Bit Shifting

What the shift?

Now that we've gotten some of the basics of number representation, we can start getting into a few of the operations. In particular, let's discuss shifts.

Note: All Java bitwise operators are performed on `int`s. Thus, to use other bit-length data types, you need to cast.

Left Shift

The symbol we use for left shifts is **<<**. Suppose we have a `byte a = 0b11001010`. We can shift all of the bits to the left by 4 bits:

```
byte a = (byte) 0b11001010;
a = (byte) (a << 4);
a == (byte) 0b10100000; // evaluates to true
```

The result is `0b10100000`. Shifting all of a number's bits to the left by 1 bit is equivalent to multiplying the number by 2. Thus, all of a number's bits to the left by `n` bits is equivalent to multiplying that number by 2^n.

Notice that we fill in the spots that open up with 0s. If a bit goes further left than the place of the most-significant digit, the bit is lost. This means that left shifting a number too far will result in *overflow*, where the number of bits that are saved by the data type are insufficient to represent the actual number.

For example, let's say I have a byte representing 4. In binary, this would be `0b00000100`. If I shift this left by 3 bits, I am multiplying 4 by 2^3. This results in 32, which is `0b00100000`.

```
0b00000100 << 3 = 0b00100000
```

However, if I shift 32 (`0b00000100`) over to the left by more than 2 bits, I lose its value. Let's say I shift it to the left by 3 bits. 32 * 2^3 is 256. Refer to the chart at the beginning of this lab. The largest number a byte can represent is 127. Thus, I cannot represent 256 using only a single byte, and when I left shift 32 over by 3 bits, it becomes 0.

```
0b00100000 << 3 = 0b00000000
```

If I was using an `int` rather than a byte, I could left shift over by a few more bits without losing my value.

Right Shifts

Right shifts can be a bit trickier because there are two types of right shifts: arithmetic and logical shifts.

Just as left shifts are equivalent to multiplying a number by 2, right shifts are equivalent to dividing a number by 2. However, when we shift bits to the right, a 1 in the sign bit can represent a larger positive number rather than a smaller negative number. Logical shifts treat the number as an unsigned number, while arithmetic shifts treat the number as a signed number.

1. **Logical Right Shift**

   Logical right shifts are denoted by **>>>**. They shift all bits to the right and fill in vacated places with 0s (quite similarly to left shifts).

   ```
   0b01001010 >>> 4 = 0b00000100
   ```

2. **Arithmetic Right Shift**

   Arithmetic right shifts are denoted by **>>**. They take into account the most significant digit such that the digit used to replace vacated positions is the value of the most-significant digit before any shifts occur.

   ```
   0b11001010 >> 4 = 0b11111100 0b01001010 >> 4 = 0b00000100
   ```

Self-tests: Shifts

Give the decimal value of `int b` after each of the following operations is applied:

```
int b = -1; b = b >> 30;
```

Toggle Solution

```
int b = -1; b = b >>> 31;
```

Toggle Solution

```
int b = 3; b = b << 3;
```

Toggle Solution

In general, what is the value of `x << n`?

Toggle Solution