Binary Number Tutorial

This brief tutorial is intended to serve as a basic introduction to binary numbers. As a programmer it is important to have a basic knowledge of the binary number system in order to better understand what happens in a computer when certain types of operations are performed on data that is stored in memory.

As humans, it is very natural for us to use the **decimal** system to express numbers. The decimal system expresses numbers built around the powers of 10. We all find it very easy to understand the meaning of numbers like 32 and 123.

Let’s take slightly more mathematical look at what it means to express numbers in the decimal system. The digits 0 through 9 can be used to express a number in the decimal system. Take the number 123 as an example. The number 123 is really:

$$1 \text{ x } 10^2 + 2 \text{ x } 10^1 + 3 \text{ x } 10^0$$

Recall from basic mathematics that 10 to the second power is 10 times 10 or 100; 10 to the first power is 10; and 10 to the zero power is 1.

Computers don’t use the decimal system when they process numbers and other information. They use zeros and ones instead. Because computers are composed of electronic switches that are either turned on or off, computers use the **binary** system to express numbers and programming instructions and basically any type of information that they need to process.

The binary system is built around powers of 2, and only the digits 0 and 1 can be used in this system. For example, the binary number 1011 is equivalent to:

$$1 \text{ x } 2^3 + 0 \text{ x } 2^2 + 1 \text{ x } 2^1 + 1 \text{ x } 2^0$$

The binary number 1011 is equivalent to the decimal number 11, since 2 to the third power is 8, 2 to the first power is 2, and 2 to the zero power is 1.

Each digit in the binary system is called a bit. When a computer reserves storage for a numerical data type, it reserves a certain number of bits depending on the specific type of number. In Java, for example, there are four integer data types: byte, short, int, and long. Java stores byte types in 8 bits of memory, short types in 16 bits of memory, int types in 32 bits of memory, and long types in 64 bits of memory.

Let’s take Java’s byte type and figure out what range of values that type can take on. The same approach can be used for the other integer types, but illustrating the byte type will get us where we need to go. Since byte types are stored in 8 bits of memory, we'll use the following table in our discussion.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	1	0	1	1

This table illustrates the binary number 1011 which we have seen is equivalent to the decimal number 11. When computer programmers think of the series of 0's and 1's associated with a binary number, they refer to this series as the **bit pattern** for the number.

If all 8 bits were used to store a byte type then byte variables could take on values between 0 (binary 00000000) and 255 (binary 11111111). But, Java doesn't use all 8 bits. It uses 7 bits to store the value of a number and 1 bit to represent a number's sign. That way, both positive and negative numbers can be allowed. The high bit is used to represent the sign of a number. A value of 0 corresponds to a positive sign and a value of 1 corresponds to a negative sign.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	1	1

So, the largest positive number that can be represented in 7 bits of memory is illustrated in the above table. If you work through the math for this bit pattern you would find that it is equivalent to the decimal value 127.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	0	0	0	0

The smallest negative number that can be represented in 7 bits of memory is represented by the above bit pattern. To deal with negative numbers, Java uses something called **two's complement notation**, which we won't get into here. So, looking at the two above bit patterns, byte variables in Java can take on integer values between -128 and 127.

Understanding bit patterns and what happens when they are manipulated will be important when we use Java's bitwise operators and type casting operations.