Two's complement notation

Sign-magnitude notation

Consider integers represented using 8 bits, as with Java's type **byte**. In the sign-magnitude representation, the leftmost bit is used for the sign: 0 means positive, 1 means negative. The other 7 bits contain the magnitude of the number, in binary. This representation is depicted in the box to the right.

Sign-magnitude has problems. First, there are two representations of 0, with different signs. Second, binary arithmetic (e.g. addition) is difficult to implement in hardware in this representation. We don't explain why, but you will see below how, in two's complement notation, addition is relatively easy.

Sign-magnitude

8 bits	decimal
00000000	+0
00000001	+1
…	
01111110	+126
01111111	+127
10000000	–0
10000001	–1
…	
11111110	–126
11111111	–127

Two's complement notation

According to Wikipedia, John von Neuman suggested using two's complement notation in a 1945 draft of a proposal for a computer. Today, just about all computers use two's complement notation for integers.

Assuming an 8-bit representation, as with type **byte**, two's-complement notation is depicted to the right. Here are important points about the notation:

Two's complement

8 bits	decimal
00000000	0
00000001	+1
…	
01111110	+126
01111111	+127
10000000	–128
10000001	–127
…	
11111110	–2
11111111	–1

1. There is only one representation of 0.
2. For a non-negative integer: First bit is 0. The rest give its binary representation.
3. For a negative integer: First bit is 1. To get the representation of its absolute value as an 8-bit binary number, complement and add 1:

integer	-1: 11111111	-127: 10000001	-128: 10000000
complement	00000000	01111110	01111111
add 1	1: 00000001	127: 01111111	128: 10000000

4. The examples below show that binary addition works even when the numbers have different signs. Just do conventional addition with carry, but use the binary number system. In the second example, there is a carry of 1 (in red); that bit is deleted since only 8 bits can be used.

```
  10000000   -128          01111111   +127
+ 00000001 + +1          + 11111111 + -1
  10000001   -127         101111110  +126
```

5. Below, you see that adding 1 to the largest number, +127, produces the smallest number! That is why, in Java and most languages, wrap-around and not overflow is used. The rightmost example also shows wraparound: 126 + 126 should be 252, but there are not enough bits to represent that number in 8 bits. Subtracting 256 (the number of different integers in this 8-bit representation) from 252 gives -4.

```
  01111111   +127          01111111   +127          01111110   +126
+ 00000001 + +1          + 10000001 + -127        + 01111110 + +126
  10000000   -128         100000000    0           11111100     -4
```

Casting in Java

To the right are the integers 127 and -127 in 8-bit and 16-bit two's complement. Evidently, cast to a wider integral type by prepending the leftmost bit an appropriate number of times.

byte	short	dec
01111111	0000000001111111	127
10000001	1111111110000001	-127

Cast to a narrower n-bit type by throwing away all but the n rightmost bits.
For example, (**short**)128 is 0000000010000000 and (**byte**)(**short**)128 is 10000000.

Comment. We have just touched the surface of interesting information about two's complement. For example, we haven't shown you subtraction. We haven't shown you how to convert a decimal integer into two's complement. We haven't explained why two's complement is easier to implement in hardware than sign-magnitude. You can find out more from other sources, like Wikipedia.