Self study

(1)In C#, the **Garbage Collector (GC)** is a part of the .NET runtime that automatically manages memory — and it's one of your best friends when working with objects and references.

**What Does the Garbage Collector Do?**

* It **cleans up memory** by removing objects that your program **no longer uses or references**.
* These objects are stored in the **heap**, and over time, they can clutter your app’s memory.
* The GC helps prevent memory leaks and keeps your app running smoothly without needing you to manually free memory.

**How It Works (Simply Explained)**

1. Your code creates objects:

Person p = new Person();

1. When p is no longer used and no part of your code refers to it...
2. The **Garbage Collector notices this** and releases that memory.

It runs **automatically**, but you can force it manually using:

GC.Collect();

That’s rarely recommended — because the runtime usually knows best when to clean up.

## Why It Matters

* You don’t have to worry about manually freeing memory (like in C or C++).
* It makes development safer and faster — especially helpful when building backend systems with lots of object usage.
* But… knowing how the GC works helps you avoid mistakes like keeping unused references alive, which can block cleanup.

(2) especially since you’re digging deep into C# and backend systems. Type safety is one of those behind-the-scenes heroes that keeps your code from going off the rails.

**What Is Type Safety?**

**Type safety** means your code only operates on data in ways that are valid for that data’s type. It prevents you from:

* Accessing memory incorrectly
* Performing invalid operations (like treating a number as a string)
* Unexpected bugs due to “wrong type” assumptions

**In C#, Why It Matters**

C# is a *strongly typed* language, so type safety is built into its DNA.

* You can’t do this:

int myAge = "twenty-five"; // ❌ Compile error

* But you *can* do this:

int myAge = int.Parse("25"); // ✅ Valid conversion

The compiler checks your types before the program even runs. That’s a huge win for avoiding runtime crashes and unpredictable behavior.

## Real Benefits in Backend Development

Especially in databases and APIs:

* You avoid mismatched types between layers (e.g., string in frontend vs int in backend)
* You catch bugs early during compilation, not during live execution
* Your ERD and DBML mappings stay aligned — because every column’s type matters

## (3) Bitwise Operators in C# — Simplified

* **AND (**&**)**: Only keeps the bits that are 1 in both numbers. Example: 5 & 3 → 0101 & 0011 = 0001 → Result: 1
* **OR (**|**)**: Keeps any bit that's 1 in either number. Example: 5 | 3 → 0101 | 0011 = 0111 → Result: 7
* **XOR (**^**)**: Keeps bits that are different. Example: 5 ^ 3 → 0101 ^ 0011 = 0110 → Result: 6
* **NOT (**~**)**: Flips all bits — creates the bitwise complement. Example: ~5 → flips 0101 → becomes something like ...11111010 in binary → Result: -6 (using two's complement)
* **Left Shift (**<<**)**: Shifts bits to the left, multiplying the number by 2^n. Example: 5 << 1 → 0101 becomes 1010 → Result: 10
* **Right Shift (**>>**)**: Shifts bits to the right, dividing the number by 2^n. Example: 5 >> 1 → 0101 becomes 0010 → Result: 2

Each bit in a number is like a switch: ON (1) or OFF (0). These operators let you flip, combine, and shift them however you want.

Want to try building a bitwise permissions system like a backend flag setup