This section under major construction.

One distinguishing feature that separates traditional computer science from scientific computing is its use of discrete mathematics (0s and 1s) instead of continuous mathematics and calculus. Transitioning from integers to real numbers is more than a cosmetic change. Digital computers cannot represent all real numbers exactly, so we face new challenges when designing computer algorithms for real numbers. Now, in addition to analyzing the running time and memory footprint, we must be concerned with the "correctness" of the resulting solutions. This challenging problem is further exacerbated since many important scientific algorithms make additional approximations to accommodate a discrete computer. Just as we discovered that some discrete algorithms are inherently too slow (polynomial vs. exponential), we will see that some floating point algorithms are too inaccurate (stable vs. unstable). Sometimes this problem can be corrected by designing a more clever algorithm. With discrete problems, the difficulty is sometimes intrinsic (NP-completeness). With floating point problems, the difficulty may also be inherent (ill conditioning), e.g., accurate long-term weather prediction. To be an effective computational scientist, we must be able to classify our algorithms and problems accordingly.

**Floating point.**

 Some of the greatest achievements of the 20th century would not have been possible without the floating point capabilities of digital computers. Nevertheless, this subject is not well understood by most programmers and is a regular source of confusion. In a February, 1998 keynote address entitled*Extensions to Java for Numerical Computing* James Gosling asserted "95% of folks out there are completely clueless about floating-point." However, the main ideas behind floating point are not difficult, and we will demystify the confusion that plagues most novices.

**IEEE 754 binary floating point representation.**

 First we will describe how floating point numbers are represented. Java uses a subset of the [IEEE 754 binary floating point standard](http://754r.ucbtest.org/standards/754xml.html) to represent floating point numbers and define the results of arithmetic operations. Virtually all modern computers conform to this standard.A float is represented using 32 bits, and each possible combination of bits represents one real number. This means that at most 232 possible real numbers can be exactly represented, even though there are infinitely many real numbers (even between 0 and 1). The IEEE standard uses an internal representation similar to scientific notation, but in binary instead of base 10. This covers a range from ±1.40129846432481707e-45 to ±3.40282346638528860e+38. with 6 or 7 significant decimal digits, including including plus infinity, minus infinity, and NaN (not a number). The number contains a sign bit *s* (interpreted as plus or minus), 8 bits for the exponent *e*, and 23 bits for the mantissa *M*. The decimal number is represented according to the following formula.

|  |
| --- |
| (-1)s × m × 2(e - 127) |

* *Sign bit s (bit 31)*. The most significant bit represents the sign of the number (1 for negative, 0 for positive).
* *Exponent field e (bits 30 - 23).* The next 8 bits represent the exponent. By convention the exponent is *biased* by 127. This means that to represent the binary exponent 5, we encode 127 + 5 = 132 in binary (10000100). To represent the binary exponent -5 we encode 127 - 5 = 122 in binary (01111010). This convention is alternative to two's complement notation for representing negative integers.
* *Mantissa m (bits 22 - 0).* The remaining 23 bits represent the mantissa, *normalized* to be between 0.5 and 1. This normalization is always possible by adjusting the binary exponent accordingly. Binary fractions work like decimal fractions: 0.1101 represents 1/2 + 1/4 + 1/16 = 13/16 = 0.825. Not every decimal number can be represented as a binary fraction. For example 1/10 = 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192 + ... In this case, the number 0.1 is approximated by the closest 23 bit binary fraction 0.000110011001100110011... One further optimization is employed. Since the mantissa always begins with a 1, there is no need to explicitly store this *hidden bit*.

As an example the decimal number 0.085 is stored as 00111101101011100001010001111011.

|  |
| --- |
| 0.085:  bits: 31 30-23 22-0  binary: 0 01111011 01011100001010001111011  decimal: 0 123 3019899 |

This exactly represents the number 2e-127 (1 + m / 223) = 2-4(1 + 3019899/8388608) = 11408507/134217728 = 0.085000000894069671630859375.

A double is similar to a float except that its internal representation uses 64 bits, an 11 bit exponent with a bias of 1023, and a 52 bit mantissa. This covers a range from ±4.94065645841246544e-324 to ±1.79769313486231570e+308 with 14 or 15 significant digits of accuracy.

**Precision vs. accuracy.**

 Precision = tightness of specification. Accuracy = correctness. Do not confuse precision with accuracy. 3.133333333 is an estimate of the mathematical constant π which is specified with 10 decimal digits of precision, but it only has two decimal digits of accuracy. As John von Neumann once said "There's no sense in being precise when you don't even know what you're talking about." Java typically prints out floating point numbers with 16 or 17 decimal digits of precision, but do not blindly believe that this means there that many digits of accuracy! Calculators typically display 10 digits, but compute with 13 digits of precision. Kahan: the mirror for the Hubble space telescope was ground with great precision, but to the wrong specification. Hence, it was initially a great failure since it couldn't produce high resolution images as expected. However, it's precision enabled an astronaut to install a corrective lens to counter-balance the error. Currency calculations are often defined in terms of a give precision, e.g., Euro exchange rates must be quoted to 6 digits.

**Roundoff error.**

 Programming with floating point numbers can be a bewildering and perilous process for the uninitiated. Arithmetic with integers is exact, unless the answer is outside the range of integers that can be represented (overflow). In contrast, floating point arithmetic is not exact since some real numbers require an infinite number of digits to be represented, e.g., the mathematical constants e and π and 1/3. However, most novice Java programmers are surprised to learn that 1/10 is not exactly representable either in the standard binary floating point. These roundoff errors can propagate through the calculation in non-intuitive ways.

* *Rounding of decimal fractions.* For example, the first code fragment below from [FloatingPoint.java](http://introcs.cs.princeton.edu/java/91float/FloatingPoint.java.html) prints false, while the second one prints true.

|  |
| --- |
| double x1 = 0.3;  double x2 = 0.1 + 0.1 + 0.1;  StdOut.println(x1 == x2);  double z1 = 0.5;  double z2 = 0.1 + 0.1 + 0.1 + 0.1 + 0.1;  StdOut.println(z1 == z2); |

* *Newton's method.* A more realistic example is the following code fragment whose intent is to compute the square root of c by iterating Newton's method. Mathematically, the sequence of iterates converges to √c from above, so that t2 - c > 0. However, a floating point number only has finitely many bits of accuracy, so eventually, we may expect t2 to equal c exactly, up to machine precision.

|  |
| --- |
| double EPSILON = 0.0;  double t = c;  while (t\*t - c > EPSILON)  t = (c/t + t) / 2.0; |

Indeed, for some values of c, the method works.

|  |
| --- |
| % java Sqrt 2 % java Sqrt 4 % java Sqrt 10  1.414213562373095 2.0 3.162277660168379 |

This might give us some confidence that our code fragment is correct. But a surprising thing happens when we try to compute the square root of 20. Our program gets stuck in an infinite loop! This type of error is called *roundoff error*. *Machine accuracy* is smallest number ε such that (1.0 + ε != 1.0). In Java, it is XYZ with double and XYZ with float. Changing the error tolerance ε to a small positive value helps, but does not fix the problem (see exercise XYZ).

We must be satisfied with approximating the square root. A reliable way to do the computation is to choose some error tolerance ε, say 1E-15, and try to find a value t such that |t - c/t| < ε t. We use *relative error* instead of *absolute error*; otherwise the program may go into an infinite loop (see exercise XYZ).

|  |
| --- |
| double epsilon = 1e-15;  double t = c;  while (Math.abs(t\*t - c) > c\*epsilon)  t = (c/t + t) / 2.0; |

* *Harmonic sum.* Possibly motivate by trying to estimate Euler's constant γ = limit as n approaches infinity of γn = Hn - ln n. The limit exists and is approximately 0.5772156649. Surprisingly, it is not even known whether γ is irrational.

Program [HarmonicSum.java](http://introcs.cs.princeton.edu/java/91float/HarmonicSum.java.html) computes 1/1 + 1/2 + ... + 1/N using single precision and double precision. With single precision, when N = 10,000, the sum is accurate to 5 decimal digits, when N = 1,000,000 it is accurate to only 3 decimal digits, when N = 10,000,000 it is accurate to only 2 decimal digits. In fact, once N reaches 10 million, the sum never increases. Although the Harmonic sum diverges to infinity, in floating point it converges to a finite number! This dispels the popular misconception that if you are solving a problem that requires only 4 or 5 decimal digits of ?precision, then you are safe using a type that stores 7. Many numerical computations (e.g., integration or solutions to differential equations) involve summing up alot of small terms. The errors can accumulate. This accumulation of roundoff error can lead to serious problems.

Better formula: γ ≈ γn - 1/(2n) + 1/(12n2). Accurate to 12 decimal places for n = 1 million.

Every time you perform an arithmetic operation, you introduce an additional error of at least ε. Kernighan and Plauger: "Floating point numbers are like piles of sand; every time you move one you lose a little sand and pick up a little dirt." If the errors occur at random, we might expect a cumulative error of sqrt(N) εm. However, if we are not vigilant in designing our numerical algorithms, these errors can propagate in very unfavorable and non-intuitive ways, leading to cumulative errors of N εm or worse.

**Financial computing.**

 We illustrate some examples of roundoff error that can ruin a financial calculation. Financial calculations involving dollars and cents involve base 10 arithmetic. The following examples demonstrate some of the perils of using a binary floating point system like IEEE 754.

* *Sales tax.* Program [Financial.java](http://introcs.cs.princeton.edu/java/91float/Financial.java.html) illustrates the danger. Calculate the 9% sales tax of a 50 cent phone call. (or 5% sales tax on a $0.70 phone call). Using IEEE 754, 1.09 \* 50 = xxx. This results gets rounded down to 0.xx even though the exact answer is 0.xx, which the telephone company (by law) is required to round down to 0.xx (using Banker's rounding). In contrast, a 14% tax on a 75 cent phone call yields xxx. might get rounded up to x.xx even though (by law) the phone company must round down the number (using Banker's rounding) to x.xx. This difference in pennies might not seem significant, and you might hope that the effects cancel each other out in the long run. However, taken over hundreds of millions of transactions, this [salami slicing](http://en.wikipedia.org/wiki/Salami_slicing) might result in millions of dollars. Reference: Superman III (1983), Hackers (1995), and Office Space (1999). In Office Space, three friends infect the accounting system with a computer virus that rounds down fractions of a cent and transfer it into their account.

For this reason, some programmers believe that you should always use integer types to store financial values instead of floating point types. The next example will show the perils of storing financial values using type int.

* *Compound interest.* This example introduces you to the dangers of roundoff error. Suppose you invest $1000.00 at 5% interest, compounded daily. How much money will you end up with after 1 year? If the bank computes the value correctly, you should end up with $1051.27 since the exact formula is

|  |
| --- |
| a \* (1 + r/n)n |

which leads to 1051.2674964674473.... Suppose, instead, that the bank stores your balance as an integer (measured in pennies). At the end of each day, the bank calculates your balance and multiplies it by 1.05 and rounds the result to the nearest penny. Then, you will end up with only $1051.10, and have been cheated out of 17 cents. Suppose instead the bank rounds *down* to the nearest penny at the end of each day. Now, you will end up with only $1049.40 and you will have been cheated out of $1.87. The error of not storing the fractions of a penny accumulate and can eventually become significant, and even fraudulent. Program [CompoundInterest.java](http://introcs.cs.princeton.edu/java/91float/CompoundInterest.java.html).

Instead of using integer or floating point types, you should use Java's BigDecimal library. This library has two main advantages. First, it can represent decimal numbers exactly. This prevents the sales tax issues of using binary floating point. Second, it can store numbers with an arbitrary amount of precision. This enables the programmer to control the degree to which roundoff error affects the computation.

**Other source of error.**

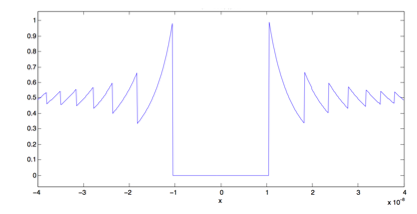
 In addition to roundoff error inherent when using floating point arithmetic, there are some other types of approximation errors that commonly arise in scientific applications.

* *Measurement error.* The data values used in the computation are not accurate. This arises from both practical (inaccurate or imprecise measuring instruments) and theoretical (Heisenberg uncertainty principle) considerations. We are primarily interested in models and solution methods whose answer is not highly sensitive to the initial data.
* *Discretization error.* Another source of inaccuracy results from discretizing continuous system, e.g., approximating a transcendental function by truncating its Taylor expansion, approximating an integral using a finite sum of rectangles, finite difference method for finding approximate solutions to differential equations, or estimating a continuous function over a lattice. Discretization error would still be present even if using exact arithmetic. It is often unavoidable, but we can reduce truncation error by using more refined discretization. Of course, this comes at the price of using more resources, whether it be memory or time. Discretization error often more important than roundoff error in practical applications.
* *Statistical error.* Not enough random samples.

**Catastrophic cancellation.**

 Devastating loss of precision when small numbers are computed from large numbers by addition or subtraction. For example if x and y agree in all but the last few digits, then if we compute z = x - y, then z may only have a few digits of accuracy. If we subsequently use z in a calculation, then the result may only have a few digits of accuracy.

* *Plotting a function.* Try plotting f(x) = (1 - cos x) / (x^2) from x = -4 \* 10^-8 to 4 \* 10^-8. In this region the mathematical function f(x) is approximately constant, with the value 0.5. However, in IEEE floating point, it is most definitely not! Program [Catastrophic.java](http://introcs.cs.princeton.edu/java/91float/Catastrophic.java.html) does this, with some very surprising results.



* *Exponential function.* Now we consider a more subtle and pernicious consequence of catastrophic cancellation. Program [Exponential.java](http://introcs.cs.princeton.edu/java/91float/Exponential.java.html) computes ex using the Taylor series

|  |
| --- |
| ex = 1 + x + x2/2! + x3/3! + x4/4! + ... |

This series converges uniformly and absolutely for all values of x. Nevertheless, for many negative values of x (e.g., -25), the program obtains no correct digits, no matter how many terms in the series are summed. For example when x = -25, the series converges in floating point to -7.129780403672078E-7 (a negative number!), but the true answer is 1.3887943864964021E-11. To see why, observe that term 24 in the sum is 2524/24! and term 25 is -2525/25!. In principle, they should exactly cancel each other out. In practice, they cancel each other out catastrophically. The size of these terms (5.7 \* 109) is 20 orders of magnitude greater than the true answer, and any error in the cancellation is magnified in the calculated answer. Fortunately, in this case, the problem is easily rectified (see Exercise XYZ).

**Numerical analysis.**

 Lloyd Trefethen (1992) "Numerical analysis is the study of algorithms for the problems of continuous mathematics."

* *Stability.* A mathematical problem is *well-conditioned* if its solution changes by only a small amount when the input parameters changes by a small amount. An algorithm is *numerically stable* if the output of the algorithm changes by only a small amount when the input data changes by a small amount. Numerical stability captures how errors are propagated by the algorithm. Numerical analysis is the art and science of finding numerically stable algorithms to solve well-conditioned problems. Accuracy depends on the conditioning of problem and the stability of the algorithm. Inaccuracy can result from applying a stable algorithm to an ill-conditioned problem or an unstable algorithm to a well-conditioned problem.
  + *Exponential function.* For a simple example, computing f(x) = exp(x) is a well conditioned problem since f(x + ε) = .... One algorithm for computing exp(x) is via its Taylor series. Suppose we estimate f(x) = exp(x) using the first four terms of its Taylor approximation: g(x) = 1 + x + x2/2 + x3/3!. Then f(1) = 2.718282, g(1) = 2.666667. However, it is unstable since if x < 0 ..... A stable algorithm is to use the Taylor series if x is nonnegative, but if x is negative, compute e-x using a Taylor series and take the reciprocal.
  + *(1 - cos x) / (x^2) function.* Give a stable way to evaluate this function.
* *Conditioning.* We saw an example of a non-stable algorithm for computing exp(x). We also saw a stable algorithm for computing it. Sometimes, we are not always so lucky. We say that a problem is *ill-conditioned* if there is no stable algorithm for solving it.

Addition, multiplication, exponentiation, and division of positive numbers are all well-conditioned problems. So is computing the roots of a quadratic equation. (See Exercise XYZ.) Subtraction is ill-conditioned. So is finding the roots of a general polynomial. The conditioning of the problem of solving Ax = b depends on the matrix A.

*Population dynamics.* The Verhulst equation is a simplified model of population dynamics.

|  |
| --- |
| xn = (R + 1)xn-1 - R (xn-1)2 |

Program [Verhulst.java](http://introcs.cs.princeton.edu/java/91float/Verhulst.java.html) reads in a command line parameter R and iterates the Verhulst equation for 100 iterations, starting with x0 = 0.5. It orders the computation in four different, but mathematically equivalent, ways. All lead to significantly different results when R = 3, none of which is remotely correct (as can be verified using exact arithmetic in Maple). ([Reference](http://www.lactamme.polytechnique.fr/Mosaic/descripteurs/Kepler.02..html))

|  |
| --- |
| (R+1)x-R(xx) (R+1)x-(Rx)x ((R+1)-(Rx))x x + R(x-xx) correct  0: 0.5000000000 0.5000000000 0.5000000000 0.5000000000 0.5000000000  10: 0.3846309658 0.3846309658 0.3846309658 0.3846309658 0.3846309658  20: 0.4188950250 0.4188950250 0.4188950250 0.4188950250 0.4188950250  30: 0.0463994725 0.0463994756 0.0463994787 0.0463994775 0.0463994768  40: 0.3201767255 0.3201840912 0.3201915468 0.3201885159 0.3201870617  50: 0.0675670955 0.0637469566 0.0599878799 0.0615028942 0.0622361944  60: 0.0011449576 0.2711150754 1.0005313342 1.2945078734 0.0049027225  70: 1.2967745569 1.3284619999 1.3296922488 0.1410079704 0.5530823827  80: 0.5530384607 0.8171627721 0.0219521770 0.0184394043 0.1196398067  90: 0.0948523432 0.1541841695 0.0483069438 1.2513933889 0.3109290854  100: 0.0000671271 0.1194574394 1.2564956763 1.0428230334 0.7428865400 |

When R > 2.57, this system exhibits chaotic behavior. The system itself is ill-conditioned, so there is no way to restructure the computation to iterate the system using floating point arithmetic. Although we cannot expect our floating point algorithms to correctly handle ill-conditioned problem, we can ask that they report back an error rangle associated with the solution so that at least we are alerted to potential problems. For example, when solving linear systems of equations (see section 9.5), we can compute something called a *condition number*: this quantity can be used to bound the error of the resulting solution.

Ill-conditioning is not just a theoretical possibility. Astrophysicists have determined that our Solar System is chaotic. The trajectory of Pluto's orbit is chaotic, as in the motion of the Jovian planets, Halley's comet, and asteroidal zone trajectories. Upon a close encounter with Venus, Mercury could be ejected from the Solar System in less than 3.5 billion year!

* *Calculating special functions.* When learning calculus, we derive convergent Taylor series formulas for calculating exponential and trigometric functions. (e.g., exp(x), log(x), sin(x), cos(x), arctan(x), etc.). However, we must take great care when applying such formulas on digital computers. Not all special functions are built in to Java's Math library, so there are times when you must create your own. Give example, e.g., error function.

Numerical analysts have derived accurate, precise, and efficient algorithms for computing classic functions (e.g., hyperbolic trigometric functions, gamma function, beta function, error function, Bessel functions, Jacobian elliptic functions, spherical harmonics) that arise in scientific applications. We strongly recommend using these proven recipes instead of devising your own ad hoc routines.

**Real-world numerical catastrophes.**

 The examples we've discussed above are rather simplistic. However, these issues arise in real applications. When done incorrectly, disaster can quickly strike.

* *Ariane 5 rocket.* Ariane 5 rocket exploded 40 seconds after being launched by European Space Agency. Maiden voyage after a decade and 7 billion dollars of research and development. Sensor reported acceleration that so was large that it caused an overflow in the part of the program responsible for recalibrating inertial guidance. 64-bit floating point number was converted to a 16-bit signed integer, but the number was larger than 32,767 and the conversion failed. Unanticipated overflow was caught by a general systems diagnostic and dumped debugging data into an area of memory used for guiding the rocket's motors. Control was switched to a backup computer, but this had the same data. This resulted in a drastic attempt to correct the nonexistent problem, which separated the motors from their mountings, leading to the [end of Ariane 5](http://www.around.com/ariane.html).
* *Patriot missile accident.* On February 25, 1991 an American Patriot missile failed to track and destroy an Iraqi Scud missile. Instead it hit an Army barracks, killing 26 people. The cause was later determined to be an inaccurate calculate caused by measuring time in tenth of a second. Couldn't represent 1/10 exactly since used 24 bit floating point. Software to fix problem arrived in Dhahran on February 26. Here is more [information](http://www.ima.umn.edu/~arnold/disasters/patriot.html).
* *Intel FDIV Bug* Error in Pentium hardwire floating point divide circuit. Discovered by Intel in July 1994, rediscovered and publicized by math professor in September 1994. Intel recall in December 1994 cost $300 million. Another floating point bug discovered in 1997.
* *Sinking of Sleipner oil rig.* Sleipner A $700 million platform for producing oil and gas sprang a leak and sank in North Sea in August, 1991. Error in inaccurate finite element approximation underestimate shear stress by 47% [Reference](http://www.ima.umn.edu/~arnold/disasters/sleipner.html).
* *Vancouver stock exchange.* Vancouver stock exchange index was undervalued by over 50% after 22 months of accumulated roundoff error. The obvious algorithm is to add up all the stock prices after Instead a "clever" analyst decided it would be more efficient to recompute the index by adding the net change of a stock after each trade. This computation was done using four decimal places and truncating (not rounding) the result to three. [Reference](http://www.ualberta.ca/~carolina/CHE374/notes/U01_errors/flerrors/fl_and_errors.html)

**Lessons.**

* Use double instead of float for accuracy.
* Use float only if you really need to conserve memory, and are aware of the associated risks with accuracy. Usually it doesn't make things faster, and occasionally makes things slower.
* Be careful of calculating the difference of two very similar values and using the result in a subsequent calculation.
* Be careful about adding two quantities of very different magnitudes.
* Be careful about repeating a slightly inaccurate computation many many times. For example, calculating the change in position of planets over time.
* Designing stable floating point algorithms is highly nontrivial. Use libraries when available.

**Q + A**

**Q.** Any good references on floating point?

**A.** Here are two articles on floating point precision: [What Every Computer Scientist Should Know About Floating-Point Arithmetic](http://docs.sun.com/source/806-3568/ncg_goldberg.html) by David Goldberg and [How Java's Floating-Point Hurts Everyone Everywhere](http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf) co-authored by Turing award winner [William Kahn](http://http.cs.berkeley.edu/~wkahan/). Here's Wikipedia's entry on [Numerical analysis](http://en.wikipedia.org/wiki/List_of_numerical_analysis_topics).

**Q.** How can I convert from IEEE bit representation to double?

**A.** Here is a [decimal to IEEE converter](http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html). To get the IEEE 754 bit representation of a double variable x in Java, use Double.doubleToLongBits(x). According to[The Code Project](http://www.codeproject.com/useritems/precisefloating.asp), to get the smallest double precision number greater than x (assuming x is positive and finite) isDouble.longBitsToDouble(Double.doubleToLongBits(x) + 1).

Q. Is there any direct way to check for overflow on integer types?

A. No. The integer types do not indicate overflow in any way. Integer divide and integer remainder throw exceptions when the denominator is zero.

Q. What happens if I enter a number that is too large, e.g., 1E400?

A. Java returns the error message "floating point number too large."

Q. What about floating point types?

A. Operations that overflow evaluate to plus or minus infinity. Operations that underflows result in plus or minus zero. Operations that have no mathematically definite evaluate to NaN (not a number), e.g., 0/0, sqrt(-3), acos(3.0), log(-3.0).

Q. How do I test whether my variable has the value NaN?

A. Use the method Double.isNan. Note that NaN is unordered so the comparison operations <, >, and = involving one or two NaNs always evaluates to false. Any != comparison involving NaN evaluates to true, even (x != x), when x is NaN.

Q. What's the difference between -0.0 and 0.0?

A. Both are representations of the number zero. It is true that if x = 0.0 and y = -0.0 that (x == y). However, 1/x yields Infinity, whereas 1/y yields -Infinity. Program [NegativeZero.java](http://introcs.cs.princeton.edu/java/91float/NegativeZero.java.html) illustrates this. It gives a surprising counterexample to the misconception that if (x == y) then (f(x) == f(y)). log 0 = -infinity, log (-1) = NaN. log(ε) vs. log(-ε).

Q. I've heard that programmers should never compare two real numbers for exact equality, but you should always use a test like if |a-b| < ε or |a-b| < ε min(|a|, |b|). Is this right?

A. It depends on the situation. The relative error method is usually preferred, but it fails if the numbers are very close to zero. The absolute value method may have unintended consequences. It is not clear what value of ε to use. If a = +ε/4 and b = -ε/4, should we really consider them equal. Transitivity is not true: if a and b are "equal" and b and c are "equal", it may not be the case that a and c are "equal."

Q. What rule does Java use to print out a double?

A. By setting all the exponent bits to 1. Here's the [java.lang.Double API](http://docs.oracle.com/javase/6/docs/api/java/lang/Double.html#toString(double)). It always prints at least one digit after the decimal. After that, it uses as many digits as necessary (but no more) to distinguish the number from the nearest representable double.

Q. How are zero, infinity and NaN represented using IEEE 754?

A. By setting all the exponent bits to 1. Positive infinity = 0x7ff0000000000000 (all exponent bits 1, sign bit 0 and all mantissa bits 0), negative infinity = 0xfff0000000000000 (all exponent bits 1, sign bit 1 and all mantissa bits 0), NaN = 0x7ff8000000000000 (all exponent bits 1, at least one mantissa bit set). Positive zero = all bits 0. Negative zero = all bits 0, except sign bit which is 1.

Q. What is the exact value represented when I store 0.1 using double precision IEEE point.

A. It's 0.1000000000000000055511151231257827021181583404541015625. You can use System.out.println(new BigDecimal(0.1)); to see for yourself.

Q. What is StrictMath?

A. Java's Math library guarantees its results to be with 1 or to 2 [ulps](http://mindprod.com/jgloss/ulp.html) of the true answer. Java's StrictMath guarantees that all results are accurate to within 1/2 ulp of the true answer. A classic tradeoff of speed vs. accuracy.

Q. What is strictfp modifier?

A. It is also possible to use the strictfp modifier when declaring a class or method. This ensures that the floating point results will be bit-by-bit accurate across different JVMs. The IEEE standard allows processors to perform intermediate computations using more precision if the result would overflow. The strictfp requires that every intermediate result be truncated to 64-bit double format. This can be a major performance hit since Intel Pentium processor registers operate using IEEE 754's 80-bit double-extended format. Not many people have a use for this mode.

Q. What does the compiler flag javac -ffast-math do?

A. It relaxes some of the rounding requirements of IEEE. It makes some floating point computations faster.

Q. Are integers always represented exactly using IEEE floating point?

A. Yes, barring overflow of the 52 bit mantissa. The smallest positive integer that is not represented exactly using type double is 253 + 1 = 9,007,199,254,740,993.

Q. Does (a + b) always equal (b + a) when a and b and floating point numbers?

A. Yes.

Q. Does x / y always equal the same value, independent of my platform?

A. Yes. IEEE requires that operations (+ \* - /) are performed exactly and then rounded to the nearest floating point number (using Banker's round if there is a tie: round to nearest even number). This improves portability at the expense of efficiency.

Q. Does (x - y) always equal (x + (-y))?

A. Yes. Guaranteed by IEEE 754.

Q. Is -x always the same as 0 - x?

A. Almost, except if x = 0. Then -x = -0, but 0 - x = 0.

Q. Does (x + x == 2 \* x)? (1 \* x == x)? (0.5 \* x == x / 2)?

A. Yes, guaranteed by IEEE 754.

Q. Does x / 1000.0 always equal 0.001 \* x?

A. No. For example, if x = 1138, they are different.

Q. If (x != y), is z = 1 / (x - y) always guaranteed not to produce division by zero.

A. Yes, guaranteed by IEEE 754 (using denormalized numbers).

Q. Is (x >= y) always the same as !(x < y)?

A. No if either x or y is NaN, or both are NaN.

Q. Why not use a decimal floating point instead of binary floating point?

A. [Decimal floating point](http://www2.hursley.ibm.com/decimal/decifaq1.html) offers several advantages over binary floating point, especially for financial computations. However, it usually requires about 20% more storage (assuming it is stored using binary hardware) and the resulting code is somewhat slower.

Q. Why not use a fixed point representation instead of a floating point? floating point?

A. Fixed point numbers have a fixed number of digits after the decimal place. Can be represented using integer arithmetic. Floating point has a sliding window of precision, which provides a large dynamic range and high precision. Fixed point numbers are used on some embedded hardware devices that do not have an FPU (to save money), e.g., audio decoding or 3D graphics. Appropriate when data is constrained in a certain range.

Q. Any good resources for numerics in Java?

A. [Java numerics](http://math.nist.gov/javanumerics/) provides a focal point for information on numerical computing in Java. [JSci](http://jsci.sourceforge.net/api/index.html): a free science API for numerical computing in Java.

Q. Why are Java's Math.sin and Math.cos functions slower than their C counterparts?

A. In the range -pi/4 to pi/4, Java uses the hardware sin and cos functions, so they take about the same time as in C. Arguments outside this range must be converted to this range by taking them modulo pi. As James Gosling [blogs](http://blogs.sun.com/roller/page/jag?entry=transcendental_meditation) on x87 platform, the hardware sin and cos use an approximation which makes the computation fast, but it is not of the accuracy required by IEEE. C implementation typically use the hardware sin and cos for all arguments, trading off IEEE conformance for speed.

**Exercises**

1. What is the result of the following code fragment?

|  |
| --- |
| double a = 12345.0;  double b = 1e-16;  System.out.println(a + b == a); |

1. List all representable numbers for 6-bit floating point numbers with 1 sign bit, 3 exponent bits, and 2 mantissa bits. What is the result of the following code fragment?
2. How many values does the following code fragment print?

|  |
| --- |
| for (double d = 0.1; d <= 0.5; d += 0.1)  System.out.println(d);  for (double d = 1.1; d <= 1.5; d += 0.1)  System.out.println(d); |

*Answer*: nine (five in the first loop and four in the second loop). The output is:

|  |
| --- |
| 0.1  0.2  0.30000000000000004  0.4  0.5  1.1  1.2000000000000002  1.3000000000000003  1.4000000000000004 |

1. What numbers are printed out when executing the following code fragment with N = 25?

|  |
| --- |
| for (int i = 0; i < N; i++) {  int counter = 0;  for (double x = 0.0; x < i; x += 0.1)  counter++;  if (counter != 10\*i) System.out.print(i + " ");  } |

*Answer*: 1 5 6 7 8 9 10 11 12 13 14 15 16 17 18. Unlikely you would predict this without typing it in. Avoid using floating point numbers to check loop continuation conditions.

1. What is the result of the following code fragment?

|  |
| --- |
| for (double t = 0.0; t < 100.0; t += 0.01)  System.out.println(t); |

*Answer:* pretty much what you expect, but there is a lot of noise in the least significant two or three digits.

1. What does the following code fragment print?

|  |
| --- |
| System.out.println(0.9200000000000002);  System.out.println(0.9200000000000001); |

Prints out 0.9200000000000002 twice!

1. Find a value of x for which (0.1 \* x) is different from (x / 10).
2. Find a real number a such that (a \* (3.0 / a)) doesn't equal 3.0. *Answer:* If a = 0, the first expression evaluates to NaN.
3. Find a real number a such that (Math.sqrt(a) \* Math.sqrt(a)) doesn't equal a. *Answer:* a = 2.0.
4. Find floating point values such that (x/x != 1), (x - x != 0), (0 != 0 \* x). *Answer:* consider values of x that are 0, +- infinity, or NaN.
5. Find floating point values x and y such that x >= y is true, but !(x < y) is false.

*Hint*: consider cases where either x or y (or both) have the value NaN.

1. What is the result of the following code fragment from [FloatingPoint.java](http://introcs.cs.princeton.edu/java/91float/FloatingPoint.java.html)?

|  |
| --- |
| float f = (float) (3.0 / 7.0);  if (f == 3.0 / 7.0) System.out.println("yes");  else System.out.println("no"); |

1. Examples from *How Reliable are Results of Computers* by S. M. Rump. See program [Rump.java](http://introcs.cs.princeton.edu/java/91float/Rump.java.html).
   * Compute a\*a - 2\*b\*b where a = 665857 and b = 470832 are of type float. Then compute a\*a - 2.0\*b\*b. Repeat with type double. Java answers with float: 0.0 and 11776.0. Java answers with double: 1.0. Exact answer: 1.0.
   * Compute 9x4 - y4 + 2y2 where x = 10864 and y = 18817 are of type double. Java answer: 2.0. Exact answer: 1.0.
   * Compute p(x) = 8118x4 - 11482x3 + x2 + 5741x - 2030 where x = 0.707107 is of type float and double. Java answer for float: 1.2207031E-4. Java answer for double: -1.9554136088117957E-11. Exact answer: -1.91527325270... \* 10^-7.
2. Do you think using the type double would be a good idea for storing stock prices that are always specified in increments of 1/64. Explain why or why not.*Answer*: yes, it would be perfectly fine since such values are represented exactly using binary floating point. If the stock prices were represented in decimal, e.g., 45.10, that could result in roundoff error.
3. Fix [Exponential.java](http://introcs.cs.princeton.edu/java/91float/Exponential.java.html) so that it works properly for negative inputs using the method described in the text.
4. Sum the following real numbers from left to right. Use floating point arithmetic, and assume your computer only stores three decimal digits of precision.

|  |
| --- |
| 0.007 0.103 0.205 0.008 3.12 0.006 0.876 0.005 0.015 |

Repeat using the priority queue algorithm. Repeat using Kahan's algorithm.

1. Write a program to read in a list of bank account values and print the average value, rounded exactly to the nearest penny using banker's rounding. The input values will be separeated by whitespace and using two decimal digits. Hint: avoid Double.parseDouble.

|  |
| --- |
| 100.00 489.12 1765.12 3636.10  3222.05 3299.20 5111.27 3542.25  86369.18 532.99 |

1. What does the following [code fragment](http://introcs.cs.princeton.edu/java/91float/Equality.java.html) print?

|  |
| --- |
| long x = 16777217; // 2^24 + 1  System.out.println(x);  float y = 16777217;  DecimalFormat df = new DecimalFormat("0.00000000000");  System.out.println(df.format(y)); |

*Answer*: 16777217 and 16777216.00000000000. 2^24 + 1 is the smallest positive integer that is not exactly representable using type float.

1. Can you find values a, b, and c such that Math.sqrt(b\*b - a\*c) is invalid (NaN) but (b\*b < a\*c) is false? Kahan.
2. What is wrong with each of the following [two loops](http://introcs.cs.princeton.edu/java/91float/FloatingLoop.java.html)?

|  |
| --- |
| int count1 = 0;  for (float x = 0.0f; x < 20000000.0f; x = x + 1.0f)  count1++;  System.out.println(count1);  int count2 = 0;  for (double x = 0.0; x < 20000000.0; x = x + 1.0)  count2++;  System.out.println(count2); |

*Answer*: The first code fragment goes into an infinite loop when x = 16777216.0f since 16777217.0f cannot be represented exactly using type float and 16777216.0f + 1.0f = 16777216.0f. The second loop is guaranteed to work correctly, but using a variable of type int for the loop variable might be more efficient.

1. One way to define e is the limit as n approaches infinity of (1 + 1/n)n. Write a computer program to estimate e using this formula. For which value of n do you get the best approximation of e? *Hint*: if n gets too large, then 1 + 1/n is equal to 1 using floating point arithmetic.
2. What's wrong with the following implementation of Math.max().

|  |
| --- |
| public static double max(double x, double y) {  if (x >= y) return x;  return y;  } |

*Answer*: double not handle NaN properly. According to this definition max(0, NaN) is NaN but max(NaN, 0) is 0. It should be NaN.

1. Not all subtraction is catastrophic. Demonstrate that the expression x2 - y2 exhibits catastrophic cancellation when x is close to y. However, the improved expression (x + y)(x - y) still subtracts nearly equal quantities, but it is a benign cancellation.
2. (Goldberg) (1 + i/n)^n arises in financial calculations involving interest. Rewrite as e^(n ln (1 + i/n)). Now trick is to compute ln(1+x). Math.log(1+x) not accurate when x << 1.

|  |
| --- |
| if (1 + x == 1) return x  else return (x \* Math.log(1 + x)) / (1 + x) - 1. |

Alternative in Java 1.5: Math.log1p(x)

1. To compute e^x - 1, use Math.expm1(x). For values of x near 0, Math.expm1(x) + 1 is more accurate than Math.exp(x).
2. Catastrophic cancellation: f(x) = ex - sin x - cos x near x = 0. Use x2 + x3/3 as an approximation near x = 0.
3. Catastrophic cancellation: f(x) = ln(x) - 1. For x near e use ln(x/e).
4. Catastrophic cancellation: f(x) = ln(x) - log(1/x). For x near 1 use 2 ln(x).
5. Catastrophic cancellation: x-2(sinx - ex + 1).
6. Find a value of x for which Math.abs(x) doesn't equal Math.sqrt(x\*x).
7. **Numerical stability.** The *golden mean* φ = sqrt(5)/2 - 1/2 = 0.61803398874989484820... It satisfies the equation φN = φN-2 - φN-1. We could use this recurrence to compute power of φ by starting with phi0 = 1, phi1 = φ, and iterating the recurrence to calculate successive powers of φ. However, floating-point roundoff error swamps the computation by about N = 40. Write a program [Unstable.java](http://introcs.cs.princeton.edu/java/91float/Unstable.java.html) that reads in a command line parameter N and prints out φNcomputed using the recurrence above and also using Math.pow(). Run your program with N = 40 and N = 100 to see the consequences of roundoff error.

**Creative Exercises**