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

Revision #5
Created 2025-10-14 18:10:09 UTC by ThanhDV
Updated 2025-10-15 20:10:55 UTC by ThanhDV