**Manipulating non-byte-sized data in Java**

These days, many computer science students don't get a chance to manipulate bit-level data representations—something mature computer scientists take for granted because they spent half their undergraduate life writing low-level assembly routines.

One might argue that high level programming, particularly with object-oriented languages, has mitigated the need to *fiddle with bits*. We just don't generally need to go that low these days. If we need a bit, then we would probably reach for an object of type Boolean and fake it. But, there are occasions when utilising individual bits, or subsequences of bits, or data that does not come in standard byte-sized pieces, is a necessary and powerful programming technique. Many web sites exist to explain basic bit operations in Java (to find out more, simply *Google* "bit operators") and how to use them to access and manipulate individual bits. The goal of this short tutorial is to give a brief overview of what is needed to insert non-byte-size data into primitive data types so that they can be output more efficiently—so-called *bit-packing*.

**operations needed for bit-packing**

When using a high level object-oriented programming language like Java or C# for bit-packing, one first needs to be able get at individual bits within larger primitive data types such as ints, floats, and chars. Simple operations like setting or testing a single bit involves the bit-wise operators for AND, OR, XOR and NOT (which are &, |, ^ and ~, respectively ... although the last one is really a bit complement operator). Plenty of web sites describe how to use these operations, so we'll take it as given that you know what they do, or can find out easily enough.

The other thing we need to be able to do is shift bit pattterns left and right to align them the way we want. To do this we need to learn bit-pattern operators like shift left (<<), signed shift right (>>) and unsigned shift right (>>>). The first one simply takes a bit-pattern and shifts it to the left (as the *arrows* imply), bringing in zero bits as needed into the right-most (i.e. least significant) bit position, and losing any left-most (i.e. most significant) bits that get shuffled off the end.

As an example, if we have a 32-bit Java int, X, with the value 6 in it then it looks like this:

X = 00000000000000000000000000000110

To shift X five bits left (and keep the result) we execute the assignment statement X = X << 5, or alternatively the shorthand shift assignment X <<= 5, giving

X = 00000000000000000000000011000000

which, incidentally, yields the same effect as multiplying X by 2 five times over (i.e. multiplying by 32), but faster—which can be a useful thing to know (but not for the discussion here).

Signed shift right (>>) does the same thing but going the other way. One big difference, however, is that bits coming into the most significant position are set to have the same value as the bit being shifted out of that position so that the sign of the resulting bit pattern is not affected. Bits going out the right are simply lost. If we want cleared bits (i.e. zero bits) coming into the most significant position then we use the unsigned right shift operator (>>>). These operations are more correctly called arithmetic shift right and logic shift right respectively, and there are few other details that are important to know about these operators, but you'' have to Google "bit shifting in Java" if you want a tutorial on that.

This discussion is about placing a bit pattern of unusual length into an output stream (or extracting it from an input stream) — which is just a little bit trickier. That is to say, we address the question of how to pack pits into an output stream when they don't fit neatly within common byte-sized data types like chars and ints?

**the bit-packing process**

Imagine you have a ten-bit unsigned integer, P', and an eight-bit character, C', that you want to *insert* into an outgoing bit-stream in Java (as you might do, for example, when implementing an LZ78 data compressor). We picture the corresponding bit patterns for these as follows:

P' = pppppppppp

C' = cccccccc

Now, if we assume we can only output data in multiples of bytes then we could pack P' and C' into, say, a 32-bit Java int, B, with fourteen bits left over (and those fourteen bits might be used for storing more irregularly sized data). Imagine that the first ten bits of B are being used for some other datum called X, and the remaining bits are available for us to use. The first thing we might do is make sure those unused bits are cleared (i.e. set to zero) which we can do by AND-ing with B a bit-mask pattern that has the ten most significant bits set and the rest cleared, such that if

X = xxxxxxxxxx

B = xxxxxxxxxx??????????????????????

MASK = 11111111110000000000000000000000

then the operation B = B & MASK yields

B = xxxxxxxxxx0000000000000000000000

Our goal is this: We want to put the pattern ppppppppppcccccccc right up against X in B, leaving a few bits unused at the right end of B.

Now we might assume P' and C' are already in 32-bit int variables, P and C, such that they really look like this

P = 0000000000000000000000pppppppppp

C = 000000000000000000000000cccccccc

The first thing we have to do is align the most significant bit of P' (i.e. the tenth bit from the right in P) so that it's in the same place respectively as the first unused bit after X. We do this by shifting P to the left 12 places using the java left shift operator in an assignment statement (i.e. P = P << 12;) . .. which gives us

X = xxxxxxxxxx

B = xxxxxxxxxx0000000000000000000000

P = 0000000000pppppppppp000000000000

Now we can see that P is lined up under the unused bits of B; and we copy them into B with the OR operator (B = B | P) giving

B = xxxxxxxxxxpppppppppp000000000000

That is, whatever bits that are in P are copied up into B while leaving the rest of B's bits untouched. Note that the twelve least significant bits of B (i.e. right most bits) are still unused/cleared. To get C in there, we first left shift C four bits (using C = C << 4) to get

B = xxxxxxxxxxpppppppppp000000000000

C = 00000000000000000000cccccccc0000

and copy C up into B the same way we got P up there, using B = B | C giving

B = xxxxxxxxxxppppppppppcccccccc0000

If we now want to put another 10-bit number after C', there's not enough room. There's a number of ways we can make room. One was is to copy the left-most two bytes of B (which are all good data bits) into a couple of bytes and write them out to their destination (because we can output bytes easily enough), then we shift the remaining/rightmost two bytes of B into the leftmost position of B using shift operations, giving

B = pppppcccccccc00000000000000000000

Now there's enough room to put more bits after C using the same technique described above.

Note that, while it is perhaps easier to see how bits can be packed in the most-signficant position (i.e. left-to-right), it may be easier to pack them in the least significant position (i.e. right-to-left) as they can be removed for output simply by casting the working 32-bit int as a byte, as with

byte b = (byte) B

**Alternative cheat**

Another way to output packed bits is to simply compute a sequence of characters made only of 1's and 0's that correspond to a printed version of your bit stream, and then feed them into a method that uses them to set the actual bits in an output buffer.

**Unpacking bits**

Recalling how the MASK was used to preserve the most significant ten-bits during packing through a bit-wise AND, one can see it can similarly be used to extract the same bits through the same operation.

Tony C. Smith, 28/03/2013