UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Spring 1998

Numbers

1 Integers

The Scheme language has a notion of integer data type that is particularly convenient for the programmer: Scheme integers correspond directly to mathematical integers and there are standard functions that correspond to the standard arithmetic operations on mathematical integers. While convenient for the programmer, this causes headaches for those who have to implement the language. Computer hardware has no built-in data type that corresponds directly to the mathematical integers, and the language implementor must build such a type out of more primitive components.

Historically, therefore, most "traditional" programming languages don't provide full mathematical integers either, but instead give programmers something that corresponds to the hardware's built-in data types. As a result, what passes for integer arithmetic in these languages is at least quite fast. What I call "traditional" languages include FORTRAN, the Algol dialects, Pascal, C, C++, Java, Basic, and many others. Here, I will discuss the integer types provided by Java, which is in many ways typical.

2 Modular arithmetic

The integral values in Java differ from the mathematical integers in that they come from a finite set (or *domain*). Specifically, the five integer types have the ranges shown in Table 1. With a limited range, the question that naturally arises is what happens when an arithmetic result falls outside this range (*overflows*). For example, what is the result of `1000000*10000`, in which the two operands are both of type `int`?

Type	Modulus	Minimum	Maximum
`long`	2^{64}	-2^{63}	$2^{63}-1$
		(-9223372036854775808)	(9223372036854775807)
`int`	2^{32}	-2^{31}	$2^{31}-1$
		(-2147483648)	(2147483647)
`short`	2^{16}	-2^{15}	$2^{15}-1$
		(-32768)	(32767)
`byte`	2^{8}	-2^{7}	$2^{7}-1$
		(-128)	(127)
`char`	2^{16}	0	$2^{16}-1$
		0	(65535)

Table 1: Ranges of values of Java integral types. Values of a type represented with n bits are computed modulo 2^n (the “Modulus” column).

Computer hardware, in fact, answers this question in various ways, and as a result, traditional languages prior to Java tended to finesse this question, saying that operations producing an out-of-range result were “erroneous,” or had “undefined” or “implementation-dependent” results. In fact, they also tended to finesse the question of the range of various integer types; in standard C, for example, the type `int` has *at least* the range of Java’s type `short`, but may have more. To do otherwise than this could make programs slower on some machines than they would be if the language allowed compilers more choice in what results to produce. The designers of Java, however, decided to ignore any possible speed penalty in order to avoid the substantial hassles caused by the fact that differences in the behavior of integers from one machine to another complicate the problem of writing programs that run on more than one type of machine.

Java solves the overflow question by saying that integer arithmetic is *modular*. In mathematics, we say that two numbers are identical “modulo N” if they differ by a multiple of N:

$$a \equiv b \bmod N \text{ iff there is an integer, } k\text{, such that } a - b = kN.$$

The numeric types in Java are all computed modulo some power of 2. Thus, the type `short` is computed modulo 2^{16}. Any attempt to convert an integral value, x, to type `short` gives a value that is equal to x modulo 2^{16}. There is an infinity of such values; the one chosen is the one that lies between -2^{15} and $2^{15}-1$, inclusive. For example, converting the values 256, 0, and -1000 to type `short` simply give 256, 0, and -1000, while converting 32768 (which is 2^{15}) to type `short` gives -32768 (because $32768 - (-32768) = 2^{16}$) and converting 131073 to type `short` gives 1 (because $1 - 131073 = -2 \cdot 2^{16}$).

It may occur to you from this last example that converting x to type `short` looks like taking the remainder of x when divided by 2^{16} (or, as it would be written in Java, ‘`x % 65536`’). This is almost true. In Java, division of integers, as in `x / y`, is defined to yield the result of the mathematical division with the remainder discarded. Thus, in Java, $\texttt{11/3} = \texttt{-11/-3} = 3$ and $\texttt{-11/3} = \texttt{11/-3} = -3$. Remainder on integer operands is then defined by the equation

```
x%y = x - (x/y)*y
```

where '/' means Java-style division of integers. Thus, in Java, $11\%3 = 11\% - 3 = 2$ and $-11\%3 = -11\% - 3 = -2$. However, to use an example from above, `32768%65536` is just 32768 back again, and one has to subtract $2^{16} = 65536$ to get the right `short` value. It is correct to say that converting x to type `short` is like taking the remainder from dividing by 2^{16} and subtracting 2^{16} if the result is $\geq 2^{15}$.

For addition, subtraction, and multiplication, it doesn't matter at what point you perform a conversion to the type of result you are after. This is an extremely important property of modular arithmetic. For example, consider the computation `527 * 1000 + 600`, where the final result is supposed to be a `byte` (range -128 to 127, modulo 256 arithmetic). Doing the conversion at the last moment gives

$$527 \cdot 1000 + 600 = 527600 \equiv -16 \bmod 256;$$

or we can first convert all the numerals to `byte`s:

$$15 \cdot -24 + 88 = -272 \equiv -16 \bmod 256;$$

or we can convert the result of the multiplication first:

$$527000 + 600 \equiv 152 + 88 = 240 \equiv -16 \bmod 256.$$

We always get the same result in the end.

Unfortunately, this happy property breaks down for division. For example, the result of converting 256/7 to a `byte` (36) is not the same as that of converting 0/7 to a `byte` (0), even though both 256 and 0 are equivalent as `byte`s (i.e., modulo 256). Therefore, we have to be a little bit specific about exactly when conversions happen during the computation of an expression involving integer quantities. The rule is:

> To compute $x \oplus y$, where $\oplus$ is any of the Java operations `+`, `-`, `*`, `/`, or `%`, and x and y are integer quantities (of type `long`, `int`, `short`, `char`, or `byte`),
>
> - If either operand has type `long`, compute the mathematical result converted to type `long`.
> - Otherwise, compute the mathematical result converted to type `int`.

By "mathematical result," I mean the result as in normal arithmetic, where '/' is understood to throw away any remainder.

So, for example, consider

```
    short x = 32767;
    byte y = (byte) (x * x * x / 15);
```

The notation ‘`(`T`)` V’ is called a *cast;* it means “V converted to type T.” The cast construct has higher precedence than arithmetic operations, so that `(byte)x*x*x` would have meant `((byte)x)*x*x`. According to the rules, `y` is computed as

```
    short x = 32767;
    byte y = (byte) ((int) ((int) (x*x) * x) / 15);
```

The computation proceeds:

```
    x*x --> 1073676289
    (int) 1073676289 --> 1073676289
    1073676289 * x --> 35181150961663
    (int) 35181150961663 --> 1073840127
    1073840127 / 15 --> 71589341
    (byte) 71589341 --> -35
```

If instead I had written

```
    byte y = (byte) ((long) x * x * x / 15);
```

it would have been evaluated as

```
    byte y = (byte) ((long) ((long) ((long) x * x) * x) / 15);
```

which would proceed:

```
    (long) x --> 32767
    32767 * x --> 1073676289
    (long) 1073676289 --> 1073676289
    1073676289 * x --> 35181150961663
    (long) 35181150961663 --> 35181150961663
    35181150961663 / 15 --> 2345410064110
    (byte) 2345410064110 --> -18
```

3 Why this way?

All these remainders seem rather tedious to us humans, but because of the way our machines represent integer quantities, they are quite easy for the hardware. Let’s take the type `byte` as an example. Typical hardware represents a byte x as a number in the range 0–255 that is equivalent to x modulo 256, encoded as an 8-digit number in the binary number system (whose digits—called *bits*—are 0 and 1). Thus,

```
0    --> 00000000₂
1    --> 00000001₂
2    --> 00000010₂
5    --> 00000101₂
127  --> 01111111₂
-128 --> 10000000₂   (≡ 128 mod 256)
-1   --> 11111111₂   (≡ 255 mod 256)
```

As you can see, all the numbers whose top bit (representing 128) is 1 represent negative numbers; this bit is therefore called the *sign bit*. As it turns out, with this representation, taking the remainder modulo 256 is extremely easy. The largest number representable with eight bits is 255. The ninth bit position (100000000_2) represents 256 itself, and all higher bit positions represent multiples of 256. That is, every multiple of 256 has the form

$$b_0 \cdots b_n 00000000,$$

which means that to compute a result modulo 256 in binary, one simply throws away all but the last eight bits.

Be careful in this notation about converting to a number format that has more bits. This may seem odd advice, since when converting (say) `byte`s (eight bits) to `int`s (32 bits), the value does not change. However, the `byte` representation of -3 is 253, or in binary 11111101_2, whereas the `int` representation of -3 is

$$11111111111111111111111111111101_2 = 4294967293.$$

In converting from a `byte` to an `int`, therefore, we duplicate the sign bit to fill in the extra bit positions (a process called *sign extension*). Why is this the right thing to do? Well, the negative quantity $-x$ as a `byte` is represented by $2^8 - x$, since $2^8 - x \equiv -x \bmod 2^8$. As an `int`, it is represented by $2^{32} - x$, and

$$2^{32} - x = (2^{32} - 2^8) + (2^8 - x) = 11111111111111111111111100000000_2 + (2^8 - x).$$

4 Manipulating bits

One can look at a number as a bunch of bits, as shown in the last section. Java (like C and C++) provides operators for treating numbers as bits. The *bitwise operators*—`&`, `|`, `^`, and `~`—all operate by lining up their operands and then performing some operation on each bit or pair of corresponding bits, according to the following tables:

		Operand Bits (L,R)			
Operation		$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
`&`	(and)	0	0	0	1
`|`	(or)	0	1	1	1
`^`	(xor)	0	1	1	0

	Operand Bit	
Operation	0	1
`~` (not)	1	0

The “xor” (exclusive or) operation also serves the purpose of a “not equal” operation: it is 1 if and only if its operands are not equal.
In addition, the operation `x<<N` produces the result of multiplying `x` by 2^N (or shifting N 0’s in on the right). `x>>>N` produces the result of shifting N 0’s in on the left, throwing away bits on the right. Finally, `x>>N` shifts N copies of the sign bit in on the left, throwing away bits on the right. This has the effect dividing by 2^N and rounding down (toward $-\infty$).

For example,

```
                  int x = 42;    // == 0...0101010 base 2
                  int y = 7;     // == 0...0000111

x & y == 2     // == 0...0000010 |  x << 2 == 168   // == 0...10101000
x | y == 47    // == 0...0101111 |  x >> 2 == 10    // == 0...00001010
x ˆ y == 45    // == 0...0101101 |  ˜y << 2 == -32  // == 1...11100000
˜y    == -8    // == 11...111000 |  ˜y >> 2 == -2   // == 1...11111110
                                 |  ˜y >>> 2 == 2^30 − 2
                                 |                  // == 00111...1110
```

As you can see, even though these operators manipulate bits, whereas `int`s are supposed to be numbers, no conversions are necessary to “turn `int`s into bits.” This isn’t surprising, given that the internal representation of an `int` actually *is* a collection of bits, and always has been; these are operations that have been carried over from the world of machine-language programming into higher-level languages. They have numerous uses; some examples follow.

Packing. Sometimes one wants to save space by packing several small numbers into a single `int`. For example, I might know that `w`, `x`, and `y` are each between 0 and $2^9 - 1$. I can *pack* them into a single `int` with

```
    z = (w<<18) + (x<<9) + y
```

and from this `z`, I can extract `w`, `x`, and `y` with

```
    w = z >>> 18; x = (z >>> 9) & 0x1ff; y = z & 0x1ff;
```

(In this case, the `>>` operator would work just as well.) Alternatively, you can extract `x` with

```
    x = (z & 0x3fe00) >>> 9;
```

Flags. A trick you will sometimes see in C and C++ programs is that of passing a bunch of *flags* in a single argument. For example, you might have some kind of formatting function that takes a number of yes/no options (is the argument hexadecimal, is it signed, is it left or right justified, etc.). If you define these flags as powers of two:

```
final static int HEX = 1, DEC = 2, OCT = 4, UNSIGNED = 8, ...;
...
/** Return a printable rendition of X, formatted according to
 *  FLAGS. */
String formatNumber(int x, int flags) ...
```

then the user can write

```
 formatNumber(z, HEX | UNSIGNED);
```

to indicate that `z` is supposed to be formatted into an unsigned, hexadecimal number. Inside `formatNumber`, you can test for the presence of these flags with conditions like this:

```
 if ((flags & UNSIGNED) != 0) ...
```

5 Unsigned numbers

The C and C++ languages have explicitly *unsigned* types corresponding to the signed types `int`, `short`, etc. For simplicity, Java does not. This is because the rules of modular arithmetic give you the *effect* of having unsigned numbers, as long as you are careful about a few things: specifically, conversions, division, comparison, input, and output.

Here, I'll stick to bytes to keep the numbers small, but the same applies to all the signed integer types. The byte value -1 has the internal representation 11111111_2, which can also be read as 255. So consider, for example, the sum `(byte) -1 - (byte) 5`. This gives -6, as you expect, whose representation is 11111010, which may also be read as 250. Or consider the product (byte) 13 * (byte) 10. When converted to a byte quantity, this gives `-126`, whose internal representation is `10000010`, which happens also to be 130.

Conversion. If you wish to treat a byte quantity as unsigned, then converting it to a larger format (one with more bits) requires care. For example, as the discussion above implies, after

```
byte b = (byte) 250;
int i = b;
```

the variable `i` will contain -6, which, if treated as an unsigned `int` quantity is $2^{32} - 6$, rather than 250. Therefore, the assignment to `i` requires a masking operation:

```
int i = b & 0xff;
```

The same holds for conversions from `byte` to the other numeric types, when the quantities are intended as unsigned, as well conversions from `short` to `int` or `long`, and from `int` to `long`. The type `char` in Java is already unsigned.

Comparison. For the signed integer types (`byte`, `short`, `int`, and `long`), the comparison operators perform signed comparisons. For example, even if you think of the contents of a `byte` variable `b` as being 250, it will still compare less than 1, since Java treats its value as -6. When comparing numeric types with fewer bits than `int`, you can use masking to get rid of the sign bits, as in

```
(b & 0xff) >= 1   /* For byte b, or */
(s & 0xffff) >= 1 /* for short s */
```

Likewise, for `int`s that are supposed to be unsigned, you can use `long`:

```
(i & 0xffffffffL) >= 1
```

Alas, this approach does not work for unsigned `longs`, which require some fancy footwork[1]. Frankly, it would be a great deal better here for Java to have unsigned integer types and unsigned comparisons, as do C and C++, but we can still get the same effect. Let's take the comparison `L0<L1` as a general example. Consider the cases:

- If both quantities are positive, then their signed values (i.e., their official Java values) are the same as the unsigned numbers they represent. In this case, the sign of `L0-L1` will be negative iff `L0<L1`.
- If both quantities are negative, then their signed values (i.e., their official Java values) are 2^{64} less than the unsigned numbers they represent. In this case, the sign of `L0-L1` will still be negative iff `L0<L1`.
- If `L0` is positive and `L1` is negative, then as unsigned numbers `L0` is less than 2^{63} and `L1` is at least 2^{63}, so `L0<L1` must be true.
- If `L0` is negative and `L1` is positive, then as unsigned numbers `L1` is less than 2^{63} and `L0` is at least 2^{63}, so `L0<L1` must be false.

In other words,

- If the sign bits of `L0` and `L1` are equal, then signed comparison and unsigned comparison `L0<L1` give the same value.
- Otherwise, the unsigned comparison `L0<L1` is true iff `L1` is negative and `L0` is non-negative.

[1]The following material on unsigned comparison of Java `long` values is of little practical importance. Personally, I've never had to do this sort of comparison of unsigned 64-bit quantities. I include it here as a useful exercise of your understanding of integer number representations and of bit manipulation. Make sure, therefore, that you understand everything here, even if you can't see where you'd ever need it.

The following expressions both yield the value **true** if and only if `L0<L1` when treated as unsigned quantities. Make sure you understand why each of them works:

```
( (L0 < 0) == (L1 < 0) && L0 < L1 ) || ( L0 >= 0 && L1 < 0 )
( (~(L0 ^ L1) & (L0 - L1)) | (L1 & ~L0) ) < 0
```

Don't be thrown off by that strange subexpression `(L0<0) == (L1<0)`. It really is a comparison of two boolean values for equality and it really does work.

Divison. As usual, unsigned division is tricky. For dividing unsigned `int`s, you can convert to type `long` and divide:

```
((long) x & 0xffffffff) / ((long) y & 0xffffffff)
// What would happen without the "& 0xff..." parts?
```

but division of `long` quantities interpreted as unsigned is tricky indeed. I'll leave that to you.

Input and Output. Java's standard library does not include any routines for treating numbers as unsigned, and that includes input and output of unsigned quantities. That is, nothing directly prints the `int` value -1 as 4294967295, although you can fake it in this case by converting -1 to an unsigned `long` value. Our `ucb.io.FormatOutputStream` class allows you to output numbers as unsigned. Format codes `%u`, `%o`, and `%x` all interpret numbers as unsigned (`%u` is unsigned decimal).

6 Floating-Point Numbers

Just as it provides general integers, Scheme also provides rational numbers—quotients of two integers. Just as the manipulation of arbitrarily large integers has performance problems, so too does the manipulation of what are essentially pairs of arbitrarily large integers. It isn't necessary, furthermore, to have large rational numbers to have large integer numerators and denominators. For example, $(8/7)^{30}$ is a number approximately equal to 55, but its numerator has 28 digits and its denominator has 27. Most of the computations we do with such numbers ultimately concern physical quantities that we can only measure to some finite precision anyway, and the precision afforded by large numerators and denominators is largely wasted.

Therefore, standard computer systems provide some form of limited-precision rational arithmetic known as *floating-point arithmetic*. This may be provided either directly by the hardware (as on Pentiums, for example), or by means of standard software (as on the older 8086 processors, for example).

Java has adopted what is called IEEE Standard Binary Floating-Point Arithmetic. The basic idea behind a floating-point type is to represent only numbers having the form

$$\pm b_0.b_1 \cdots b_{n-1} \times 2^e,$$

where n is a fixed number, e is an integer in some fixed range (the *exponent*), and the b_i are binary digits (0 or 1), so that $b_0.b_1 \cdots b_{n-1}$ is a fractional binary number (the *significand*). In Java, there are two floating-point types:

- `float`: $n = 24$ and $-127 < e < 128$;
- `double`: $n = 53$ and $-1023 < e < 1024$.

In addition to these numbers, both `float` and `double` contain some additional special values. Here they are for type `double` (The type `float` has similar definitions):

- $\pm\infty$: the results of producing a result that is outside the range of representable values. The Java constants `Double.NEGATIVE_INFINITY` and `Double.POSITIVE_INFINITY` have these values. You can test `x` to see if it is infinite with `Double.isInfinite(x)`.
- NaN (Not a Number). There are actually several of these. They represent the results of undefined expressions, such as `0/0`, $\infty - \infty$, or any arithmetic operation on `NaN`s. One checks a value `x` to see if it is not a number with `Double.isNaN(x)` (you can't use `==` for this test because a NaN has the odd property that it is not equal, greater than, or less than any other value, including itself!)
- `-0.0`. Mathematically, $0 = -0$. However, in IEEE floating point, this value is distinct from 0 (except, confusingly, that `0.0==-0.0`). The difference shows up in the fact that `1/-0.0` is negative infinity. We won't get into why this is a useful thing. Take a course in numerical analysis if you are curious.

In what follows, I am going to talk only about the type `double`. This is the default type for floating-point literals, and in the type commonly used for computation. The type `float` is entirely analogous, but since it is not as often used, I will avoid redundancy and not mention it further. The type `float` is useful in places where space is at a premium and the necessary precision is not too high.

The result of any arithmetic operation involving floating-point quantities is rounded to the nearest representable floating-point number (or to $\pm\infty$ if out of range). In case of ties, where the unrounded result is exactly halfway between two floating-point numbers, one chooses the one that gives a last binary digit of 0 (the rule of *round to even.*) The only exception to this rule is that conversions of floating-point to integer types, using a cast such as `(int) x`, always *truncate*—that is, round to the number nearest to 0, throwing the fractional part away[2].

In principle, I've now said all that needs to be said. However, there are many subtle consequences of these rules. You'll have to take a course in numerical analysis to learn all of them, but for now, here are a few important points to remember.

[2]The handling of rounding to integer types is *not* the IEEE convention; Java inherited it from C and C++.

6.1 Binary vs. decimal

Computers use binary arithmetic because it leads to simple hardware (i.e., cheaper than using decimal arithmetic). There is, however, a cost to our intuitions to doing this: although any fractional binary number can be represented as a decimal fraction, the reverse is not true. For example, the nearest `double` value to the decimal fraction 0.1 is

$$0.1000000000000000055511151231257827021181583404541015625$$

so when you write the literal `0.1`, or when you compute `1.0/10.0`, you actually get the number above. You'll see this sometimes when you print things out with a little too much precision. For example, the nearest `double` number to 0.123 is

$$0.122999999999999998223643160\ldots$$

so that if you print this number with the `%24.17e` format from our library, you'll see that bunch of 9s. Fortunately, less precision will get rounded to something reasonable.

6.2 Round-off

For two reasons, the loop

```
double x; int k
for (x = 0.1, k = 1; x <= N; x += 0.1, k += 1)
{ ... }
```

will not necessarily execute $10N$ times. The first reason is the one mentioned above: 0.1 is only approximately representable. The second is that each addition of this approximation to `x` may round. The rounding is sometimes up and sometimes down, but eventually the combined effects of these two sources of error will cause `x` to drift away from the mathematical value of $0.1k$ that the loop naively suggests. To get the effect that was probably intended for the loop above, you need something like this:

```
for (int kx = 1; kx <= 20; k += 1) {
    double x = kx * 0.1;
    // or double x = (double) kx / 10.0;
```

(The division is more accurate, but slower). With this loop, the values of `x` involve only one or two rounding errors, rather than an ever-increasing number.

On the other hand, since integers up to $2^{53} - 1$ (about 9×10^{15}) *are* represented exactly,

```
for (double x = 1.0, k = 1; x <= N; x += 1.0, k += 1) { ... }
```

will execute exactly N times (if $N < 2^{53}$) and `x` and `k` will always have the same mathematical value. In general, operations on integers in this range (except, of course, division) give exact results. If you were doing a computation involving integers having 10–15 decimal digits, and you were trying to squeeze seconds, floating-point might be the way to go, since for operations like multiplication and division, it can be faster than integer arithmetic on `long` values. I doubt that such an occasion is likely to arise, but you never know.

In fact, with care, you might even use floating-point for financial computations, computed to the penny (it has been done). I say "with care," since 0.01 is not exactly representable in binary. Nevertheless, if you represent quantities in pennies (or possibly mills) instead of in dollars, you can be sure of the results of additions, subtractions, and multiplications, at least up to $9,999,999,999,999.99.

When the exponents of results exceed the largest one representable (*overflow*), the results are approximated by the appropriate infinity. When the exponents get too small to represent at all (*underflow*), the result will be 0. In IEEE (and Java) arithmetic, there is an intermediate stage called *gradual underflow*, which occurs when the exponent is at its minimum, and the first significand bit (b_0) is 0.

We often describe the rounding properties of IEEE floating-point by saying that results are correct "to 1/2 unit in the last place (ulp)," because rounding off changes the result by at most that much. Another, looser characterization is to talk about *relative error*. The relative-error bound is pessimistic, but has intuitive advantages. If `x` and `y` are two `double` quantities, then (in the absence of overflow or any kind of underflow) the computed result, `x*y`, is related to the mathematical result, $\mathtt{x} \cdot \mathtt{y}$, by

$$\mathtt{x} * \mathtt{y} = \mathtt{x} \cdot \mathtt{y} \cdot (1 + \epsilon), \text{ where } |\epsilon| \leq 2^{-53}.$$

and we say that ϵ is the relative error (it's bound is a little larger than 10^{-16}, so you often hear it said that double-precision floating point gives you something over 15 significant digits). Division has essentially the same rule.

Addition and subtraction also obey the same form of relative-error rule, but with an interesting twist: adding two numbers with opposite signs and similar magnitudes (meaning within a factor of 2 of each other) always gives an *exact* answer. For example, in the expression `0.1-0.09`, the subtraction itself does not cause any round-off error (why?), but since the two operands are themselves rounded off, the result is not exactly equal to 0.01. The subtraction of nearly equal quantities tends to leave behind just the "noise" in the operands (but it gets that noise absolutely right!).

6.3 Spacing

Unlike integers, floating-point numbers are not evenly spaced throughout their range. Figure 1 illustrates the spacing of simple floating-point numbers near 0 in which the significand has 3 bits

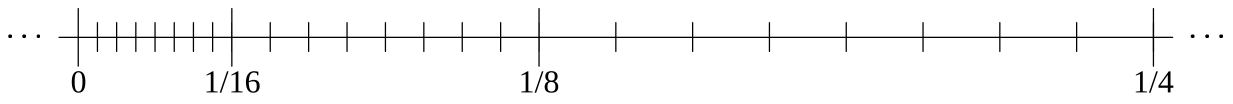

Figure 1: Non-negative 3-bit floating-point numbers near 0, showing how the spacing expands as the numbers get bigger. Each tick mark represents a floating-point number. Assume that the minimum exponent is -4, so that the most closely spaced tick marks are $2^{-6} = 1/64$ apart.

rather than Java's 24 or 53. Because the numbers get farther apart as their magnitude increases, the absolute value of any round-off error also increases.

There are numerous pitfalls associated with this fact. For example, many numerical algorithms require that we repeat some computation until our result is "close enough" to some desired result. For example, we can compute the square root of a real number y by the recurrence

$$x_{i+1} = x_i + \frac{y - x_i^2}{2x_i}$$

where x_i is the i^{th} approximation to $\sqrt{y}$. We could decide to stop when the error $|y - x_i^2|$ become small enough[3]. If we decided that "small enough" meant, say, "within 0.001," then for values of y less than 1 we would get very few significant digits of precision and for values of y greater than 10^{13}, we'll never stop. This is one reason relative error, introduced in the last section, is useful; no matter where you are on the floating-point scale, round off always produces the same relative error.

6.4 Equality

Many textbooks incorrectly tell you never to compare floating-point numbers for equality, but rather to check to see whether they are "close" to each other. This is highly misleading advice (that's more diplomatic than "wrong," isn't it?). It is true that naive uses of `==` can get you into trouble; for example, you should not expect that after setting `x` to `0.0001`, `x*10000==1.0`. That simply follows from the behavior of round off, as we've discussed.

However, one doesn't have to be naive. First, we've seen that (up to a point) `double` integers work like `ints` or `longs`. Second, IEEE standard arithmetic is designed to behave very well around many singularities in one's formulas. For example, suppose that $f(x)$ approaches 0 as x approaches 1—for concreteness, suppose that $f(x)$ approximates $\sin \pi x$—and that we want to compute $f(x)/(1-x)$, giving it the value 1 when $x = 1$. We can write the following computation in Java:

```
  if (x == 1.0)
```

[3]In actual practice, by the way, this convergence test isn't necessary, since the error in x_i^2 as a function of i is easily predictable for this particular formula.

```
      return 1.0;
   else
      return f(x) / (1.0 - x);
```

and it will always return a normal number (neither NaN nor infinity) when `x` is close to 1.0. Despite rounding errors, IEEE arithmetic guarantees that `1.0-x` will evaluate to 0 if and only if `x==1.0`.

6.5 Closing Words

You will find that on the subject of floating-point arithmetic in general, programmers are rather superstitious (that is a slightly more diplomatic word than *ignorant*, isn't it?). I'd bet that many would be doubtful about some of the properties I described above (see how many of your "experienced" acquaintances believe that subtraction of nearly equal quantities introduces no additional round-off error). To some extent, their superstition may be explained by the fact that historically, different machines have produced different results given the same operands. Some rounded their results; others chopped (rounded towards 0); others produced results for some operations that were off by more than a unit in the last digit; some actually produced results that could differ from run to run; some had infinities; some threw exceptions on overflow; and so forth. This all led to the common notion that floating-point values are mysterious, rather fuzzy quantities with unpredictable behaviors. Although for many calculations, this rather cavalier approach makes little difference, there are numerous examples where a little more specificity is very useful. That was a prime motivation for the IEEE standard.