

1 Arithmetic and Precision

There are five *primitive* data types in Java

- **boolean** : This kind of variable takes on values **true** and **false**.
- **char** : In most previous programming languages, this type of variable would take one byte of space for each variable needed. Java is more modern than that, however, and in order to accommodate all the languages in which text will exist, Java uses *two* bytes of space for each **char** variable and stores the *Unicode* value for the character that the variable holds. The Unicode value is a multi-language code that will allow for not just the usual Roman alphabet but also for all the diacriticals and such that are needed to render nearly any common human language into computer-storable format.

One byte is 8 bits, so with two bytes there are $2^{16} - 1 = 65536$ different bit patterns. Each of these bit patterns can be viewed as a positive integer in the range 0 through 65535. The one-byte integer values for standard English and the other “characters” represented in the ASCII character set are shown on page 795 of your text. You will note that the ASCII character set has only the basic Roman alphabetic characters together with some control codes, numerals, etc.

If you ever wind up dealing with foreign languages, diacriticals, or even ideographic languages like Chinese, Japanese, or Korean, you will have to think about Unicode representations of characters. We don’t need to worry about that in this course. If you are interested in Unicode, just google “unicode” and the references will pop up immediately.

- **int** : The **int** variables are for storing whole-number integer values, positive and negative.
- **float** : This is the 32-bit analog of the 64-bit precision of the **double**. We won’t do much with these, because the only real reason for using them is to save memory (half as much as for a **double**), and memory these days is cheap. Besides, virtually all scientific computing uses “double precision floating point” as the standard.

The only real purpose that I can see for **float** variables these days is that it is easier to demonstrate problems with accuracy in numerical codes using **floats** instead of **doubles**.

- `double` : These are for storing numbers that have fractional parts, or decimal points, or are too large or too small to be expressed with an `int` variable.

These are the primitive data types. In addition to these, there are classes/objects `Integer`, `Float`, and `Double` that can be used for “object-like” actions on basic numerical values. If you google “java integer” or “java float” or “java double” then you will pull up the documentation for these classes. Indeed, there are some slight complications in going back and forth between primitive variables and classes that are almost the same thing. But if anything it would be the primitive variables that would be dropped if one were to “simplify” the language, not the classes, so you should think of the primitive variables as providing a simpler way to write programs than would be the “normal” way.

2 Precision of integer values

The first bitter lesson about computing is that not all numbers can be expressed exactly, and that not all arithmetic operations that once might like to do can be done without having “the answer” from the computer being actually incorrect. Integers these days in most modern programming languages and on most computers are stored using four bytes of memory. This provides 32 bits of precision, so there are a total of 2^{32} bit patterns available for storing numbers. If we didn’t need to deal with negative numbers, we could use those bit patterns exactly as the integers running 0 through $2^{32} - 1$. But we need negative integers, and what is done on (practically?) all computers these days is *twos-complement* arithmetic. If we had only four bits (instead of 32), twos complement would work as follows:

-8	1000	8	-4	1100	12	0	0000	0	4	0100	4
-7	1001	9	-3	1101	13	1	0001	1	5	0101	5
-6	1010	10	-2	1110	14	2	0010	2	6	0110	6
-5	1011	11	-1	1111	15	3	0011	3	7	0111	7

In each of the four blocks of the table, the first column is the number to be represented in twos-complement. The second number is the bit pattern that is used to represent the number, assuming that we are using four bits. The third column has the decimal equivalent of the binary number represented

by the bit pattern. For example, the integer -5 is represented with the bit pattern 1011 , and that particular bit pattern, if interpreted as a binary number, would be $8 + 2 + 1 = 11$ in decimal.

The way to convert from a positive number (say 6 , with bit pattern 0110) to the negative value -6 , is this; change all the 0 bits to 1 bits and all the 1 bits to zero bits, so that 0110 becomes 1001 . Then add 1 , so the resulting bit pattern is 1010 after the carry, and that's the bit pattern for -6 .

You're probably asking why one would do this. The reason is this. Converting from human form (decimal, with plus and minus signs), into twos-complement is usually done only once, when the program starts up, and converting back to decimal so humans can read the numbers easily, is only done at the end of the program. All the rest of the time, we should try to make the arithmetic easy for the computer. And twos-complement is especially easy to do for the computer and the computer hardware.

The main advantage of something like twos-complement is that the mechanical process of doing arithmetic, as it is done by the hardware, can be done without regard to whether the numbers are positive or negative. One has integers to add? Just add them. If both integers are positive, then arithmetic is exactly as it would normally be done. If one or both integers is negative, then the only thing to remember is that there will be a carry left, *and we simply choose to ignore that carry.*

If we add 6 to -2 , for example, we add the binary numbers 0110 and 1110 to get the five-bit number 10100 . In decimal (using the third column in each block of the table above) this is $6 + 14 = 20$. Now just throw away that leftmost bit, which is 16 in decimal, and $20 - 16 = 4$, which is the correct answer.

Multiplication is similarly simple, and multiplication illustrates the standard problem of finite precision in dealing with integers. Just do the mechanical process of multiplying the numbers together, and throw away the high order bits. If you multiply 56 times 23 , for example, you get the four digit number 1288 . But if you only have space for two digits in the answer, which two do you keep? With computers and integer arithmetic, invariably it is the *most significant* bits that are thrown away. Integer arithmetic would return 88 for the product above, and throw away the 1200 . (This works exactly backwards from the way that double precision arithmetic works.)

The example with 1288 isn't quite correct, in the following way. Multiplication of an n -digit number times an m -digit number will produce a product that $n + m$ digits long. In twos complement notation, if it happens that

the leftmost bit is a 1, then that bit pattern happens to represent a *negative* number, so it's possible to multiply two positive numbers and have the computer come back with a negative result.

The moral of all of this is: If you are doing arithmetic, you need to verify that the precision you need for the arithmetic (and this includes all the intermediate operations, not just the inputs and the eventual outputs) stays within the (usually) 32 bits of integer precision. If you need precision beyond this, then all bets are off as to what the result will be.

3 Precision of floating point values

The generic name for arithmetic using either `float` or `double` variables is *floating point arithmetic*, so named because it works like scientific notation in which the decimal (or binary) point floats as needed to maintain the most significant part of the arithmetic result.

Any number that requires a fractional part, meaning a decimal point, has to be done using floating point arithmetic.

Necessarily, since computers work in binary, and the precision is finite, the arithmetic is approximate and not exact. This is the same problem as trying to represent $1/3$ in decimal; at some point the infinite fraction $0.3333\dots$ has to be truncated to a finite number of places.

In the old days, FP arithmetic was done using a variety of hardware algorithms, and the answers one got from the same program would vary if done on different computers. Further, there was no agreement about what should happen in case of arithmetic errors. To solve these problems, or at least to standardize the glitches, the IEEE 754 floating point standard was adopted (google "IEEE 754").

The simple (and not 100% correct) version of IEEE 754 is this:

The IEEE 754 standard for 64-bit floating point arithmetic uses the leftmost bit of a 64-bit long pattern as the sign bit. If this is zero, then the number that is represented is positive; if 1, then negative. The next eleven bits are for the exponent, as a power of 2, in "excess-1023" notation. Eleven bits allow for numbers from 0 through 2047 to be represented. The actual number stored in the exponent block is the exponent plus 1023, so that both positive and negative exponents can be represented.

The remaining 52 bits are used for the *significand*, basically a fraction written in binary. For *normalized* numbers (these are the only ones I am

going to talk about, and that's why I said above that this is not 100% correct, because the full truth is a lot more complicated and we don't need to go into it), the fraction is left-adjusted in the 52-bit field. That is, we would normalize a binary number 0.001 to be 0.1×2^{-2} . Further, since we normalize numbers, the left most bit is *always* 1 and we don't have to store that bit, which means we get one extra bit of precision.

There are a great many rules about floating point numbers in the 754 standard. These deal with the representation of infinity, of the difference between positive and negative infinity, of the "not a number" (NaN) values, and such. You get something that is not a number by doing arithmetic with legitimate numbers but whose result can no longer properly be represented using the 754 standard.

Note that integer arithmetic fixes the right hand location of the point and throws away the high order, most significant, digits. Floating point arithmetic works more normally by holding on to the left end of the bit pattern of the significand, thus keeping the most significant bits and throwing away the bits that are least important.

All of this leads to the following observation: computer arithmetic is not exact due to the finiteness of the precision of the bit patterns used to store numbers. In order to get "correct" answers from a computation, you the programmer need to make sure in advance that the operations you are doing do not cause overflow or other kinds of errors. With integer arithmetic, you usually get no warning whatsoever that an error has occurred in the arithmetic. With the IEEE 754 standard, you don't get any information about the normal loss of precision (in truncating a repeating 0.333333..., for example), but you usually do get some indication that the result might be infinite because you divided by zero.

The other lesson to remember when using floating point numbers is that you absolutely never test for exactly zero in floating point. There are so many bits of precision, and there is always some roundoff in any real computation, so even though algebraically you might think something ought to be zero, it's unlikely to be the case that the computer is going to have an absolute zero stored in the variable. When testing for what you think is zero, you should test that the absolute value of the result is smaller than some tolerance value (like 10^{-6} , say).