Special Topic 4.2

Binary Numbers

You are familiar with decimal numbers, which use the digits 0, 1, 2, . . . , 9. Each digit has a place value of 1, 10, $100 = 10^2$, $1000 = 10^3$, and so on. For example,

$$435 = 4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$$

Fractional digits have place values with negative powers of ten: $0.1 = 10^{-1}$, $0.01 = 10^{-2}$, and so on. For example,

$$4.35 = 4 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

Computers use binary numbers instead, which have just two digits (0 and 1) and place values that are powers of 2. Binary numbers are easier for computers to manipulate, because it is easier to build logic circuits that differentiate between "off" and "on" than it is to build circuits that can accurately tell ten different voltage levels apart.

It is easy to transform a binary number into a decimal number. Just compute the powers of two that correspond to ones in the binary number. For example,

$$1101 \text{ binary} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13$$

Fractional binary numbers use negative powers of two. For example,

$$1.101 \text{ binary} = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 0.5 + 0.125 = 1.625$$

Converting decimal numbers to binary numbers is a little trickier. Here is an algorithm that converts a decimal integer into its binary equivalent: Keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the last one. For example,

$$\begin{aligned}
100 \div 2 &= 50 \text{ remainder } 0 \\
50 \div 2 &= 25 \text{ remainder } 0 \\
25 \div 2 &= 12 \text{ remainder } 1 \\
12 \div 2 &= 6 \text{ remainder } 0 \\
6 \div 2 &= 3 \text{ remainder } 0 \\
3 \div 2 &= 1 \text{ remainder } 1 \\
1 \div 2 &= 0 \text{ remainder } 1
\end{aligned}$$

Therefore, 100 in decimal is 1100100 in binary.

To convert a fractional number <1 to its binary format, keep multiplying by 2. If the result is >1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the first one. For example,

$$\begin{aligned}
0.35 \cdot 2 &= 0.7 \\
0.7 \cdot 2 &= 1.4 \\
0.4 \cdot 2 &= 0.8 \\
0.8 \cdot 2 &= 1.6 \\
0.6 \cdot 2 &= 1.2 \\
0.2 \cdot 2 &= 0.4
\end{aligned}$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 . . .

To convert any floating-point number into binary, convert the whole part and the fractional part separately. For example, 4.35 is 100.01 0110 0110 0110 . . . in binary.

You don't actually need to know about binary numbers to program in Java, but at times it can be helpful to understand a little about them. For example, knowing that an `int` is represented as a 32-bit binary number explains why the largest integer that you can represent in Java is 0111 1111 1111 1111 1111 1111 1111 1111 binary = 2,147,483,647 decimal. (The first bit is the sign bit. It is off for positive values.)

To convert an integer into its binary representation, you can use the static `toString` method of the `Integer` class. The call `Integer.toString(n, 2)` returns a string with the binary digits of the integer `n`. Conversely, you can convert a string containing binary digits into an integer with the call `Integer.parseInt(digitString, 2)`. In both of these method calls, the second parameter denotes the base of the number system. It can be any number between 0 and 36. You can use these two methods to convert between decimal and binary integers. However, the Java library has no convenient method to do the same for floating-point numbers.

Now you can see why we had to fight with a roundoff error when computing 100 times 4.35. If you actually carry out the long multiplication, you get:

```
1 1 0 0 1 0 0 * 1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0 ...

1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0 ...
  1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 ...
    0
      0
        1 0 0.0 1|0 1 1 0|0 1 1 0 ...
          0
            0
_____________________________________
1 1 0 1 1 0 0 1 0.1 1 1 1 1 1 1 1 ...
```

That is, the result is 434, followed by an infinite number of 1s. The fractional part of the product is the binary equivalent of an infinite decimal fraction 0.999999 . . . , which is equal to 1. But the CPU can store only a finite number of 1s, and it discards some of them when converting the result to a decimal number.