Please explain the following code so I may use it in an interview.

public class B0050 {

    public double **myPow**(double x, int n) {

        if(n < 0){

            n = -n;

            x = 1 / x;

        }

        double pow = 1;

        while(n != 0){

            if((n & 1) != 0){

                pow \*= x;

            }

            x \*= x;

            n >>>= 1;

        }

        return pow;

    }

}

This code defines a class **B0050** which has a single method, **myPow**. The method takes two arguments, a double **x** and an integer **n**, and returns a double.

The method first checks if **n** is less than 0, and if so, sets **n** to its absolute value and sets **x** to the reciprocal of **x** (i.e. 1/x).

A variable **pow** is initialized to 1. The method then enters a while loop that continues as long as **n** is not equal to 0. Within the loop, if the binary AND of **n** and 1 is not equal to 0 (i.e. if the least significant bit of **n** is 1), **pow** is multiplied by **x**.

Then, **x** is multiplied by itself and **n** is assigned the result of a bitwise unsigned right shift by 1 (**>>>= 1**)

Finally, the method returns **pow**

This code is using bit manipulation and exponentiation by squaring to compute x^n in O(log n) time. It is more efficient than a simple for loop or recursion.

What is absolute value and reciprocal of x?

The absolute value of a number is its distance from 0 on the number line, regardless of whether the number is positive or negative. For example, the absolute value of -5 is 5, and the absolute value of 5 is also 5.

Reciprocal of x, is the multiplicative inverse of x, which is 1/x. For example, the reciprocal of 2 is 1/2, and the reciprocal of 1/3 is 3. In the code provided, when n<0, x is set to the reciprocal of x, this is done in order to calculate the power of x when the exponent is negative.

Explain to me like I am 12 about absolute value

Okay, imagine a number line, like the one you might see in a math book, with negative numbers on the left, zero in the middle, and positive numbers on the right. The absolute value of a number is how far that number is from zero, no matter whether it's on the left side or the right side of zero.

For example, if the number is -5, it is 5 spaces away from zero on the left side, so the absolute value of -5 is 5. If the number is 5, it is also 5 spaces away from zero on the right side. So, no matter if the number is negative or positive, the absolute value is always positive.

Does that make sense?

If the absolute value is always positive then why n = -n ?

You're correct that the absolute value of a number is always positive. The line **n = -n** is used to change the sign of **n** from negative to positive when **n** is less than zero. This is done because the rest of the code is written to calculate the power of x when the exponent is positive, So, to use the same code for negative exponents, the sign of the exponent must be flipped to positive before proceeding. The result is then multiplied by 1/x, which is equivalent to taking the reciprocal of x to get the correct output for negative exponents.

Why you need to use this code

x \*= x;

n >>>= 1;

The code **x \*= x;** is used to square the value of **x**. This is important because it allows the method to use a technique called "exponentiation by squaring" to calculate the power of **x** to the exponent **n**.

Exponentiation by squaring is a way to calculate the power of a number more efficiently than by repeatedly multiplying the number by itself. The basic idea is that if you need to calculate **x^n**, instead of multiplying **x** by itself **n** times, you can repeatedly square **x**, and at each step, if the current exponent is odd, you also multiply by **x** one more time.

The line **n >>>= 1;** is used to shift the bits of **n** to the right by 1 bit. This is the same as dividing **n** by 2, but it's more efficient to do it this way. This is done to keep track of the exponent after each step of the while loop, since we are doing the exponentiation by squaring, in each step we are dividing the exponent by 2.

By using both of these operations together, the method is able to calculate the power of **x** to the exponent **n** more efficiently than by using a simple loop or recursion, by reducing the number of multiplications that need to be done.

Why you need to use this code if((n & 1) != 0){ pow \*= x; }

The line **if((n & 1) != 0)** is used to check if the least significant bit of **n** is 1. The **&** operator is a bitwise AND operator, which compares each bit of the first operand (**n**) to the corresponding bit of the second operand (1) and returns 1 if both bits are 1, and 0 otherwise. So, **n & 1** will be 1 if the least significant bit of **n** is 1, and 0 otherwise. Then, the != 0 check is used to check if the result of n & 1 is not equal to zero, which is true if the least significant bit of n is 1.

If the least significant bit of **n** is 1, this means that the current value of **n** is odd, and that **x** should be multiplied by **pow**. The line **pow \*= x;** is used to do this. This is done because of the exponentiation by squaring, as I mentioned before, we are dividing the exponent by 2 in each step, by checking if the exponent is odd or even, we know if we need to multiply the current value of pow by x or not.

So, this line of code is used to check if the current exponent is odd, and if it is, it multiplies the **pow** by **x** one more time.

In summary, this part of the code is used to check if the current exponent is odd, and if it is, it multiplies the **pow** by **x** one more time, which is needed to correctly calculate the power of x to the exponent n using the exponentiation by squaring.