# 3.1 Supplement Notes on Floating Point Numbers

Ordinary binary integers cover a relatively limited range of values about ±32,767 for 16-bit and ±2,147,483,647 for 32 bits. Many real-world problems cover a wider range of values than this, from very small values to very large values. These “scientific” values are represented in most computers by “real” or “floating point” numbers. Many calculations with only addition and subtraction can be done with integers. Multiplication quickly outgrows the integer range (overflow), while division almost always generates either a remainder or a number with a repeating fraction. Thus real values or floating point numbers arise quite easily in many problems.

## 3.1.1 Scientific Notation

The representation is similar to “scientific” or “exponent” notation for numbers, which also allows very large and very small numbers to be written easily. For example, the velocity of light is 299,792,458 $ms^{-1}$ , or $2.99792458*10^{8}$ $ms^{-1}$ . For most practical purposes we can use the approximate values 300,000,000 $ms^{-1}$ , or $3*10^{8}$ $ms^{-1}$ . The decimal point is normally to the right of the last digit (the 8). To convert to scientific, the point is first shifted so that it follows the first digit, a shift of 8 places. To correct for this shift, a multiplier of $10^{8}$ is included. For very large and very small numbers the scientific notation is very efficient. Thus the charge on the electron is about 0.00000000000000000016021892 C or, even better, $1.6021892*10^{-19}$ C. With the first form there is easily trouble counting zeros. Similarly, the number of atoms in one mole of gas is about 602204500000000000000000 or $6.022045*10^{23}$ . The scientific form makes things easier in several ways —

1. The exponent tells very quickly how large the number is (+ve exponent) or how small (-ve exponent). It also reduces the problems in counting many following or preceding zeros (I hope the counting was correct in the two examples!).
2. The number of digits tells how accurately the value is known. Thus a value of 98270 known to an accuracy of ±10 could be written $9.827*10^{4}$ (the unit digit is not certain), whereas if the last digit is certain, it would be written as $9.8270*10^{4}$ . Writing the speed of light as $3*10^{8}$ $ms^{-1}$ means that we worry about only that first digit, whereas writing it as $3.00*10^{8}$ $ms^{-1}$ means that the first 3 digits are correct. (Writing to another digit must be $2.998*10^{8}$ $ms^{-1}$ when the last digit is rounded.)

Computer real numbers are held in a similar way, except that all values are normally binary.

A real value is held as two parts –

1. The significand or mantissa is usually about 24 or 50 bits and usually gives a value 0.5 <= V <=1.0, with the binary point at or near the left-most bit.
2. The exponent or characteristic is a smaller 8 or 12 bit value which gives a multiplier for the significand. For most numbers the value for a significand S and an exponent E, is $S*2^{E}$ . The value could be written as a binary value with integral and fractional parts; the exponent tells by how many bits the binary point must be shifted.

For example, $0.022045 * 10^{23}$ ← exponent

↑ Mantissa

## 3.1.2 Normalisation

The value $2.99792458*10^{8}$ could be written as $0.299792458*10^{9}$, perhaps $0.00299792458*10^{11}$, or $29.9792458*10^{7}$ — all are equivalent. By convention, scientific numbers are always written with one digit before the point, giving a “normalised” representation. Binary floating point numbers are similarly *normalised* to the binary value 0.1…, or 1. *xx*… by balancing a left shift of the digits with a decrease in the exponent (or a right shift of the digits with an increase in the exponent).

**Examples:**

Decimal: 12.34 => $1.234 * 10^{1}$
Decimal: 0.034 => $0.34 * 10^{-1}$

Since it is difficult to write exponents with superscipts in a text editor, programming languages need an alternative for writing numbers in scientific notation. Most programming languages use an e or E for Exponent to do this. So we would write $6.02 \times 10^{23}$ as 6.02e23 and $6.63 \times 10^{-34}$ as 6.63e-34.

## 3.1.3 Multiplication & Division

If you like, you can think of the exponent as specifying the number of places the decimal point was moved. $6.02 \times 10^{23}$ has its decimal point shifted left 23 places. A negative exponent just means that the decimal point was shifted to the other way, so $6.63 \times 10^{-34}$ had its exponent shifted to the right 34 places. Use positive exponents for big numbers, negative exponents for numbers very close to zero.

Scientific notation has the nice property that it is easy to use in multiplication and division. When you multiply two numbers in scientific notation, multiply the mantissas and add the exponents.

**Example 37:**

$6.02 \text{ x } 10^{23} * 6.63 \text{ x } 10^{-34}$
$= (6.02 * 6.63) \text{ x } 10^{23 + -34}$
$= 39.9126 \text{ x } 10^{-11}$

To divide, divide the mantissas and subtract the exponents.

**Example 38:**

$6.02 \text{ x } 10^{23} / 6.63 \text{ x } 10^{-34}$
$= (6.02 / 6.63) \text{ x } 10^{23 - -34}$
$= 0.908 \text{ x } 10^{57}$

## 3.2 IEEE 754 Floating Point Representation

Most computer manufacturers used to develop their own floating point representations for their own computers. Not only were they different, but also many had serious design errors. The IEEE 754 standard attempts to overcome these problems and has been adopted in most modern computers.

IEEE floating point numbers come in two sizes, four-byte single precision and eight-byte double precision numbers. The layouts for the parts of a floating point number are:

### 3.2.1 Single Precision (Bit Layout: sxxx xxxx xfff ffff ffff ffff ffff ffff)

The IEEE 754 standard defines several number formats and precisions. The 32 bit format has a 1-bit sign, an 8-bit exponent with a bias of 127, and a 23-bit significand. The significand is always stored in “normalised” form with its most significant bit “1”. As this bit is always a 1, it is redundant and can be omitted from the stored number and automatically inserted in the arithmetic unit when calculations are to be done. The bits are used as — sxxx xxxx xfff ffff ffff ffff ffff ffff where s is the sign bit, x…x are the exponent bits and f…f the significand bits. The value of a number is then

$$(-1)^{sign} * (1.0 + significand) * 2^{(exponent-127)}$$

<u>Sign Bit:</u>
A sign bit of zero indicates a positive number and a sign bit of one indicates a negative number. The **mantissa** is always interpreted as a positive base-two number (unsigned). It is not a twos-complement number. If the sign bit is one, the floating-point value is negative, but the mantissa is still interpreted as a positive number that must be multiplied by -1.

<u>Exponent Bits:</u>
The exponent field is interpreted in one of three ways.

- An exponent of all ones indicates the floating-point number has one of the special values of plus or minus infinity, or "not a number" (NaN). NaN is the result of certain operations, such as the division of zero by zero.
- An exponent of all zeros indicates a denormalized floating-point number.
- Any other exponent indicates a normalized floating-point number.

Exponents that are neither all ones nor all zeros indicate the power of two by which to multiply the normalized mantissa. The power of two can be determined by interpreting the exponent bits as a positive number, and then subtracting a bias from the positive number. For a float, the bias is 127.

<u>Mantissa Bits:</u>
The mantissa contains one extra bit of precision beyond those that appear in the mantissa bits. **(1.0+*mantissa*)** The mantissa of a float, which occupies only 23 bits, has 24 bits of precision. The mantissa of a double, which occupies 52 bits, has 53 bits of precision. The exponent of floating-point numbers indicates whether or not the number is normalized. If the exponent is all zeros, the floating-point number is denormalized and the most significant bit of the mantissa is known to be a zero. Otherwise, the floating-point number is normalized and the most significant bit of the mantissa is known to be one.

**Example 39:**

0 01111101 101 0000 0000 0000 0000 0000
- ➔ Sign bit = 0
- ➔ Exponent bit = 01111101 = 125
- ➔ Mantissa bit = 1010…0 = 0.5 + 0.125 = 0.625

$(-1)^0 * (1.0 + 0.625) * 2^{(125-127)} = (1.625) * 2^{-2} = \mathbf{0.40625}$

An exponent field in a float of 01111101 yields a power of two by subtracting the bias (127) from the exponent field interpreted as a positive integer (125). The power of two, therefore, is 125 - 127, which is -2. Mantissa (Significand) values are 101 0000 0000 0000 0000 0000. The answer is 0.625. Therefore the final answer is (1.0 + 0.625) multiply two to the power of (-2) and it is equal to 0.40625.

**Special Numbers:**

The IEEE 754 standard has quite complicated rules on the rounding of numbers. It also has ways of representing underflowed and overflowed numbers and special error values called "Not a Number" (NaN), from cases like 0/0. Normalisation is also rather more complicated than is described here, to handle a "gradual underflow".

NaN
An exponent of all ones with any other mantissa is interpreted to mean "not a number".
The JVM always produces the same mantissa for NaN, which is all zeros except for the most significant mantissa bit that appears in the number (1 11111111 10000000000000000000000)

Infinity:
An exponent of all ones with a mantissa whose bits are all zero indicates infinity. The sign of the infinity is indicated by the sign bit.

Denormalized
An exponent of all zeros indicates the mantissa is denormalized, which means the unstated leading bit is a zero instead of a one.

**Denormalized float values**

| Value | Float bits (sign exponent mantissa) |
|---|---|
| Smallest positive (non-zero) float | 0 00000000 00000000000000000000001 |
| Smallest negative (non-zero) float | 1 00000000 00000000000000000000001 |
| Largest denormalized float | 0 00000000 11111111111111111111111 |
| Positive zero | 0 00000000 00000000000000000000000 |
| Negative zero | 1 00000000 00000000000000000000000 |

**Table: Single Precision (Reference only):**

| **Range Name** | **S**<br>**1** | **E**<br>**8** | **M**<br>**23** | Hexadecimal Range | **Range** | **Decimal Range** § |
|---|---|---|---|---|---|---|
| Quiet<br>-NaN | 1 | 11..11 | 11..11<br>:<br>10..01 | FFFFFFFF<br>:<br>FFC00001 | | |
| Indeterminate | 1 | 11..11 | 10..00 | FFC00000 | | |
| Signaling<br>-NaN | 1 | 11..11 | 01..11<br>:<br>00..01 | FFBFFFFF<br>:<br>FF800001 | | |
| -Infinity<br>(Negative Overflow) | 1 | 11..11 | 00..00 | FF800000 | <<br>$-(2-2^{-23}) \times 2^{127}$ | < -3.4028235677973365E+38 |
| **Negative Normalized**<br>**$-1.m \times 2^{(e-127)}$** | 1 | 11..10<br>:<br>00..01 | 11..11<br>:<br>00..00 | FF7FFFFF<br>:<br>80800000 | $-(2-2^{-23}) \times 2^{127}$<br>:<br>$-2^{-126}$ | -3.4028234663852886E+38<br>:<br>-1.1754943508222875E-38 |
| **Negative Denormalized**<br>**$-0.m \times 2^{(-126)}$** | 1 | 00..00 | 11..11<br>:<br>00..01 | 807FFFFF<br>:<br>80000001 | $-(1-2^{-23}) \times 2^{-126}$<br>:<br>$-2^{-149}$<br>$(-(1+2^{-52}) \times 2^{-150})$ * | -1.1754942106924411E-38<br>:<br>-1.4012984643248170E-45<br>(-7.0064923216240862E-46) * |
| Negative Underflow | 1 | 00..00 | 00..00 | 80000000 | $-2^{-150}$<br>:<br>< -0 | -7.0064923216240861E-46<br>:<br>< -0 |
| **-0** | 1 | 00..00 | 00..00 | 80000000 | -0 | -0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **+0** | 0 | 00..00 | 00..00 | 00000000 | 0 | 0 |
| Positive Underflow | 0 | 00..00 | 00..00 | 00000000 | > 0<br>:<br>$2^{-150}$ | > 0<br>:<br>7.0064923216240861E-46 |
| **Positive Denormalized**<br>**0.*m* × 2$^{(-126)}$** | 0 | 00..00 | 00..01<br>:<br>11..11 | 00000001<br>:<br>007FFFFF | $((1+2^{-52}) \times 2^{-150})$ *<br>$2^{-149}$<br>:<br>$(1-2^{-23}) \times 2^{-126}$ | (7.0064923216240862E-46) *<br>1.4012984643248170E-45<br>:<br>1.1754942106924411E-38 |
| **Positive Normalized**<br>**1.*m* × 2$^{(e-127)}$** | 0 | 00..01<br>:<br>11..10 | 00..00<br>:<br>11..11 | 00800000<br>:<br>7F7FFFFF | $2^{-126}$<br>:<br>$(2-2^{-23}) \times 2^{127}$ | 1.1754943508222875E-38<br>:<br>3.4028234663852886E+38 |
| +Infinity<br>(Positive Overflow) | 0 | 11..11 | 00..00 | 7F800000 | > $(2-2^{-23}) \times 2^{127}$ | > 3.4028235677973365E+38 |
| Signaling<br>+NaN | 0 | 11..11 | 00..01<br>:<br>01..11 | 7F800001<br>:<br>7FBFFFFF | | |
| Quiet<br>+NaN | 0 | 11..11 | 10..00<br>:<br>11..11 | 7FC00000<br>:<br>7FFFFFFF | | |

**Exercise:**

What is the decimal value represented by the following numbers?

```
a) 1 00000000 00000000000000000000000
b) 1 11111111 11111111111111111111111
c) 0 11111111 00000000000000000000000
```

## 3.2.2 Double precision numbers (**Bit Layout: <u>sxxx xxxx xxxx ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff</u>)**

As described earlier, the 32-bit representation is barely adequate for serious computation; the precision is limited and "rounding" errors accumulate very quickly. Some computations lasting only a second or two can become quite meaningless. Also, the number range of $10^{\pm 38}$ is too small to handle some physical quantities, or formula involving them. The 754 standard therefore includes a 64- bit representation to overcome these problems.

In Java 32-bit quantities are of type float, while 64-bit are double. Floating point results are normally produced with type double and a cast is necessary to store into a float variable. IEEE 754 double precision uses a 52-bit significand (giving about 16 decimal digits of precision) and an 11-bit exponent with a bias of 1023 (a range of about $10^{\pm 300}$). The underlying principles are as for the 32-bit representation.

$$(-1)^{sign} * (1.0 + significand) * 2^{(exponent-1023)}$$

**Example 40:**

```
0 01111111101 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
    ➔ Sign bit = 0
    ➔ Exponent bit = 01111111101 = 1021
    ➔ Mantissa bit = 1010…0 = 0.5 + 0.125 = 0.625
```
$(-1)^0 * (1.0 + 0.625) * 2^{(1021-1023)} = (1.625) * 2^{-2} =$ **0.40625**

An exponent field in a float of 01111111101 yields a power of two by subtracting the bias (1023) from the exponent field interpreted as a positive integer (1021). The power of two, therefore, is 1021 - 1023, which is -2. Mantissa (Significand) values are 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. The answer is 0.625. Therefore the final answer is (1.0 + 0.625) multiply two to the power of (-2) and it is equal to 0.40625.

**Table: Double Precision (for reference only):**

| Range Name | S<br>1 | E<br>11 | M<br>52 | Hexadecimal Range | Range | Decimal Range § |
|---|---|---|---|---|---|---|
| Quiet<br>-NaN | 1 | 11..11 | 11..11<br>:<br>10..01 | FFFFFFFFFFFFFFFF<br>:<br>FFF8000000000001 | | |
| Indeterminate | 1 | 11..11 | 10..00 | FFF8000000000000 | | |
| Signaling<br>-NaN | 1 | 11..11 | 01..11<br>:<br>00..01 | FFF7FFFFFFFFFFFF<br>:<br>FFF0000000000001 | | |
| -Infinity<br>(Negative Overflow) | 1 | 11..11 | 00..00 | FFF0000000000000 | $< -(2-2^{-52}) \times 2^{1023}$ | -1.7976931348623158E+308 |
| **Negative Normalized**<br>**$-1.m \times 2^{(e-1023)}$** | 1 | 11..10<br>:<br>00..01 | 11..11<br>:<br>00..00 | FFEFFFFFFFFFFFFF<br>:<br>8010000000000000 | $-(2-2^{-52}) \times 2^{1023}$<br>:<br>$-2^{-1022}$ | -1.7976931348623157E+308<br>:<br>-2.2250738585072014E-308 |
| **Negative Denormalized**<br>**$-0.m \times 2^{(-1022)}$** | 1 | 00..00 | 11..11<br>:<br>00..01 | 800FFFFFFFFFFFFF<br>:<br>8000000000000001 | $-(1-2^{-52}) \times 2^{-1022}$<br>:<br>$-2^{-1074}$<br>$(-(1+2^{-52}) \times 2^{-1075})$ * | -2.2250738585072010E-308<br>:<br>-4.9406564584124654E-324<br>(-2.4703282292062328E-324) * |
| Negative Underflow | 1 | 00..00 | 00..00 | 8000000000000000 | $-2^{-1075}$<br>:<br>< -0 | -2.4703282292062327E-324<br>:<br>< -0 |
| **-0** | 1 | 00..00 | 00..00 | 8000000000000000 | -0 | -0 |
| **+0** | 0 | 00..00 | 00..00 | 0000000000000000 | 0 | 0 |
| Positive Underflow | 0 | 00..00 | 00..00 | 0000000000000000 | > 0<br>:<br>$2^{-1075}$ | > 0<br>:<br>2.4703282292062327E-324 |
| **Positive Denormalized**<br>**$0.m \times 2^{(-1022)}$** | 0 | 00..00 | 00..01<br>:<br>11..11 | 0000000000000001<br>:<br>000FFFFFFFFFFFFF | $((1+2^{-52}) \times 2^{-1075})$ *<br>$2^{-1074}$<br>:<br>$(1-2^{-52}) \times 2^{-1022}$ | (2.4703282292062328E-324) *<br>4.9406564584124654E-324<br>:<br>2.2250738585072010E-308 |
| **Positive Normalized**<br>**$1.m \times 2^{(e-1023)}$** | 0 | 00..01<br>:<br>11..10 | 00..00<br>:<br>11..11 | 0010000000000000<br>:<br>7FEFFFFFFFFFFFFF | $2^{-1022}$<br>:<br>$(2-2^{-52}) \times 2^{1023}$ | 2.2250738585072014E-308<br>:<br>1.7976931348623157E+308 |
| +Infinity<br>(Positive Overflow) | 0 | 11..11 | 00..00 | 7FF0000000000000 | $> (2-2^{-52}) \times 2^{1023}$ | 1.7976931348623158E+308 |
| Signaling<br>+NaN | 0 | 11..11 | 00..01<br>:<br>01..11 | 7FF0000000000001<br>:<br>7FF7FFFFFFFFFFFF | | |
| Quiet<br>+NaN | 0 | 11..11 | 10..00<br>:<br>11..11 | 7FF8000000000000<br>:<br>7FFFFFFFFFFFFFFF | | |

§ Your least significant digits may differ.

## 3.2.3 Converting from IEEE 754 Floating Point Representation to Decimal

**Example 41:**

$40900000_{16}$ = 0100 0000 1001 0000 … 0000

- ➔ Sign bit = 0
- ➔ Exponent bit = 100 0000 1 = 129

➔ Mantissa bit = 001 0000 … 0000 = 0.125
Answer = $(-1)^0 * (1.0 + 0.125) * 2^{(129-127)} = (1.125) * 2^2 = 4.5$

Exercise:
Convert $C2100000_{16}$ from IEEE 754 Floating Point (Single Precision) to decimal

### 3.2.4 Convert from Decimal to IEEE 754 Floating Point Rrepresentation

**Example 42:**

**(1)** $-1.25_{10}$, white the number in binary format: $-1.01_2$
- ➔ $1.01_2$ is already in normalized format, so don't need to do any shifting
- ➔ IEEE Sign bit = negative = 1
- ➔ IEEE Significand bit => 0.01… = 0.01...0
- ➔ Exponent = 127 = $01111111_2$

Answer: 1 01111111 0100…0 = $BFA00000_{16}$

**(2)** $0.15625_{10}$, write the number in binary format: $0.00101_2$
- ➔ Normalize => Shift point to the right for three places= $1.01_2 \times 2^{-3}$
- ➔ IEEE Sign bit = positive = 0
- ➔ IEEE Significand bit => 0.01 = 0.010...0
- ➔ Exponent = 127 + three places right shift = 127 + (-3) = 124 = $01111100_2$

Answer: 0 01111100 0100000…0 = $3E200000_{16}$

**(3)** $19.5_{10}$, white the number in binary format: 10011.100…
- ➔ Normalise => Shift point to the left for four places = $1.0011100… \times 2^4$
- ➔ IEEE Sign bit = positive = 0
- ➔ IEEE Significand bit => 0.00111000 = 0.00111...0
- ➔ Exponent = 127 + four places left shift = 127 + 4 = 131 = $10000011_2$

Answer: 0 10000011 0011100000…0

Then, what is the value of $4.875_{10}$?
- ➔ $19.5_{10}$ = 0 10000011 0011100000…0 from the above calculation
- ➔ And $4.875_{10} = 19.5 / 4 = 19.5 * \mathbf{2^{-2}}$
- ➔ Sign Bit: unchanged = 0
- ➔ Significand bit: unchanged = 001110000…0
- ➔ Exponent bit = exponent of part (a) + -2 = $10000001_2 - 2_{10} = 10000001_2$

Answer is 0 10000001 00111000…0 = $409C0000_{16}$

$6.5_{10}$, white the number in binary format: 110.100…
- ➔ Normalise => Shift point to the left for two places = $1.10100… \times 2^2$
- ➔ IEEE Sign bit = positive = 0
- ➔ IEEE Significand bit => 0.101000… = 0.1010...0
- ➔ Exponent = 127 + two places left shift = 127 + 2 = 129 = $10000001_2$

Answer: 0 10000001 10100000…0

Then, what is the value of $52_{10}$?
- ➔ $6.5_{10}$ = 0 10000001 10100000…0 from the above calculation
- ➔ And $52_{10} = 6.5 * \mathbf{8} = 6.5 * \mathbf{2^3}$
- ➔ Sign Bit: unchanged = 0
- ➔ Significand bit: unchanged = 10100000…0
- ➔ Exponent bit = exponent of part (a) + 3 = $10000001_2 + 3_{10} = 10000100_2$

Answer is 0 10000100 10100000…0 = $42500000_{16}$

Then, what is the value of $3.25_{10}$?
- ➔ And $3.25_{10} = 6.5 / 2 = 6.5 * 2^{-1}$
- ➔ Sign Bit: unchanged = 0
- ➔ Significand bit: unchanged = 10100000…0
- ➔ Exponent bit = exponent of part (a) + (-1) = $10000001_2 - 1_{10} = 10000000_2$

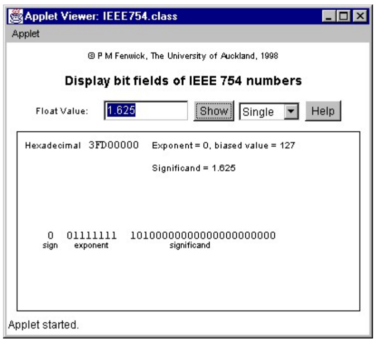
Answer is 0 10000000 10100000…0 =$40500000_{16}$

**Exercises:**

Convert from Decimal to IEEE 754 Floating Point (Single Precision)
**(1)** 2.25

| |
|---|
| **(2)** 4.5 |

### 3.2.5 Applet




### 3.2.6 IEEE 754 Floating Point in JAVA

Java provides both single (32-bit) and double (64-bit) floating point types, with the default being double.

**Example 43:**

```
public class IEEETestFloat {
  public static void main (String args[]) {
    float f = 4.5F;
    System.out.println(Integer.toHexString(Float.floatToIntBits (f)));

    f = Float.intBitsToFloat(0xbfa00000);
    System.out.println(f);
  }
}
```

**Output:**

```
> java IEEETestFloat
40900000
-1.25
```

**Methods:**

| | |
|---|---|
| public static int **floatToIntBits**(float value) | Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout. |
| public static int **floatToRawIntBits**(float value) | Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout, preserving Not-a-Number (NaN) values. |

**Example 44:**

```
public class IEEETestDouble {
  public static void main (String args[]) {
    double d = 4.5;
     System.out.println(Long.toHexString(Double.doubleToLongBits (d)));

    d = Double.longBitsToDouble(0x4012000000000000L);
    System.out.println(d);
  }
}
```

**Output:**

```
> java IEEETestDouble
4012000000000000
4.5
```

**Methods:**

| | |
|---|---|
| public static long **doubleToLongBits**(double value) | Returns the actual bit pattern of the IEEE 754 representation of the value. (It is useful for analysing or "dismantling" a floating point number.) |
| public static double **longBitsToDouble**(long bits) | takes the bit pattern in bits and returns it as a double value. (It is useful for building a floating point number.) |

### 3.2.7 Extra Note:

Internally, most computers use a base of 2 (i.e. the fraction is multiplied by 2 *exponent* ), or less often 16 or 8. The significand is usually held in sign & magnitude form, with the exponent in say excess 127. A 0.0 value is an all-0 word. Important points to remember are

- Do not confuse the *range* and the *precision* of floating point numbers.
- The *range* is determined by the exponent and determines how close to zero or far from zero a number may be. It is closely connected to the exponent form of scientific notation. An 8-bit signed exponent can have values from -128 to +127 (–126 to +127 in IEEE 754 floating point). The smallest representable number will be about $2^{-128}$ and the largest $2^{+127}$ . Remembering that $\log_2 10$ is closed to 1/0.3 (page 5 of booklet), the number range is about $10^{-38}$ and the largest $10^{+38}$ . It is shown later that this range is quite inadequate for some calculations.
- The *precision* is governed by the significand (or fraction or mantissa) and gives the accuracy with which a number may be represented. Remember that *N* bits equals about 0.3**N* decimal digits. A standard "32-bit real" has 23-bit precision, or not quite 7 decimal digits. A 64-bit "double" has 52-bit or 15 decimal digits. Even a 32 bit real can handle the accuracy of most physical measurements, but much of the precision is lost by rounding in lengthy calculations; this is the real justification for using 64- bit or 128-bit floating point numbers.

Floating point arithmetic is subject to rounding and truncation errors. The significand can represent only so many bits; any less significant bits must be discarded. Often, if the first discarded bit is a 1, we add 1 onto the significand to "round" the result. Thus 1.7 would round to 2, which is probably a better result than 1 (from just forgetting the bits).

Care is needed when using real-number arithmetic. Some of the problems seem to disappear with "long" numbers, but really stay there and are never more than reduced.

- The 32 bit floating point "real" on many computers is quite limited in comparison with many scientific calculators. Its range is about $10^{\pm 38}$ , and its precision is not quite 7 decimal digits. Even short calculation sequences can overwhelm it.
- Arithmetic with floating point numbers is seldom exact and great care must be taken in long calculation sequences as "round-off" errors accumulate. For example a solution of a set of 40 simultaneous equations had the 3–4 least significant decimal digits quite meaningless.
- Beware of mathematical techniques which involve differences of large quantities. This is related to the previous point. Say we have two values close to 1000, both with the last decimal digit uncertain, such as 102*x* and 99*x* (both known to about 10 parts in 1000, or 1%). Subtracting gives a value 3*x*, where the last digit is still uncertain, but the error is now 10 parts in 30, or about 30%. Two moderately accurate values have combined to give a value which is nearly meaningless. Some types of statistical calculation are especially sensitive to this problem.
- The result is that floating point arithmetic is not exact. Because of possible disappearance of low-order bits we cannot guarantee that (A+B)+C = A+(B+C). In most cases it is very nearly true, if we are careful, but in extreme cases it is anything but true.
- Be very careful if using floating point arithmetic for financial calculations. Rounding errors may make it almost impossible to achieve reliable balances, especially if the number precision is barely adequate to represent the whole amount.