm** trên thread GC; chỉ dùng khi giữ **tài nguyên unmanaged**.
- **`IDisposable`/`using`**: giải phóng tài nguyên **đúng lúc** (socket, file, handle…). Với unmanaged, dùng pattern `Dispose` + `SafeHandle`.
- Quy tắc: **Ưu tiên `Dispose`**; chỉ dùng destructor như “lưới an toàn”.

Nh?ng hi?u l?m th??ng g?p

- “Dùng `GC.Collect()` là tối ưu” → **Sai**: thường làm **tệ hơn** vì tăng STW.
- “Dùng List/Dictionary để tránh phân mảnh” → **Không chính xác**: GC mới xử lý phân mảnh; vấn đề lớn ở **LOH** và **pinning**.
- “GC luôn chậm” → **Phụ thuộc mẫu cấp phát**; tối ưu **mẫu cấp phát** thường hiệu quả hơn “tắt/bắt” GC.

Best practices

- **Giảm cấp phát**: tái sử dụng buffer; dùng `ArrayPool<T>`, `StringBuilder`, cache hợp lý.
- **Tránh boxing** (struct → object), tránh LINQ tạo rác trong hot path.
- **Chú ý vòng đời tham chiếu**: đừng giữ tham chiếu lâu “vô tình” (static cache, event không hủy đăng ký) → đối tượng lên **Gen2**.
- **LOH**: hạn chế tạo nhiều mảng lớn; chia nhỏ/thuê từ `ArrayPool<T>`.
- **Pinning** (`fixed`, `GCHandleType.Pinned`): dùng ít – pin lâu gây phân mảnh.
- **API hiện đại**: `Span<T>`, `Memory<T>`, `ReadOnlySpan<T>` cho xử lý dữ liệu không cấp phát.
- **Thiết lập GC**:
  - **Server GC** (dịch vụ/đa nhân nặng) vs **Workstation GC** (ứng dụng desktop).
  - `GCLatencyMode.LowLatency`/`TryStartNoGCRegion` chỉ dùng ngắn hạn, có kiểm soát.

V?i Unity

- **Incremental GC** (Unity 2019.3+): giảm thời gian dừng bằng cách chia nhỏ công việc GC. Bật trong **Project Settings > Player > Garbage Collector**.
- **Tránh cấp phát mỗi frame**:
  - Không tạo string/`new` trong `Update()`; tránh LINQ/closure/boxing trong `Update()`.
  - Dùng **object pooling** cho bullets, VFX, particle, v.v.
  - Ưu tiên **struct**/Burst/DOTS cho luồng hiệu năng cao.
- **Profiler**: theo dõi `GC.Alloc` và spikes; kiểm tra callsite gây rác.
- **Addressables/Assets**: giải phóng đúng cách (`Release`/`UnloadUnusedAssets`) để giảm áp lực heap.

Checklist t?i ?u GC

- Không gọi `GC.Collect()` trừ khi thật cần.
- Dọn (unsubscribe) để tránh giữ tham chiếu lâu.
- Dùng `using`/`Dispose()` cho tài nguyên ngoài managed.
- Pool hóa bộ nhớ & đối tượng “ngắn hạn nóng”.
- Tránh cấp phát trong `Update()`/hot path; kiểm tra bằng .
- Cẩn trọng với & ; ưu tiên chia nhỏ hoặc `ArrayPool<T>`.
- Với Unity: bật khi phù hợp, tránh tạo rác frame-by-frame.

> *Tối ưu GC = tối ưu mẫu cấp phát + quản vòng đời tham chiếu*, không phải “gọi GC nhiều hơn”.