Modular Operation(Math.floorMod(), %)

Code:

```
System.out.println("-17 mod 12 = " + Math.floorMod(-17, 12));
System.out.println("-17 mod 12 = " + (-17 % 12));
System.out.println("-5 mod 2 = " + Math.floorMod(-5, 2));
System.out.println("-5 mod 2 = " + (-5 % 2));
System.out.println("-5 mod 2 = " + Math.abs(-5 % 2));
```

Output:

```
-17 mod 12 = 7
-17 mod 12 = -5
-5 mod 2 = 1
-5 mod 2 = -1
-5 mod 2 = 1
```

Modular Arithmetic in Java

Expression	Dividend	Divisor	Result	Calculation
17 % 12	17	12	5	17 ÷ 12 = 1 (truncated). Remainder = 17 - (1 × 12) = 5. Positive dividend → positive result.
Math.floorMod(17, 12)	17	12	5	17 ÷ 12 = 1 (floor = 1). Remainder = 17 - (1 × 12) = 5. Matches %for positive dividend.
-17 % 12	-17	12	-5	-17 ÷ 12 = -1 (truncated toward 0). Remainder = -17 - (-1 × 12) = -5. Sign follows dividend.
Math.floorMod(-17, 12)	-17	12	7	-17 ÷ 12 = -2 (floor = -2). Remainder = -17 - (-2 × 12) = 7. Always non-negative if divisor > 0.
-5 % 2	-5	2	-1	-5 ÷ 2 = -2 (truncated). Remainder = -5 - (-2 × 2) = -1.

Expression	Dividend	Divisor	Result	Calculation
Math.floorMod(-5, 2)	-5	2	1	Sign follows dividend. -5 ÷ 2 = -3 (floor = -3). Remainder = -5 - (-3 × 2) = 1. Positive result.
Math.abs(-5 % 2)	-5	2	1	Compute -5 % 2 = -1, then absolute → 1. Works here, but fails if divisor is negative.

Key Takeaways

- For **positive dividends**, % and floorMod give the **same result**.
- For **negative dividends**, % keeps the sign of the dividend, while floorMod ensures the result stays **non-negative** (if divisor > 0).
- Never rely on Math.abs(a % b) as a replacement for floorMod.

Use-Cases

- % → **signed remainder** (useful for *profit/loss type calculations*).
- Math.floorMod() → **true modulus** (useful for *clock arithmetic, array wrapping, parity checks*).

Demo:

```java
public class ModulusDemo {
    public static void main(String[] args) {
        // =========================
        // 1. Profit/Loss Example (% is correct)
        // =========================
        int profitLoss = -17 % 12;  // the remainder keeps sign
        System.out.println("Profit/Loss calculation:");
        System.out.println("-17 % 12 = " + profitLoss);
        // Output: -5 → means loss of 5
        System.out.println();

        // =========================
        // 2. Clock Arithmetic (floorMod is correct)
        // =========================
        System.out.println("Clock calculation (2 o’clock - 5 
hours):");
        int hour = Math.floorMod(2 - 5, 12);
        System.out.println("Math.floorMod(2-5, 12) = " + hour);
        // Output: 9 → valid hour on a clock
        System.out.println();

        // ==========================
        // 3. Array Index Wrapping (floorMod is correct -
Circular Way)
        // ==========================
        int arraySize = 10;
        int index = Math.floorMod(-1, arraySize);
        System.out.println("Array index wrapping:");
        System.out.println("Math.floorMod(-1, 10) = " + index);
        // Output: 9 → valid last index
    }
}
```

Output:

```
Profit/Loss calculation:

-17 % 12 = -5


Clock calculation (2 o’clock - 5 hours):

Math.floorMod(2-5, 12) = 9


Array index wrapping:

Math.floorMod(-1, 10) = 9
```

Java floorMod() Logic

```
public static int floorMod(int x, int y) {
   final int r = x % y;
   // if the signs are different and modulo not zero, adjust result
   if ((x ^ y) < 0 && r != 0) {
      return r + y;
   }
   return r;
}
```

What is XOR?

- ^ in Java is **bitwise XOR** (exclusive OR).
- Rule: For each bit, result is 1 if bits differ, 0 if bits are same.
- So:

  1 ^ 1 = 0
  0 ^ 0 = 0
  1 ^ 0 = 1
  0 ^ 1 = 1

Example 1: floorMod(-17, 12)

- Step 1: r = -17 % 12 = -5
- Step 2: (x ^ y) < 0 → (-17 ^ 12) is negative → signs differ
- Step 3: r != 0 → -5 != 0
- Step 4: return r + y = -5 + 12 = 7

Result: 7

Example 2: floorMod(17, 12)

- Step 1: r = 17 % 12 = 5
- Step 2: (17 ^ 12) is positive → signs same
- Skip adjustment.
- Return r = 5

Result: 5

Example 3: floorMod(-20, 5)

- Step 1: r = -20 % 5 = 0
- Step 2: (x ^ y) negative → signs differ
- Step 3: But r == 0
- Return 0

Result: **0** (divides evenly, no adjustment needed)
