BINARY IN PROGRAMMING

At their very lowest-level, binary is what drives all electronics. As such, encountering binary in computer programming is inevitable.

Representing binary values in a program

In Arduino, and most other programming languages, a binary number can be represented by a `0b` preceding the binary number. Without that `0b` the number will just be a decimal number.

For example, these two numbers in code would produce two very different values:

```
COPY CODE

a = 0b01101010; // Decimal 106
c = 01101010;  // Decimal 1,101,010 - no 0b prefix means decimal
```

Bitwise operators in programming

Each of the bitwise operators discussed a few pages ago can be performed in a programming language.

AND bitwise operator

To AND two different binary values, use the **ampersand**, &, operator. For example:

```
COPY CODE

x = 0b10011010 & 0b01000110;
// x would equal 0b00000010
```

AND'ing a binary value is useful if you need to apply a **bit-mask** to a value, or check if a specific bit in a binary number is *1*.

The AND bitwise operator shouldn't be confused with the AND conditional operation, which uses the double-ampersand (&&) and produces a true or false based on the input of multiple logic statements.

OR bitwise operator

The OR bitwise operator is the **pipe** | (shift+\, the key below backspace). For example:

```
COPY CODE

y = 0b10011010 | 0b01000110;
// y would equal 0b11011110
```

OR'ing a binary value is useful if you want to set one or more bits in a number to be *1*.

As with AND, make sure you don't switch up the OR bitwise operator with the OR conditional operator - the double-pipe (||).

NOT bitwise operator

The bitwise NOT operator is the **tilde** `~` (shift+`, the key above tab). As an example:

```
COPY CODE

z = ~(0b10100110);
// z would equal 0b01011001
```

XOR bitwise operator

To XOR two values use the **caret** (`^`) between them:

```
COPY CODE

r = 0b10011010 ^ 0b01000110;
// r would equal 0b11011100
```

XOR is useful for checking if bits are different, because it'll only result in a *1* if it operates on both a *0* or *1*.

Shifting left and right

To shift a binary number left or right *n* bits, use the `<<n` or `>>n` operators. A couple examples:

```
COPY CODE

i = 0b10100101 << 4;    // Shift i left 4 bits
// i would equal 0b101001010000
j = 0b10010010 >> 2;    // Shift j right 2 bits
// j would equal 0b00100100
```

Shifts are an especially efficient way to multiply or divide by powers of two. In the example above, shifting four units to the left multiplies that value by 2^4 (16). The second example, shifting two bits to the right, would divide that number by 2^2 (4).

Source: https://learn.sparkfun.com/tutorials/binary