JavaBits

Table of Contents

Introduction 2

Number Bases 3

- Base 10 3
- Base 2 3
- Hexadecimal 5
- Octal 7

Bitwise Logic 8

- Java logical operators 9
- The bitwise operators 9
- Bit shifts 10
- A better binary 10
- Bit masks 11

Characters 13

- Terminology 13
- Common coded character sets 14
- Unicode 14
- Characters in Java 15

Integers 18

- Two's complement 19
- Range of this method 20
- Sign and magnitude 20
- Overflow 21

Floating Point 22

- Mantissa and exponent 22
- Mantissa and exponent in binary 22
- IEEE754 22

Introduction

This is about bit patterns in digital systems. It uses Java code to look at the principles, and examines how bit patterns are used in Java.

A bit pattern is something like 0011 1010 0011 0000 - a pattern of 1s and 0s. All software in a digital system is made of bit patterns. That means all program code and all data. The data is might be numbers - these are bit patterns. Or the data might be an image, made of pixels, made of bit patterns. Or audio. Or text. All bit patterns.

Bit patterns are often grouped into sections of 8 bits, known as a *byte*. So 0011 1010 0011 0000 is 2 bytes long.

Program code includes source code, which is a sequence of characters (Unicode for Java), and this is a sequence of bit patterns. So is bytecode. So is native machine code.

To be precise, everything is a set of states of two-state devices. RAM consists of electronic switches which are either open or closed, giving two different voltage levels. A standard hard disc stored data as magnetic patterns, magnetised in one direction or the other. A CD has a spiral of short or long pits on a shiny layer. For each of these, we interpret the two states as being a 1 or a 0 (or a yes or no, or true or false). When the CD is read, the pits are converted voltage levels - but the bit pattern is the same.

If we have a pattern like 0011 1010 0011 0000 - is that a machine code instruction, or a number, or a pixel, or text? Could be any. They are all represented by bit patterns, so they all look the same. Only the context tells us if we should treat these as pixels or text or whatever.

We talk about the 'most significant bits' or 'leading bits' or 'upper bits' in a bit pattern as being the left-most ones, and the least significant bits as the right-hand ones.

Number Bases

A 'bit' is a binary digit - a 0 or a 1. These are the digits in 'base 2'. So we need to look at number bases first.

Get the idea of the difference between a *number* and the way it is *represented*. The number 3 can be represented as III in Roman numerals, or 11 in base 2, or 3 in base 10. III and 11 and 3 are three different representations of the same number.

A *digit* is one symbol - like 3 or 7 or 9. A *number* is made of one or more digits - like 12.34.

Base 10

'Ordinary' numbers are written in *base 10* (or decimal or denary). That means we use a set of digits, with different places having different *place values*, like this:

Place value	Thousands	Hundreds	Tens	Units
	10^3	10^2	10^1	10^0
Digits	3	2	4	6

So the number 3246 means 3 thousands, 2 hundreds, 4 tens and 6 units.

Notice the place values go up in powers of 10, and we have 10 digits to use - 0 1 2 3 4 5 6 7 8 and 9.

The pattern continues to the right of the 'decimal point', like this:

Place value	Tens	Units	Tenths	Hundredths
	10^1	10^0	10^{-1}	10^{-2}
Digits	7	2	. 1	6

So 72.16 is 7 tens, 2 units, 1 tenth and 6 hundredths.

Base 2

We can use the same idea in other bases. In base 2 the place values are powers of 2 - units, 2, 4 8 16 and so on. We only have the digits 0 and 1 to use:

Place value	Eights	Fours	Twos	Units
	2^3	2^2	2^1	2^0
Digits	1	1	0	1

So 1101 is 1 eight, 1 four, 0 twos and 1 unit = 8 + 4 + 1 = 13.

When we are writing in different number bases, we use a subscript to show which base it is in. So $1101_2 = 13_{10}$

Here is counting in base 2:

Base 2	Base 10
0	0
1	1
10	2
11	3
100	4
101	5
110	6

111	7
1000	8
1001	9
1010	10

To change a number from base 2 to decimal, just add up the place values where there is a 1 - for example:
$10010_2 = 16 + 2 = 20_{10}$

To change decimal to binary, for small numbers, pick out powers of 2 to total the number. For example

$19_{10} = 16 + 3 = 16 + 2 + 1 = 10011_2$

For larger numbers, repeatedly divide by 2 and track the remainders. For example to change 328 into binary:

2 into 328 = 164 remainder 0
2 into 164 = 82 remainder 0
2 into 82 = 41 remainder 0
2 into 41 = 20 remainder 1
2 into 20 = 10 remainder 0
2 into 10 = 5 remainder 0
2 into 5 = 2 remainder 1
2 into 2 = 1 remainder 0
so reading the remainders back:

$328_{10} = 101001000_2$

In effect we are finding powers of 2 as before but more methodically. In practice it is faster and safer to do this using a calculator in appropriate mode.

Exercise

1. Change binary 11001 into decimal
2. Change 12 into binary
3. Change 421 into binary
4. How can you tell if a binary number is even, by looking at it?

To convert a decimal fraction to binary:

1. Multiply by 2
2. The whole number part is the next bit. Ignore it when you..
3. Repeat step 1, until you get to zero, or a repeating pattern.

Example: 0.75 - multiply by 2:

1.50 First bit is 1. Now use 0.50

1.00 Second bit is 1, Got to .00, so end

So $0.75_{10}=0.11_2$ (since 0.75 = 1/2 + 1/4)

Example 0.625

1.250 first bit 1, use .250

0.50 second bit 0

1.00 third bit 1, ended

So $0.625_{10}=0.101_2$
Example 0.1

0.2 first bit 0
0.4
0.8
1.6
1.2 (but we've had .2 before..)
0.4
0.8
1.6
1.2
0.4
0.8
1.6
1.2 ..
so 0.1 decimal = binary .00011001100110011..
So 0.1 is an example to show that some values have infinite binary fraction expansions - just like 1/3 is 0.3333.. in base 10. We will see later that computer arithmetic with numbers other than integers has only limited accuracy.

Exercise

1. Change decimal 0.125 into binary

In source code we can use binary like:

```
    int a=0b1101;
    System.out.println(a); // 13
    int b=0b1100_1111;
    String t = Integer.toBinaryString(b);
    System.out.println(t); // 11001111
```

Binary literals were only introduced in Java 7.

Hexadecimal

We often use numbers in base 16, known as hexadecimal or hex. Here the place values are powers of 16. For example:

Place value	4096's	256's	Sixteens	Units
	16^3	16^2	16^1	16^0
Digits	2	0	7	1

So 2071_{16} = 2 X 4096 + 7 X 16 + 1 X 1 = 8305_{10}

The big difference is that we need 16 different digits, and we only have 10, as 0 to 9. The problem is solved by using A to F as 10 to 15. So counting in hex (and binary) is like this:

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	00010000
11	17	00010001

The reason that hex is used so often is that it is very easy to convert from binary to hex, but hex notation is much shorter than binary. You change hex to binary by replacing each hex digit by the corresponding 4 bits - like

$A2B3_{16}$ = $1010\ 0010\ 1011\ 0011_2$

and in reverse for binary to hex. Most current processors are 64 bit, meaning they handle 64 bits or 8 bytes at a time. So a value a processor might process might be

1111 1010 1000 0011 0000 1110 1111 1010 1000 0101 1100 0011 1111 1010 1000 0011

which is almost impossible to write down without making a mistake. The corresponding hex version is not quite so bad:

F A 8 3 0 E F A 8 5 C 3 F A 8 3

One byte is 2 hex digits or 8 bits.

Exercise

1. Change binary 1001 into hex
2. Change AF_{16} into binary
3. Change 12_{16} into decimal
4. Change 1100 0011 into hex

We can use hex in Java:

```
     int a=0xC;
     System.out.println(a); // 12
     int b=12;
     String t = Integer.toHexString(b);
     System.out.println(t); // c
```

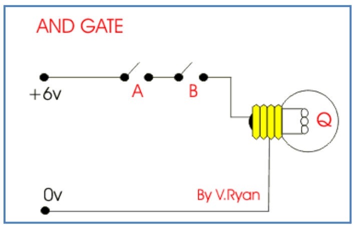
Octal

This is base 8. Place values are powers of 8, and we use the digits 0 to 7. This is like hex, but using groups of 3 bits:

Octal	Decimal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
10	8	001000
11	9	001001

Bitwise Logic

Digital electronics consists of 'logic gates'. One of these is an *AND gate*. The image show sthe idea. We have two inputs A and B, and an output Q. The lamp is only lit (output 1) if both switches A and B are closed - 2 1 inputs. (image by V. Ryan from http://www.technologystudent.com/). Real AND gates use transistors or diodes instead of switches, they are formed on silicon on chips, are extremely small and extremely fast. The current record is over 2.5 billion transistors, in Intel's 10-core Xeon Westmere-EX. But the idea is just the same as a couple of switches.



We can summarise what an AND gate does using a 'truth table'

Inputs		Output
0	0	0
0	1	0
1	0	0
1	1	1

A truth table shows all possible combinations of inputs, and the corresponding output. We could say an AND gate is 'both'.

A second basic gate is an *OR*. Its truth table is

Inputs		Output
0	0	0
0	1	1
1	0	1
1	1	1

So OR means either, or both.

The third is an *XOR*, or exclusive OR. This is 'either, but not both':

Inputs		Output
0	0	0
0	1	1
1	0	1
1	1	0

Another way to see this is the output is 1 if the inputs are different.

The final gate we need is a *NOT*, which has just one input, and the output is the 'opposite':

Input	Output
0	1
1	0

These hardware gates have equivalents in Java code, in two very different ways.

Java logical operators

These take two (or one for !) operators of type boolean, and produce a boolean result. && is AND, || is OR, and ! is NOT. These are often used in 'if' statements, like:

```
if ( (x==2) && (y<3) )
.. what to do if x is 2 and y is less than 3
```

The bitwise operators

These take 1 or 2 *bit patterns*, and produce a new bit pattern as the result. The result is formed by applying the operation to each pair of bits. & is AND, | is OR. For example:

```
    int b1=0B11111010;
    int b2=0B11111100;
    int b3 =  (b1 & b2);
    String r=Integer.toBinaryString(b3);
    System.out.println(r); // 1111 1000
```

This uses 'binary literals' introduced in Java 7. You might need to update.

Why 1111 1000? We are ANDing pairs of bits:

b1	1	1	1	1	1	0	1	0
b2	1	1	1	1	1	1	0	0
b3	1	1	1	1	1	0	0	0
	1 AND 1 = 1					0 AND 1 = 0	1 AND 0 = 0	0 AND 0 = 0

Or:

```
    int b1=0B11111010;
    int b2=0B11111100;
    int b3 =  (b1 | b2);
    String r=Integer.toBinaryString(b3);
    System.out.println(r); // 1111 1110
```

The bitwise NOT is ~

```

    int b1=0B11111111_11111111_11111111_11111010;
    int b3 =  ~b1;
    String r=Integer.toBinaryString(b3);
    System.out.println(r); // 101
```

We need to explain this. The Java 7 binary literal is of type int, and so has 4 bytes = 32 bits. When we say

```
    int b1=0B11111010;
```

the leading bits are taken to be 0s, so this is actually 0000 0000 1111 1010.

We want the leading bits to be 1's, so we say

```
    int b1=0B11111111_11111111_11111111_11111010;
```

This also uses the feature of being able to put in _ to track the bit positions.

When we ~ this, all those 1s turn to 0s. The 3 least significant bits 010 are inverted to 101, which is output. toBinaryString omits leading 0s.

The XOR bitwise operator is ^.

Bit shifts

The bit shift operators produce a new bit pattern by moving the bits in an input pattern left or right a number of places. >> shifts to the right, and << to the left. For example:

int a = b << 4;

means a has the same bits as b, but shifted 4 places to the left.

We will return to bit shifts after seeing two's complement.

We can use a bit shift to write a better routine to convert an int to a displayable binary pattern.

A better binary

Integer.toBinaryString suppresses leading 0s, and does not insert spaces every 4 bits. Here is a better version:

```
  static String binary(int i) {
    StringBuilder s = new StringBuilder();
    for (int bitPlace = 0; bitPlace < 32; bitPlace++) {
      int bit = i & 1;
      i >>=  1;
      if (bitPlace % 4 == 0) {
        s.append(" ");
      }
      s.append(bit);
    }
    s = s.reverse();
    return new String(s);
  }
```

This forms the result in 's'. A StringBuilder is like a String, but is mutable so a lot more efficient here.

We have a loop going round 32 places, since we know ints have 32 bits.

```
      int bit = i & 1;
```

This uses the idea of *a bit mask*. We are ANDing each bit in i with 1, which in binary will be 00000..001. If you AND a bit with 0, you get 0. If you AND it with 1, you get the bit unchanged ( ) AND 1 = 0, 1 AND 1 = 1). So i & 1 makes every bit 0, except for the right-most one, which is unchanged. So i & 1 gets us the rightmost bit.

```
      i >>=  1;
```

is the same as

```
      i = i >>  1;
```

so it shifts the bits in i right one place.

```
      s.append(bit);
```

puts our bit on the end of s.

```
        if (bitPlace % 4 == 0) {
          s.append(" ");
        }
```
puts a space in the result every four places to make it more readable.

The append builds up the bits left to right, so we need

```
      s = s.reverse();
```
Finally

```
      return new String(s);
```
returns a new String constructed from the StringBuilder s.

For example

```
      int b1 = 0B1010;
      System.out.println(binary(b1));
      // get 0000 0000 0000 0000 0000 0000 0000 1010
      int b2 = b1 << 4;
      System.out.println(binary(b2));
      // get 0000 0000 0000 0000 0000 0000 1010 0000
```

Bit masks

A bit mask is a bit pattern which we AND or OR with a given pattern, to isolate or change specified bits.

If you AND something with a bit mask, you force to zero those bits where you have a 0, and leave the rest unchanged. For example:

```
      int b1 = 0B1010;
      System.out.println(binary(b1));
      // 0000 0000 0000 0000 0000 0000 0000 1010
      int b2 = b1 & 0B0011;
      System.out.println(binary(b2));
      // 0000 0000 0000 0000 0000 0000 0000 0010
```
If you OR something with a bit mask, you force bits to 1 where the mask has a 1:

```
      int b1 = 0B1010;
      System.out.println(binary(b1));
      // 0000 0000 0000 0000 0000 0000 0000 1010
      int b2 = b1 | 0B11110000;
      System.out.println(binary(b2));
      // 0000 0000 0000 0000 0000 0000 1111 1010
```
If you XOR a bit mask, you invert the 1 bits. If you do it twice, you get back to where you started from

```
int b1 = 0B1010;
System.out.println(binary(b1));
\\ 0000 0000 0000 0000 0000 0000 0000 1010
int b2 = b1 ^ 0B11111111;
System.out.println(binary(b2));
\\ 0000 0000 0000 0000 0000 0000 1111 0101
b2 = b2 ^ 0B11111111;
System.out.println(binary(b2));
\\ 0000 0000 0000 0000 0000 0000 0000 1010
```

Characters

The Java char primitive uses Unicode. We can easily convert these to a binary String:

```
  static String binary(char i) {
    StringBuilder s = new StringBuilder();
    for (int bitPlace = 0; bitPlace < 16; bitPlace++) {
      int bit = i & 1;
      i >>= 1;
      if (bitPlace % 4 == 0) {
        s.append(" ");
      }
      s.append(bit);
    }
    s = s.reverse();
    return new String(s);
  }
```

and use this like:

```
    char c = 'A'; // as in ASCII
    System.out.println(binary(c));
    // top 8 bits 0, bottom same as ASCII
    // 0000 0000 0100 0001
    c = 'Δ'; //Greek upper case delta
    System.out.println(binary(c));
    // 0000 0011 1001 0100
```

This seems straight-forward - simple and dull. It's not. Oh boy, it's not.

Terminology

One problem is that getting a precise definition of the terms involved with this is difficult, together with the fact that people do not use definitions consistently. A good discussion is:

http://www.ardentex.com/publications/charsets-encodings-java.pdf

Here we go:

Character

An abstracted idea of the smallest semantic element of 'writing'. So 'i' is a character. The dot on the 'i' is not. 'a' and 'b' are different characters. 'a' and '*a*' and '**a**' are the same character.

Glyph

A graphic corresponding to a character. A picture shape

Font

A set of glyphs. Like Helvetica

Character set

A set of characters

Coded character set

A set of characters where each one has a unique number. Unicode calls the number the *code point*.

Encoding

A mapping between code points of a coded character set and digital representations. The digital representation might use varying numbers of bits, and if there is more than 1, the bytes might be in different orders. Hence different encodings.

We could have the same encoding (say 8 bits) on different coded character sets. People often do not distinguish between coded character set and encoding.

Common coded character sets

ASCII

From 1963, having 128 characters. Also called US_ASCII.

Encodings - the lower 7 bits in 1 byte per character, top bit zero.

Or as 7 bits in a byte, with the top bit used as a *parity bit*.

ISO 8859-1

256 characters. The first 128 are the same as ASCII, and 160-255 other characters, including accented ones, and 128 to 159 undefined.

Encodings - just 1 - 1 byte per character.

Windows 1252

Also called 'code page 1252' or 'ANSI code page' (not really an ANSI standard).

Same as ISO 8859-1, except that 128 to 159 get Windows-specific definitions.

Encodings - just 1 - 1 byte per character.

Unicode

The first version of Unicode was published in 1991, with code points going from 0000 to FFFF and so had 64k possible characters. But in 1996 Unicode 2.0 was released, with more. The 2.0 version extended the range 16-fold, by adding an extra 15 'code planes', so that 00 0000 - 00 FFFF was the 'basic multi-lingual plane = BMP' (plane 0) , 10 0000 - 1F FFFF was the 'supplementary multi-lingual plane SMP' (plane 1), 20 000 - 2F FFFF was the 'supplementary ideographic plane' (plane 2), followed by 11 unassigned planes, a special-purpose plane and a private use plane (a private jet).

Several different *encodings* are used with the Unicode *coded character set* (in other words, how the code point is digitally represented).

UTF-16

This uses 1 or 2 16-bit words for each Unicode character. The BMP (0000 to FFFF) is stored as 1 16-bit word. However, D800 to DFFF are never used to represent 'real' characters - these are reserved, and a pair of these (a 'surrogate pair') are used to represent characters with code point over 10000. The algorithm to split such a code point into a pair is:

Subtract 0x10000 from the code point (the offset over 10000)

Add the top 10 bits of this to D800. This gives the first of the pair.

Add the bottom 10 bits to DC00. This gives the second of the pair.

Software reading UTF-16 encoded data can detect 16bits in the range D800 to DFFF, and will know this must be the first word of a 2 word pair, so can reverse the algorithm to obtain the code point.

The two bytes in each 16bit word might be lower byte first (*little endian*) or upper byte first (*big endian*). UTF-16 encoded data therefore starts with a *byte order mark BOM*. FFFE means little endian, and FEFF means big endian.

UTF-16LE and UTF-16BE
Same as UTF-16 except little endian or big endian. Therefore no *BOM*.

UTF-8
Unicode characters are represented as 1 to 4 bytes. The first 128 are 1 byte. The next 1920 are in 2 bytes. The rest of the BMP are in 3 bytes, and the 4 bytes accomodate the rest.

Leading bits tell software how many further bytes to read. For example if the leading bit of the first byte is 0, no more bytes are needed.

UTF-8 encoded data often has a BOM of EFBBBF at the start. This is widely recognised as nonsense. The byte order of UTF-8 is fixed (effectively big endian) so a BOM is not needed, and Unicode recommends it is omitted - but lots of software puts it in. It is possible to use the EFBBBF to detect that the data is UTF-8.

UTF-32
All Unicode characters are stored as 4 bytes. Simple but inefficient. UTF-32 with BOM, UTF-32BE and UTF-32LE resolve endianess.

Characters in Java
With this background we can understand characters in Java a little better.

Java uses the Unicode coded character set.

Internally (in chars and Strings) it uses UTF-16 encoding.

For external data, it can use (read or write) a range of encodings. In other words it can read a data file encoded in UTF-8, and convert it to the internal UTF-16 encoding. Obviously this will only work if the data actually is UTF-8, so this is a possible problem.

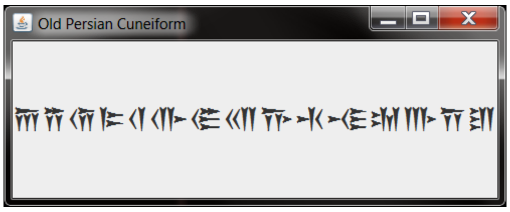
Unicode in Swing
You cannot System.out.println a SMP character. The console is unlikely to be able to cope with most of the BMP, let alone the SMP. Swing can do this, provided we are using a font which is able to render these. As of 2012, the only font which seems to do this is code2001 by James Klass. This is freeware, but the official download site has gone - which is a pity, since the next time you need to display Old Persian Cuneiform, you'll need it. It is now mirrored on

http://waltermilner.com

The heart of the following is Character.toChars(codePoint) which produces a char array for the corresponding int codePoint:

```
    StringBuilder s = new StringBuilder();
    for (int codePoint = 0x103a0; codePoint < 0x103AF; codePoint++) {
      s.append(Character.toChars(codePoint));
    }
    String string = new String(s);

    JFrame frame = new JFrame("Old Persian Cuneiform");
    frame.setVisible(true);
    frame.setSize(500, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JLabel label = new JLabel(string);
    Font font = new Font("Code2001", Font.PLAIN, 26);
    label.setFont(font);
    frame.add(label);
```

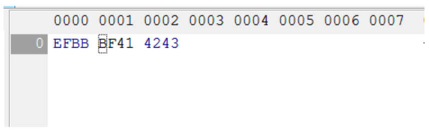


File reading and writing

Suppose we write a file:

```
try {
      File makefile = new File("output.txt");
      FileWriter fwrite = new FileWriter(makefile);
      fwrite.write('A');
      fwrite.write('B');
      fwrite.write('C');
      fwrite.flush();
      fwrite.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
```

What actually do we get in output.txt?



```
    0000 0001 0002 0003 0004 0005 0006 0007
  0 EFBB BF41 4243
```

So we get 6 bytes. The first 3 - EF BB BF - are the crazy UTF-8 BOM. Then we get A B and C as 8 bit bytes each. Why is it UTF-8? Because that's the default encoding:

```
Charset charset = Charset.defaultCharset();
System.out.println("Default encoding: " + charset + " (Aliases: "+ charset.aliases() + ")");
```

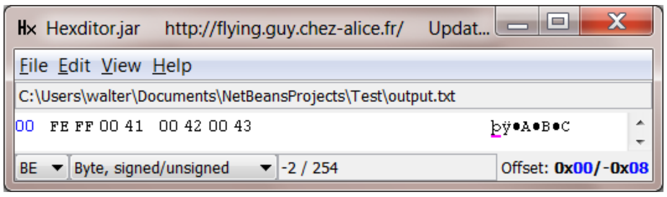
Default encoding: UTF-8 (Aliases: [UTF8, unicode-1-1-utf-8])

Java refers to an encoding as a CharSet. The API says " The name of this class is taken from the terms used in [RFC 2278](). In that document a *charset* is defined as the combination of a coded character set and a character-encoding scheme."

To choose an encoding we can use an OutputStreamWriter:

```
        Charset charset = Charset.forName("UTF-16");
        OutputStream outputStream = new FileOutputStream("output.txt");
        Writer writer = new OutputStreamWriter(outputStream, charset);
        writer.write('A');
        writer.write('B');
        writer.write('C');
        writer.close();
```

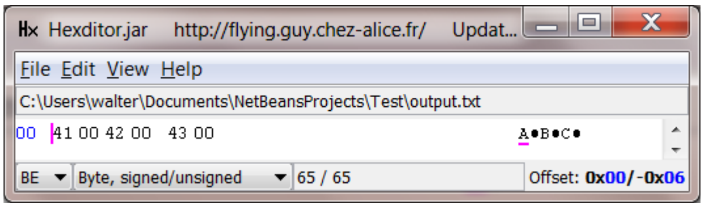
then the file contains:



```
C:\Users\walter\Documents\NetBeansProjects\Test\output.txt
00  FE FF 00 41  00 42 00 43          þÿ•A•B•C
BE  Byte, signed/unsigned  -2 / 254   Offset: 0x00/-0x08
```

This is the big endian BOM FEFF followed by 3 16bit words for A B and C.

If we change the encoding to UTF-16LE, we lose the BOM and get lower byte first:



```
C:\Users\walter\Documents\NetBeansProjects\Test\output.txt
00  41 00 42 00  43 00          A•B•C•
BE  Byte, signed/unsigned  65 / 65   Offset: 0x00/-0x06
```

Integers

We have seen how numbers can be written in binary. The obvious way to represent integers is as straight binary, so 3 would be 11 in memory, and 8 would be 1000. There are two problems with this. Bigger numbers have more bits, so how would we know how long the number is? For example how do we tell if 1111 is two 3s next to each other, or 15? Second problem is how to store negative numbers.

The first problem is solved by having fixed width types. A byte is 1 byte long, a short is 2 bytes, an int 4 bytes, and a long 8 bytes. We typically default to using an int. Use a byte if you know the value will be less than 128 and you have an enormous number of them. Use a long if the value will be over the 2 billion odd maximum of an int. If you need indefinitely large integers with hundreds or thousands of digits, use the BigInteger class.

Let's look at the negative number issuse - using bytes rather than ints, to handle fewer bits. This converts a byte to a binary string:

```
  static String binary(byte i) {
    StringBuilder s = new StringBuilder();
    for (int bitPlace = 0; bitPlace < 8; bitPlace++) {
      int bit = i & 1;
      i >>= 1;
      if (bitPlace % 4 == 0) {
        s.append(" ");
      }
      s.append(bit);
    }
    s = s.reverse();
    return new String(s);
  }
```

and this tries it out:

```
    for (byte b = 10 ; b>-11; b--)
      System.out.println(b+" : "+binary(b));
```

which produces:

```
10 : 0000 1010
9 : 0000 1001
8 : 0000 1000
7 : 0000 0111
6 : 0000 0110
5 : 0000 0101
4 : 0000 0100
3 : 0000 0011
2 : 0000 0010
1 : 0000 0001
0 : 0000 0000
-1 : 1111 1111
-2 : 1111 1110
-3 : 1111 1101
-4 : 1111 1100
-5 : 1111 1011
-6 : 1111 1010
-7 : 1111 1001
```

-8 : 1111 1000
-9 : 1111 0111
-10 : 1111 0110

So 0, 1,2, 3.. are in binary, in 8 bits, since we are using a byte. An int would be the same, but in 32 bits.
But -1, -2, -3.. are different. This is like a car mileometer. Going forward from 0 you get 1,2,3. Going backwards from 0 you get 999, 998, 997, 996 - but this is binary so you get 11111111, 11111110 and so on.

Two's complement

This method of representation is called two's complement. To form it, this is what you do. If the number is positive, you just change it to base 2. If it is negative, you change it to binary, invert it (change 0 to 1 and vice versa) and add 1. For example, -3

1. In binary it is 0000 0011 (in 8 bits)

2. Invert: 1111 1100

3. Add 1:

1	1	1	1	1	1	0	0	
							1	+
1	1	1	1	1	1	0	1	

So -3 is 1111 1101

Sometimes you need a carry. For example -4:

1. In binary it is 0000 0100 (in 8 bits)

2. Invert: 1111 1011

3. Add 1:

1	1	1	1	1	0	1	1	
							1	+
1	1	1	1	1	1	0	0	
					1	1		Carry

So -4 is 1111 1100

To interpret a two's complement number (change from binary to normal) : if the leading bit is 0, the number is positive - just change from base 2 to base 10. If it is 1, the number is negative. Form the two's complement, then change to decimal : the value is minus what you get.

For example: 1110 1010

The leading bit is 1, so this is negative:

Invert it : 0001 0101

Add 1:

0	0	0	1	0	1	0	1	
							1	+
0	0	0	1	0	1	1	0	
						1		Carry

10110 = 2 + 4+16 =22

so 1110 1010 = -22

Range of this method

If you are using n bits, the largest positive number you can have is 01111.. (n-1 bits). You can't have 1111111.. since that will be negative (in fact, -1). This is 2^n-1.

The smallest (biggest and negative) is 10000..(n-1 bits). This is -2^n

So in 8 bits the range is from 2^8-1 = 127, to -2^8 which is -128

Arithmetic and logical bit shifts

There are in fact two right shifts.

>> is an arithmetic right shift. If the number is negative, it will have leading bit 1, and 1's are shifted in - to keep it negative. If it is positive, 0s are shifted in.

>>> is a logical shift. Always 0s are shifted in.

For example:

```
    int b=-3;
    System.out.println(binary(b));
    System.out.println(binary(b>>1));
    System.out.println(binary(b>>>1));
```

```
1111 1111 1111 1111 1111 1111 1111 1101
1111 1111 1111 1111 1111 1111 1111 1110
0111 1111 1111 1111 1111 1111 1111 1110
```

Exercise

What is the fastest way to divide an int by 2?

Sign and magnitude

This is an alternative method. The leading bit is a sign bit : 0 for + and 1 for negative.

For example 0000 0010 = +2

1000 0001 = -1

This is not usually used (and is not used for ints by Java). We have

0000 0000 = +0 and

1000 0000 = -0

so we have two versions of zero. Also the range is 2^n-1 to -(2^n-1) which is one less than two's complement.

Overflow

```
byte b1 = 127;
byte b2= 1;
byte b3= (byte) (b1+b2);
System.out.println(b3); // result -128
```

When you add two bytes, the result is an int, so we have to cast it to a byte.

This goes wrong, because +128 is larger than the maximum we can get in 1 byte. It actually does this:

0111 1111 + 1 = 1000 0000

but this has leading bit 1, so is negative = -128.

This is arithmetic overflow. An unhandled arithmetic overflow (in Ada) in the engine steering software was the primary cause of the crash of the maiden flight of the Ariane 5 rocket.

Floating Point

We have seen that Java represents integral types (byte, short int and long) as two's complement. What about decimals? We have float and double types. How do they represent values?

Mantissa and exponent

For example, the number 283.89. In 'scientific notation' this is written as 2.8389 X 10^2. This the idea - to store the number in two parts:

28389	The mantissa
2	The exponent

That's it. This is the basic idea.

We will look at a simpel version, then how it works in Java.

Mantissa and exponent in binary

Since we are using digital systems, everything must be in binary. So the mantissa and exponent are in binary, and the exponent is the power of two, not ten.

They are both fixed width. Suppose we have an 8 bit mantissa and 4 bit exponent (the real thing, a double in Java, has 53 bit mantissa and at least 15 bits exponent).

What would 13.75 look like? First write it in binary

$13.75_{10} = 1101.11_2$ (the binary fraction values are 1/2 and 1/4, so 0.75 = .11)

Then adjust the point to before the first digit

1101.11 = .110111 X 2^4

So the exponent is 410=1002, and the mantissa is 0.110111. In our format this would be

0	.	1	1	0	1	1	1	0	0	1	0	0
Mantissa									Exponent			

What is the biggest number we can have? The biggest exponent is 0111 (1000 would be negative, since this is two's complement). The biggest mantissa is 0111 1111. So the biggest value is

0.111 1111 X 2^{0111} = 0.111 1111 X 2^7 = 1111111.0_2 = 127

The biggest Java double is 1.79769313486231570 X 10^{308} which is Double.MAX_VALUE.

IEEE754

Java uses the IEEE754 standard for representing 32bit floats and 64bit doubles. We will look at 32bit floats - 64bit is the same idea, but with added bits.

The 32 bits are used as three fields - a sign bit, an 8bit exponent and a 23bit mantissa:

S	8 bit exponent								23bit mantissa																						

Sign - 0 for positive, 1 for a negative number

Exponent - the 8 bits hold the 'actual' exponent plus a 'bias' of 127

Mantissa - the fraction part, with the number adjusted so that the leading 1 bit is to the left of the point - for example 1.0101. Since we know the leading bit is 1, we do not need to store it. So if the stored mantissa is 0101000000.. this represents 1.0101

Here is a routine to output a float formatted accordingly, without going mad counting bit places:

```
  static String IEEE754(float f) {
    // change 32bit float to an int with same bits
    int i = Float.floatToRawIntBits(f);
    // form corresponding binary String
    StringBuilder s = new StringBuilder();
    for (int bitPlace = 0; bitPlace < 32; bitPlace++) {
      int bit = i & 1;
      i >>= 1;
      // put in spaces to split fields
      if (bitPlace == 31 || bitPlace == 23) {
        s.append(" : ");
      }

    s = s.reverse();
    return new String(s);
  }
```

We will use this to look at some examples:

```
    float f = 0.15625f;
    System.out.println(IEEE754(f));
```

output :

0 : 01111100 : 01000000000000000000000

now $0.1565_{10} = 0.00101_2 = 1.01 \times 2^{-3}$

The sign bit is 0 because the number is positive.

The mantissa is 01000.. representing 1.010000..

The exponent is -3 +127 (bias) = 124 = 1111100_2

Another:

```
    float f = -12.75f;
    System.out.println(IEEE754(f));
```

output:

1 : 10000010 : 10011000000000000000000

$12.75_{10}=1100.11_2 = 1.10011 \times 2^3$

Sign bit is 1 because the number is negative

The mantissa is 10011000.. representing 1.10011

The exponent is 3 + 127 = 130 = 10000010

Most important example:

```
    float f = 0.1f;
    System.out.println(IEEE754(f));
```

output:

0 : 01111011 : 10011001100110011001101

0.1_{10} = .00011001100110011 0011.. = 1.10011001100110011... X 2^{-4}

Sign bit 0 : positive number

Exponent = -4+127 = 123 = 1111011

Mantissa corresponds to 1.10011001100110011...

But the point is that the mantissa only holds 23 bits, whereas the exact value has an *infinite* number of bits. So 0.1 *cannot be represented exactly* in this format. The same is true for many values - they *cannot be represented exactly*. In turn arithmetic on them will not be exact. If we use double instead, we get more accuracy - but still not exact.

To confirm, let's print out 0.1 to 30 decimal places using printf:

```
    float x=1.0f/10.0f;
    System.out.printf("%.30f \n",x);
```

0.100000001490116120000000000000

The details of how floating point values are represented are rarely important. But this point is:

FLOATING POINT IS NOT EXACT

One consequence of this is that floating point types should not be used to represent currency. Customers will not be at all happy with approximate invoices.

Finally:

```
    double root2 = Math.sqrt(2);
    if (root2*root2 == 2.0)
      System.out.println("Right");
    else
      System.out.println("Wrong!!"); // guess what
```

The square root of 2 squared should be 2 of course, but that's not what you get. But this is a bit 😊 unfair. √2 is an *irrational number*. That means it has an infinite fractional expansion in *any* number base, and Math.sqrt cannot be expected to produce an exact answer.

In general

```
if ( <double or float> == <another double or float> )..
```

will often not work as expected. Instead be happy with a sufficient level of accuracy:

```
    double root2 = Math.sqrt(2);
    if (Math.abs(root2 * root2 - 2.0) < 1.0E-6) {
      System.out.println("Right");
    } else {
      System.out.println("Wrong!!");
    }
```

use Math.abs because you do not know if it is too big or too small.