Self-Study

1. Bitwise Operators

   Bitwise operators manipulate individual bits of integers at the binary level. They are useful for low-level programming, flags, or optimizing certain operations.

   Bitwise Operators in C#

   1. ***AND (&)***: Compares bits; returns 1 if both bits are 1, else 0.
   2. **OR (|)**: Returns 1 if at least one bit is 1, else 0.
   3. **XOR (^)**: Returns 1 if exactly one bit is 1, else 0.
   4. **NOT (~)**: Flips all bits (1 to 0, 0 to 1).
   5. **Left Shift (<<)**: Shifts bits left, filling with zeros (multiplies by powers of 2).
   6. **Right Shift (>>)**: Shifts bits right, filling with the sign bit for signed types (divides by powers of 2).

Simple Example

Let's use two integers: a = 5 (binary: 0101) and b = 3 (binary: 0011).

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011

// Bitwise AND (&)
Console.WriteLine(a & b);  // Output: 1 (Binary: 0101 & 0011 = 0001)


// Bitwise OR (|)
Console.WriteLine(a | b);  // Output: 7 (Binary: 0101 | 0011 = 0111)

// Bitwise XOR (^)
Console.WriteLine(a ^ b);  // Output: 6 (Binary: 0101 ^ 0011 = 0110)

// Bitwise NOT (~)
Console.WriteLine(~a);     // Output: -6 (Binary: ~0101 = 1010, interpreted as
two's complement)

// Left Shift (<<)
Console.WriteLine(a << 1); // Output: 10 (Binary: 0101 << 1 = 1010, equivalent to
5 * 2)

// Right Shift (>>)
Console.WriteLine(a >> 1); // Output: 2 (Binary: 0101 >> 1 = 0010, equivalent to 5
/ 2)
```

2. Hashing *V*s Encoding

Hashing and encoding are distinct concepts in computer science, often confused due to their use in data transformation. Below is a concise comparison of **hashing** vs. **encoding** in the context of C#.

Hashing

Hashing transforms data (e.g., a string or object) into a fixed-length value (hash) using a hash function. The process is **one-way** (irreversible) and designed for data integrity, lookup efficiency, or security.

Irreversible: You cannot retrieve the original data from the hash.

Fixed Length: Output size is constant regardless of input size.

Deterministic: Same input always produces the same hash.

Collision Resistance: Ideally, different inputs produce different hashes (though collisions are possible).

Use Cases: Password storage, data integrity checks, hash tables, digital signatures.

C# Example (Using SH*A*256 for hashing):

```
using System.Security.Cryptography;
using System.Text;

string input = "Hello, World!";
byte[] hashBytes = SHA256.HashData(Encoding.UTF8.GetBytes(input));
string hash = Convert.ToBase64String(hashBytes);
Console.WriteLine(hash); // Output: Fixed-length base64 string (e.g.,
"pZGm1...==")
```

Encoding

Encoding transforms data into a different format or representation, typically for storage, transmission, or compatibility. It is **reversible** and preserves the original data.

Reversible: You can decode the output to recover the original data.

***V*ariable Length**: Output size depends on input size and encoding scheme.

Not for Security: Encoding is not designed for data protection (e.g., anyone can decode Base64).

Use Cases: Data serialization (e.g., Base64 for binary data), character encoding (e.g., UTF-8), URL encoding.

C# Example (Using Base64 encoding):

```
string input = "Hello, World!";
string encoded = Convert.ToBase64String(Encoding.UTF8.GetBytes(input));
Console.WriteLine(encoded); // Output: "SGVsbG8sIFdvcmxkIQ=="


// Decoding back
byte[] decodedBytes = Convert.FromBase64String(encoded);
string decoded = Encoding.UTF8.GetString(decodedBytes);
Console.WriteLine(decoded); // Output: "Hello, World!"
```

3. Checked Block or Keyword In C#

In C#, the checked block (or keyword) is used to enforce arithmetic overflow checking for integral-type operations and conversions. When code runs inside a checked block, any arithmetic operation or type conversion that results in an overflow (i.e., a value exceeding the target type's range) throws a System.OverflowException. This contrasts with the default behavior in C#, where overflows are silently ignored unless explicitly checked.

The unchecked keyword, conversely, explicitly disables overflow checking, allowing overflows to wrap around (e.g., exceeding the maximum value of an int resets to the minimum).

What Does a Checked Block Do?

Ensures that arithmetic operations (e.g., addition, multiplication) or type conversions involving integral types (int, long, byte, etc.) throw a System.OverflowException if the result exceeds the type's valid range.

Applies to all integral arithmetic and conversions within the block unless overridden by an unchecked block inside it.

Without checked, C# performs arithmetic in an unchecked context, where overflows wrap around (e.g., int.MaxValue + 1 becomes int.MinValue).

Example of checked

```
int max = int.MaxValue; // 2147483647

checked
{
   int result = max + 1; // Throws System.OverflowException
}
```