Java theory and practice: Where's your point?

Tricks and traps with floating point and decimal numbers

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

01 January 2003

Many programmers go their entire career without using fixed point or floating point numbers, with the possible exception of the odd timing test or benchmark. The Java language and class libraries support two sorts of non-integral numeric types -- IEEE 754 floating point (`float` and `double`, and the wrapper classes `Float` and `Double`), as well as arbitrary-precision decimal (`java.math.BigDecimal`.) In this month's *Java theory and practice*, Brian Goetz looks at some of the traps and "gotchas" often encountered when using non-integral numeric types in Java programs.

View more content in this series

While nearly every processor and programming language supports floating point arithmetic, most programmers pay little attention to it. This is understandable -- most of us rarely require the use of non-integral numeric types. With the exception of scientific computing and the occasional timing test or benchmark, it just doesn't come up. The arbitrary-precision decimal numbers provided by `java.math.BigDecimal` are similarly ignored by most developers -- the vast majority of applications have no use for them. However, the vagaries of representing non-integral numbers do occasionally sneak into otherwise integer-centric programs. For example, JDBC uses `BigDecimal` as the preferred interchange format for SQL `DECIMAL` columns.

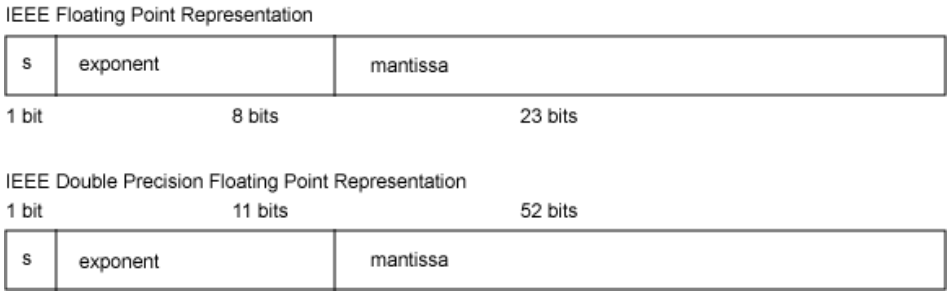
IEEE floating point

The Java language supports two primitive floating point types: `float` and `double`, and their wrapper class counterparts, `Float` and `Double`. These are based on the IEEE 754 standard, which defines a binary standard for 32-bit floating point and 64-bit double precision floating point binary-decimal numbers.

IEEE 754 represents floating point numbers as base 2 decimal numbers in scientific notation. An IEEE floating point number dedicates 1 bit to the sign of the number, 8 bits to the exponent, and 23 bits to the mantissa, or fractional part. The exponent is interpreted as a signed integer, allowing negative as well as positive exponents. The fraction is represented as a binary (base 2) decimal,

meaning the highest-order bit corresponds to a value of Â½ (2^{-1}), the second bit Â¼ (2^{-2}), and so on. For double-precision floating point, 11 bits are dedicated to the exponent and 52 bits to the mantissa. The layout of IEEE floating point values is shown in Figure 1.

Figure 1. IEEE 754 floating point layout



Because any given number can be represented in scientific notation in multiple ways, floating point numbers are normalized so that they are represented as a base 2 decimal with a 1 to the left of the decimal point, adjusting the exponent as necessary to make this requirement hold. So, for example, the number 1.25 would be represented with a mantissa of 1.01 and an exponent of 0: `(-1)`

The number 10.0 would be represented with a mantissa of 1.01 and an exponent of 3: `(-1)`

Special numbers

In addition to the standard range of values permitted by the encoding (from 1.4e-45 to 3.4028235e +38 for `float`), there are special values that represent infinity, negative infinity, `-0`, and NaN (which stands for "not a number"). These values exist so that error conditions such as arithmetic overflow, taking the square root of a negative number, and dividing by `0` can yield a result that can be represented within the floating point value set.

These special numbers have some unusual characteristics. For example, `0` and `-0` are distinct values, but when compared for equality, are considered equal. Dividing a nonzero number by infinity yields `0`. The special number NaN is unordered; any comparison between NaN and other floating point values using the `==`, `<`, and `>` operators will yield `false`. Even `(f == f)` will yield `false` if `f` is NaN. If you want to compare a floating point value with NaN, use the `Float.isNaN()` method instead. Table 1 shows some of the properties of infinity and NaN.

Table 1. Properties of special floating point values

Expression	Result
`Math.sqrt(-1.0)`	`-> NaN`
`0.0 / 0.0`	`-> NaN`
`1.0 / 0.0`	`-> Infinity`
`-1.0 / 0.0`	`-> -Infinity`
`NaN + 1.0`	`-> NaN`

`Infinity + 1.0`	`-> Infinity`
`Infinity + Infinity`	`-> Infinity`
`NaN > 1.0`	`-> false`
`NaN == 1.0`	`-> false`
`NaN < 1.0`	`-> false`
`NaN == NaN`	`-> false`
`0.0 == -0.01`	`-> true`

Primitive float type and wrapper class float have different comparison behavior

To make matters worse, the rules for comparing NaN and `-0` are different between the primitive `float` type and the wrapper class `Float`. For `float` values, comparing two NaN values for equality will yield `false`, but comparing two NaN `Float` objects using `Float.equals()` will yield `true`. The motivation for this is that otherwise it would be impossible to use an NaN `Float` object as a key in a `HashMap`. Similarly, while `0` and `-0` are considered equal when represented as float values, comparing `0` and `-0` as `Float` objects using `Float.compareTo()` indicates that `-0` is considered to be less than `0`.

Floating point hazards

Because of the special behavior of infinity, NaN, and `0`, certain transformations and optimizations that may appear harmless are actually incorrect when applied to floating point numbers. For example, while it may seem obvious that `0.0-f` and `-f` are equivalent, this is not true when `f` is `0`. There are other similar gotchas, some of which are shown in Table 2.

Table 2. Invalid floating point assumptions

This expression...	isn't necessarily the same as this...	when...
`0.0 - f`	`-f`	f is `0`
`f < g`	`! (f >= g)`	f or g is NaN
`f == f`	`true`	f is NaN
`f + g - g`	`f`	g is infinity or NaN

Rounding errors

Floating point arithmetic is rarely exact. While some numbers, such as `0.5`, can be exactly represented as a binary (base 2) decimal (since `0.5` equals 2^{-1}), other numbers, such as `0.1`, cannot be. As a result, floating point operations may result in rounding errors, yielding a result that is close to -- but not equal to -- the result you might expect. For example, the simple calculation below results in `2.600000000000001`, rather than `2.6`:

```
double s=0;

for (int i=0; i<26; i++)
  s += 0.1;
System.out.println(s);
```

Similarly, multiplying `.1*26` yields a result different from that of adding `.1` to itself 26 times. Rounding errors become even more serious when casting from floating point to integer, because casting to an integral type discards the non-integral portion, even for calculations that "look like" they should have integral values. For example, the following statements:

```
double d = 29.0 * 0.01;
System.out.println(d);
System.out.println((int) (d * 100));
```

will produce as output:

```
0.29
28
```

which is probably not what you might expect at first.

Guidelines for comparing floating point numbers

Because of the unusual comparison behavior of NaN, and the rounding errors that are virtually guaranteed in nearly all floating point calculations, interpreting the results of the comparison operators on floating point values is tricky.

It would be best to try to avoid floating point comparison entirely. This is, of course, not always possible, but you should be aware of the limitations of floating point comparison. If you must compare floating point numbers to see if they are the same, you should instead compare the absolute value of their difference with some pre-chosen epsilon value, so you are instead testing whether they are "close enough." (If you don't know the scale of the underlying measurements, using the test "abs(a/b - 1) < epsilon" is likely to be more robust than simply comparing the difference.) Even testing a value to see if it is greater than or less than zero is risky -- calculations that are "supposed to" result in a value slightly greater than zero may in fact result in numbers that are slightly less than zero due to accumulated rounding errors.

The non-ordered nature of NaN adds further opportunities for error when comparing floating point numbers. A good rule of thumb for steering clear of many of the gotchas surrounding infinity and NaN when comparing floating point numbers is to test a value explicitly for validity, rather than trying to exclude invalid values. In Listing 1, there are two possible implementations of a setter for a property that can only take on non-negative values. The first will accept NaN, the second will not. The second form is preferable because it tests explicitly for the range of values you consider to be valid.

Listing 1. Better and worse ways to require a float value be non-negative

```
// Trying to test by exclusion -- this doesn't catch NaN or infinity
public void setFoo(float foo) {
  if (foo < 0)
      throw new IllegalArgumentException(Float.toString(f));
    this.foo = foo;
}

// Testing by inclusion -- this does catch NaN
public void setFoo(float foo) {
  if (foo >= 0 && foo < Float.INFINITY)
    this.foo = foo;
  else
    throw new IllegalArgumentException(Float.toString(f));
}
```

Don't use floating point numbers for exact values

Some non-integral values, like dollars-and-cents decimals, require exactness. Floating point numbers are not exact, and manipulating them will result in rounding errors. As a result, it is a bad idea to use floating point to try to represent exact quantities like monetary amounts. Using floating point for dollars-and-cents calculations is a recipe for disaster. Floating point numbers are best reserved for values such as measurements, whose values are fundamentally inexact to begin with.

Big decimals for small numbers

Since JDK 1.3, Java developers have another alternative for non-integral numbers: `BigDecimal`. `BigDecimal` is a standard class, with no special support in the compiler, to represent arbitrary-precision decimal numbers and perform arithmetic on them. Internally, `BigDecimal` is represented as an arbitrary-precision unscaled value and a scale factor, which represents how many places to move the decimal point left to obtain the scaled value. Thus, the number represented by the `BigDecimal` is `unscaledValue*10`$^{-scale}$.

Arithmetic on `BigDecimal` values is provided by methods for addition, subtraction, multiplication, and division. Because `BigDecimal` objects are immutable, each of these methods produces a new `BigDecimal` object. As a result, `BigDecimal` is not well-suited for intensive numeric calculation because of the object creation overhead, but it is designed for representing exact decimal numbers. If you are looking to represent exact quantities such as monetary amounts, `BigDecimal` is well-suited to the task.

All equals methods are not created equal

Like floating point types, `BigDecimal` also has a few quirks. In particular, be careful about using the `equals()` method to test for numeric equality. Two `BigDecimal` values that represent the same number, but have different scale values (for instance, `100.00` and `100.000`) will not be considered equal by the `equals()` method. However, the `compareTo()` method will consider them to be equal, so you should use `compareTo()` instead of `equals()` when comparing two `BigDecimal` values numerically.

There are some cases where arbitrary-precision decimal arithmetic is still not sufficient to maintain exact results. For example, dividing `1` by `9` yields the infinite repeating decimal `.111111...` For this

reason, `BigDecimal` gives you explicit control over rounding when performing division operations. Exact division by a power of ten is supported by the `movePointLeft()` method.

Use BigDecimal as an interchange type

SQL-92 includes a `DECIMAL` data type, which is an exact numeric type for representing fixed point decimal numbers and performs basic arithmetic operation on decimal numbers. Some SQL dialects prefer to call this type `NUMERIC`, and others also include a `MONEY` data type, which is defined as a decimal number with two places to the right of the decimal.

If you want to store a number to a `DECIMAL` field in a database, or retrieve a value from a `DECIMAL` field, how do you ensure that the number will be transmitted exactly? You don't want to use the `setFloat()` and `getFloat()` methods provided by the JDBC `PreparedStatement` and `ResultSet` classes, since the conversion between floating point and decimal may cause exactness to be lost. Instead, use the `setBigDecimal()` and `getBigDecimal()` methods of `PreparedStatement` and `ResultSet`.

Similarly, XML data binding tools like Castor will generate getters and setters for decimal-valued attributes and elements (which are supported as a basic data type in the XSD schema) using `BigDecimal`.

Constructing BigDecimal numbers

There are several constructors available for `BigDecimal`. One takes a double-precision floating point as input, another takes an integer and a scale factor, and another takes a `String` representation of a decimal number. You should be careful with the `BigDecimal(double)` constructor because it can allow rounding errors to sneak into your calculations before you know it. Instead, use the integer or `String`-based constructors.

Improper use of the `BigDecimal(double)` constructor can show up as strange-seeming exceptions in JDBC drivers when passed to the JDBC `setBigDecimal()` method. For example, consider the following JDBC code, which wants to store the number `0.01` into a decimal field:

```
PreparedStatement ps =
  connection.prepareStatement("INSERT INTO Foo SET name=?, value=?");
ps.setString(1, "penny");
ps.setBigDecimal(2, new BigDecimal(0.01));
ps.executeUpdate();
```

Depending on your JDBC driver, this seemingly harmless code can throw some confusing exceptions when executed because the double-precision approximation of `0.01` will result in a large scale value, which may confuse the JDBC driver or database. The exception will originate in the JDBC driver, but will probably give little indication as to what is actually wrong with your code, unless you are aware of the limitations of binary floating point numbers. Instead, construct the `BigDecimal` using `BigDecimal("0.01")` or `BigDecimal(1, 2)` to avoid this problem, since either of these will result in an exact decimal representation.

Summary

Using floating point and decimal numbers in Java programs is fraught with pitfalls. Floating point and decimal numbers are not nearly as well-behaved as integers, and you cannot assume that floating point calculations that "should" have integer or exact results actually do. It is best to reserve the use of floating point arithmetic for calculations that involve fundamentally inexact values, such as measurements. If you need to represent fixed point numbers, such as dollars and cents, use `BigDecimal` instead.

Resources

- David Goldberg's classic paper, What Every Scientist Should Know About Floating-Point Arithmetic, explores the tradeoffs and limitations of various floating point representations.
- Bill Venners looks at floating point support in the JVM in his Under the hood column on *JavaWorld*.
- William Kahan, one of the principal architects of IEEE 754, critiques the incomplete Java floating point implementation in "How Java's Floating-Point Hurts Everyone Everywhere" (PDF).
- The Intel Architecture Software Developer's Manual (PDF) contains detailed information about the most common implementation of IEEE 754 floating point.
- Steve Hollasch of Microsoft offers a nice summary of IEEE Standard 754 floating point numbers.
- The Castor XML data binding framework uses `BigDecimal` as an interchange type for decimal numbers.
- The *JavaWorld* article "Make cents with BigDecimal" offers some sensible tips for performing financial calculations with `BigDecimal` and `NumberFormat`.

About the author

Brian Goetz

Brian Goetz is a software consultant and has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California. See Brian's published and upcoming articles in popular industry publications.