Java floating point numbers review

Nasser M. Abbasi

Nov 15, 2000 Compiled on January 29, 2024 at 3:00am

Contents

1 Java primitive types sizes

type	size in bytes
byte	1
short	2
int	4
long	8
float	4 (IEEE 754)
double	8 (IEEE 754)

2 Maximum value in signed and unsigned integers

Signed integer table

number of bits	Java type	range	range in base 10
8	*byte*	$2^7 - 1 \ldots - 2^7$	$127 \ldots - 128$
16	*short*	$2^{15} - 1 \ldots - 2^{15}$	$32,767 \ldots - 32,768$
32	*int*	$2^{31} - 1 \ldots - 2^{31}$	$2,147,483,647 \ldots - 2,147,483,648$
64	*long*	$2^{63} - 1 \ldots - 2^{63}$	$9,223,372,036,854,775,807 \ldots - 9,223,372,036,854,7$

number of bits	Java type	range	range in HEX
8	*byte*	$2^7 - 1 \ldots - 2^7$	7F $\ldots$ − 80
16	*short*	$2^{15} - 1 \ldots - 2^{15}$	7F FF $\ldots$ − 8000
32	*int*	$2^{31} - 1 \ldots - 2^{31}$	7F FF FF FF $\ldots$ − 80000000
64	*long*	$2^{63} - 1 \ldots - 2^{63}$	7F FF FF FF FF FF FF FF $\ldots$ − 8000000000000000

Unsigned integer table

number of bits	Java type	range	range in base 10
8	*byte*	$2^8 - 1 \ldots 0$	$255 \ldots 0$
16	*short*	$2^{16} - 1 \ldots 0$	$65,535 \ldots 0$
32	*int*	$2^{32} - 1 \ldots 0$	$4,294,967,295 \ldots 0$
64	*long*	$2^{64} - 1 \ldots 0$	$18,446,744,073,709,551,615 \ldots 0$

number of bits	Java type	range	range in HEX
8	*byte*	$2^8 - 1 \ldots 0$	FF $\ldots$ 00
16	*short*	$2^{16} - 1 \ldots 0$	FF FF $\ldots$ 0000
32	*int*	$2^{32} - 1 \ldots 0$	FF FF FF FF $\ldots$ 00000000
64	*long*	$2^{64} - 1 \ldots 0$	FF FF FF FF FF FF FF FF $\ldots$ 0000000000000000

3 Some bits table

The max value that can be obtained using n bits is found by using the formula $2^n - 1$, this assume unsignd values.

bit pattern	base 10	Hex
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
1 0000	16	10
1 0001	17	11
1 0010	18	12
1 0011	19	13
1 0100	20	14
1 0101	21	15
1 0110	22	16
1 0111	23	17
1 1000	24	18
1 1001	25	19
1 1010	26	1A
1 1011	27	1B
1 1100	28	1C
1 1101	29	1D
1 1110	30	1E
1 1111	31	1F
10 0000	32	20
0111 1111	127	7F
10000000	128	80
11111111	255	FF
1 00000000	256	1 00
1111 11111111	4,095	F FF
11111111 11111111	65,535	FF FF
1111 11111111 11111111	1,048,575	F FF FF
11111111 11111111 11111111	16,777,215	FF FF FF

So, 16 bits needs 5 digits in base 10 to represent it.
32 bits needs 10 digits in base 10 to represent it.
64 bits needs 20 digits in base 10 to represent it.

So, it looks like the number of digits in base 10 to represent a bit pattern of length n is $(1/3)n$
So 128 bits will require about 42 digits in base 10 to represent externally.

4 Power of 2 table

power of two	base 2	base 10	Hex
2^0	1	1	1
2^1	01	2	2
2^2	100	4	4
2^3	1000	8	8
2^4	1 0000	16	10
2^5	10 0000	32	20
2^6	100 0000	64	40
2^7	1000 0000	128	80
2^8	1 0000 0000	256	1 00
2^9	10 0000 0000	512	2 00
2^{10}	...	(1K) 1,024	4 00
2^{11}		2,048	8 00
2^{12}		4,096	10 00
2^{13}		8,192	20 00
2^{14}		16,384	40 00
2^{15}		32,768	80 00
2^{16}		65,536	1 00 00
2^{17}		131,072	2 00 00
2^{18}		262,144	4 00 00
2^{19}		524,288	8 00 00
2^{20}		(1 MB) 1,048,576	10 00 00
2^{21}		2,097,152	20 00 00
2^{22}		4,194,304	40 00 00
2^{23}		8,388,608	80 00 00
2^{24}		16,777,216	1 00 00 00
2^{25}		33,554,432	2 00 00 00
2^{26}		67,108,864	4 00 00 00
2^{27}		134,217,728	8 00 00 00
2^{28}		268,435,456	10 00 00 00
2^{29}		536,870,912	20 00 00 00
2^{30}		(1 GB) 1,073,741,824	40 00 00 00
2^{31}		2,147,483,648	80 00 00 00
2^{32}		4,294,967,296	1 00 00 00 00
2^{33}		8,589,934,592	2 00 00 00 00
2^{34}		517,179,869,184	4 00 00 00 00
2^{35}		34,359,738,368	8 00 00 00 00
2^{36}		68,719,476,736	10 00 00 00 00

power of two	base 2	base 10	Hex
2^{47}	100000...	140,737,488,355,328	80 00 00 00 00 00
2^{48}		281,474,976,710,656	1 00 00 00 00 00 00
2^{49}		562,949,953,421,312	2 00 00 00 00 00 00
2^{50}		1,125,899,906,842,624	4 00 00 00 00 00 00
2^{51}		2,251,799,813,685,248	8 00 00 00 00 00 00
2^{52}		4,503,599,627,370,496	10 00 00 00 00 00 00
2^{53}		9,007,199,254,740,992	20 00 00 00 00 00 00
2^{54}		18,014,398,509,481,984	40 00 00 00 00 00 00
2^{55}		36,028,797,018,963,968	80 00 00 00 00 00 00
2^{56}		72,057,594,037,927,936	1 00 00 00 00 00 00 00
2^{57}		144,115,188,075,855,872	2 00 00 00 00 00 00 00
2^{58}		288,230,376,151,711,744	4 00 00 00 00 00 00 00
2^{59}		576,460,752,303,423,488	8 00 00 00 00 00 00 00
2^{60}		1,152,921,504,606,846,976	10 00 00 00 00 00 00 00
2^{61}		2,305,843,009,213,693,952	20 00 00 00 00 00 00 00
2^{62}		4,611,686,018,427,387,904	40 00 00 00 00 00 00 00
2^{63}		9,223,372,036,854,775,808	80 00 00 00 00 00 00 00
2^{64}		18,446,744,073,709,551,616	1 00 00 00 00 00 00 00 00

5 Float and Double in Java

Java uses IEEE 754.

A number such as 0.125 is expressed as $1.25 \cdot 10^{-1}$ or $1 \cdot 2^{-3}$.

In floating point, the second form above is used. i.e. base 2 is used for the exponent.

The sign uses 1 bit. 0 for positive and 1 for negative. The exponent uses the next 8 bits (biased by 127), and the exponent uses the next 23 bits.

In Java, a float uses IEEE 754. The following explains how float and double represented in Java.

$$
\begin{aligned}
s \cdot m \cdot 2^{E-N+1} \\
s \text{ is the sign, and can be} -1 \text{or} +1 \\
1 \leq m \leq 2^{24} - 1 = 16,777,215 \\
-126 \leq E \leq +127 \\
N = 24
\end{aligned}
$$

So, from the above, a float f in IEEE 754 is in the range

$$
\begin{aligned}
-1 \cdot 16777215 \cdot 2^{-126-24+1} \leq f \leq +1 \cdot 16777215 \cdot 2^{127-24+1} \\
-16777215 \cdot 2^{-149} \leq f \leq +16777215 \cdot 2^{104} \\
-2.35 \cdot 10^{-38} \leq f \leq 3.4 \cdot 10^{38}
\end{aligned}
$$

In Java a double is expressed as

$$
\begin{aligned}
s \cdot m \cdot 2^{E-N+1} \\
s \text{ is the sign, and can be} -1 \text{or} +1 \\
1 \leq m \leq 2^{53} - 1 = 9,007,199,254,740,991 \\
-1022 \leq E \leq +1023 \\
N = 24
\end{aligned}
$$

So, from the above, a double f in IEEE 754 is in the range

$$
\begin{aligned}
-1 \cdot 9007199254740991 \cdot 2^{-1022-24+1} \leq f \leq +1 \cdot 9007199254740991 \cdot 2^{1023-24+1} \\
-9007199254740991 \cdot 2^{-1045} \leq f \leq +9007199254740991 \cdot 2^{1000} \\
-2.2 \cdot 10^{-308} \leq f \leq 1.8 \cdot 10^{308}
\end{aligned}
$$

5.1 How to read a floating point?

Given this example:

11000011100101100000000000000000

The above is binary representation of single precision floating point (32 bit).

Reading from the left most bit (bit 31) to the right most bit (bit 0).

bit 31 is 1, so this is a negative number. bits 30 ... 23 is the exponent, which is 10000111 or 135. But since the exponent is biased by 127, it is actually 8, so now we have the

exponent part which is 2^8. Next is bits $22 \ldots 0$, which is 00101100000000000000000, since there is an implied 1, the above can be re-written as 1.00101100000000000000000, which is read as follows:

$1 + 0(1/2) + 0(1/4) + 1(1/8) + 0(1/16) + 1(1/32) + 1(1/64) + 0(1/128) + 0(1/256) + \ldots allzeros$

which is $1 + (1/8) + (1/32) + (1/64) = 1 + (11/64) = 75/64$

Hence the final number is $-(75/64) \cdot 2^8 = -(75/64) \cdot 256 = -300$.

The above implies that a number that be can't be expressed as sum of power of 2, can't be represented exactly in a floating point. Since a float is represented as $m \cdot 2^e$, assume $e = 0$, then the accuracy of a float goes like this: $1, 1 + (1/2), 1 + (1/2) + (1/4), 1 + (1/2) + (1/4) + (1/8), 1 + (1/2) + (1/4) + (1/8) + (1/16), \ldots$ or $1, 1.5, 1.75, 1.87, \ldots$,

So, a number such as 1.4 can't be exactly expressed in floating point ! because the .4 value can't be expressed as a sum of power of 2.

The greatest number that has an exact IEEE single-precision representation is 340282346638528859811704 $(2^{128} - 2^{104})$, This is 40 digits number, which is represented by 01111111011111111111111111111111

6 References

The Java programing language specifications.

`http://www.math.grin.edu/~stone/courses/fundamentals/IEEE-reals.html`