Chapter 1

Fun with Floats

Floats are inexact by nature and this can confuse programmers. In this chapter we present an introduction to this problem. The basic message is that Floats are what they are inexact but fast numbers.

1.1 Never test equality on floats

The first basic principle is to never compare float equality. Let's take a simple case: the addition of two floats may not be equal to the float representing their sum. For example 0.1 + 0.2 is not equal to 0.3.

```
(0.1 + 0.2) = 0.3
   returns false
```

Hey, this is unexpected, you did not learn that in school, did you? This behavior is surprising indeed, but it's normal since floats are inexact numbers. What is important to understand is that the way floats are printed is also impacting our understanding. Some approaches prints a simpler representation of reality than others. In early versions of Pharo printing 0.1 + 0.2 was printing 0.3, now printing it returns 0.30000000000000004. This change was guided by the idea that it is better not to lie to the user. Showing the inexactness of float is better than hiding because one day or another we can be deeply bitten by them.

```
(0.2 + 0.1 ) printString
   returns '0.30000000000000004'

0.3 printString
   returns '0.3'
```

We can see that we are in presence of two different numbers by looking at the hexadecimal values.

```
(0.1+0.2) hex
   returns '3FD3333333333334'
0.3 hex
   returns '3FD3333333333333'
```

The method storeString also conveys that we are in presence of two different numbers.

```
(0.1+0.2) storeString
   returns    '0.30000000000000004'
0.3 storeString
   returns '0.3'
```

About closeTo:. One way to know if two floats are probably close enough to look like the same number is to use the message closeTo:

```
(0.1 + 0.2) closeTo: 0.3
   returns true

0.3 closeTo: (0.1 + 0.2)
   returns true
```

About Scaled Decimals. Scaled Decimals are exact numbers so they exhibit the behavior you expected.

```
0.1s2 + 0.2s2 = 0.3s2
   returns true
```

Analyzing more deeply above example

To understand what operation is involved in above addition, we must know how Float are represented internally in the computer: Pharo's Float format is a wide spread standard found on most computers - IEEE 754-1985 double precision on 64 bits (See http://en.wikipedia.org/wiki/IEEE_754-1985 for more details). In this format, a Float is represented in base 2 by this formula:

$$sign \cdot mantissa \cdot 2^{exponent}$$

- The sign is represented on 1 bit.
- The exponent is represented on 11 bits.

- The mantissa is a fractional number in base two, with a leading 1 before decimal point, and with 52 binary digits after fraction point.
  For example, a serie of 52 bits:
  0110010000000000000000000000000000000000000000000000
  means the mantissa is:
  1.0110010000000000000000000000000000000000000000000000
  which also means

  $$1 + \frac{0}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{0}{2^5} + \frac{1}{2^6} + \cdots + \frac{0}{2^{52}}$$

  The mantissa is thus between 1 (included) and 2 (excluded) for normal numbers.

Let us construct such a mantissa:

```
(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat.
   returns 1.390625
```

Now let us multiply by 2^3 to get a non null exponent:

```
(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat timesTwoPower: 3.
   returns 11.125
```

In Pharo, you can retrieve these informations:

```
11.125 sign.
   returns 1

11.125 significand.
   returns 1.390625

11.125 exponent.
   returns 3
```

Generally you won't handle the mantissa, but rather the mantissa shifted 52 bits to the left in order to operate on exact integers. The result includes the leading 1 and should thus be 53 bits long for a normal number (that's the float precision):

```
11.125 significandAsInteger printStringBase: 2.
   returns '10110010000000000000000000000000000000000000000000000'

11.125 significandAsInteger highBit.
   returns 53

Float precision.
   returns 53
```

You can also retrieve the exact fraction corresponding to the internal representation of the Float:

```
11.125 asTrueFraction.
  returns  (89/8)

(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) * (2 raisedTo: 3).
  returns  (89/8)
```

Until there we retrieved the exact input we've injected into the Float. Are Float operations exact after all? Hem, no, we only played with fractions having a power of 2 as denominator and a few bits in numerator. If one of these conditions is not met, we won't find any exact Float representation of our numbers. In particular, it is not possible to represent 1/5 with a finite number of binary digits. Consequently, a decimal fraction like 0.1 cannot be represented exactly with above representation.

```
(1/5) asFloat = (1/5).
  returns false

(1/10) = 0.1
  returns false
```

Let us see in detail how we could get the fractional bits of 2r1 / 2r101: we must pose the division:

```
1          | 101
10         |-----------
100        | 0.00110011
1000       |
-101       |
  11       |
  110      |
 -101      |
    1      |
    10     |
    100    |
    1000   |
   -101    |
      11   |
      110  |
     -101  |
        1  |
```

What we see is that we get a cycle: every 4 Euclidean divisions, we get a quotient 2r0011 and a remainder 1. That means that we need an infinite serie of this bit pattern 0011 to represent 1/5 in base 2. Let us see how Pharo dealt to convert (1/5) to a Float:

```
(1/5) asFloat significandAsInteger printStringBase: 2.
   returns '11001100110011001100110011001100110011001100110011010'

(1/5) asFloat exponent.
   returns −3
```

That's the bit pattern we expected, except the last bits 001 have been rounded to upper 010. This is the default rounding mode of Float, round to nearest even. We now know why 0.2 is represented inexactly in machine. It's the same mantissa for 0.1, and its exponent is −4.

So, when we entered 0.1 + 0.2, we didn't get exactly (1/10) + (1/5). Instead of that we got:

```
0.1 asTrueFraction + 0.2 asTrueFraction.
   returns  (10808639105689191/36028797018963968)
```

But that's not all the story... Let us inspect above fraction:

```
10808639105689191 printStringBase: 2.
   returns  '100110011001100110011001100110011001100110011001100111'

10808639105689191 highBit.
   returns  54

10808639105689191 lowBit.
   returns  1

36028797018963968 printStringBase: 2.
   returns  '100000000000000000000000000000000000000000000000000000000'
```

The denominator is a power of 2 as we expect, but we need 54 bits of precision to store the numerator... Float only provides 53. There will be another rounding error to fit into Float representation:

```
(0.1 asTrueFraction + 0.2 asTrueFraction) asFloat = (0.1 asTrueFraction + 0.2
      asTrueFraction).
   returns false

(0.1 asTrueFraction + 0.2 asTrueFraction) asFloat significandAsInteger.
   returns '10011001100110011001100110011001100110011001100110100'
```

To summarize what happened, including conversions of decimal representation to Float representation:

(1/10) asFloat	0.1	inexact (rounded to upper)
(1/5) asFloat	0.2	inexact (rounded to upper)
(0.1 + 0.2) asFloat	$\cdots$	inexact (rounded to upper)

3 inexact operations occurred, and, bad luck, the 3 rounding operations were all to upper, thus they did cumulate rather than annihilate. On the other side, interpreting 0.3 is causing a single rounding error (3/10) asFloat. We now understand why we cannot expect 0.1 + 0.2 = 0.3.

As an exercise, you could show why 1.3 * 1.3 $\neq$ 1.69.

1.2 With floats, printing is inexact

One of the biggest trap we learned with above example is that despite the fact that 0.1 is printed '0.1' as if it were exact, it's not. The name absPrintExactlyOn:base: used internally by printString is a bit confusing, it does not print exactly, but it prints the shortest decimal representation that will be rounded to the same Float when read back (Pharo always converts the decimal representation to the nearest Float).

Another message exists to print exactly, you need to use printShowingDecimalPlaces: instead. As every finite Float is represented internally as a Fraction with a denominator being a power of 2, every finite Float has a decimal representation with a finite number of decimals digits (just multiply numerator and denominator with adequate power of 5, and you'll get the digits). Here you go:

```
0.1 asTrueFraction denominator highBit.
  returns 56
```

This means that the fraction denominator is 2^{55} and that you need 55 decimal digits after the decimal point to really print internal representation of 0.1 exactly.

```
0.1 printShowingDecimalPlaces: 55.
  returns '0.1000000000000000055511151231257827021181583404541015625'
```

And you can retrieve the digits with:

```
0.1 asTrueFraction numerator * (5 raisedTo: 55).
  returns 1000000000000000055511151231257827021181583404541015625
```

You can just check our result with:

```
1000000000000000055511151231257827021181583404541015625/(10 raisedTo: 55) =
      0.1 asTrueFraction
  returns true
```

You see that printing the exact representation of what is implemented in machine would be possible but would be cumbersome. Try printing 1.0e−100 exactly if not convinced.

1.3 Another example: rounding is inexact

While float equality is known to be evil, you have to pay attention to other aspects of floats. Let us illustrate that point with the following example.

```
2.8 truncateTo: 0.01
   returns 2.8000000000000003

2.8 roundTo: 0.01
   returns 2.8000000000000003
```

It is surprising but not false that `2.8 truncateTo: 0.01` does not return `2.8` but `2.8000000000000003`. This is because `truncateTo:` and `roundTo:` perform several operations on floats: inexact operations on inexact numbers can lead to cumulative rounding errors as you saw above, and that's just what happens again.

Even if you perform the operations exactly and then round to nearest Float, the result is inexact because of the initial inexact representation of `2.8` and `0.01`.

```
(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat
   returns 2.8000000000000003
```

Using `0.01s2` rather than `0.01` let this example appear to work:

```
2.80 truncateTo: 0.01s2
   returns  2.80s2

2.80 roundTo: 0.01s2
   returns  2.80s2
```

But it's just a case of luck, the fact that `2.8` is inexact is enough to cause other surprises as illustrated below:

```
2.8 truncateTo: 0.001s3.
   returns 2.799s3

2.8 < 2.800s3.
   returns true
```

Truncating in the Float world is absolutely unsafe. Though, using a ScaledDecimal for rounding is unlikely to cause such discrepancy, except when playing with last digits.

1.4 Fun with Inexact representations

To add a nail to the coffin, let's play a bit more with inexact representations. Let us try to see the difference between different numbers:

```
{
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) − (2.8 predecessor)) abs −> −1.
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) − (2.8)) abs −> 0.
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) − (2.8 successor)) abs −> 1.
} detectMin: [:e | e key ]

returns the pair
   0.0−>1
```

you get `0.0−>1`, which means that: `(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat = (2.8 successor)`

But remember that

```
(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) ~= (2.8 successor)
```

It must be interpreted as the nearest Float to `(2.8 asTrueFraction roundTo: 0.01 asTrueFraction)` is `(2.8 successor)`.

If you want to know how far it is, then get an idea with:

```
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) − (2.8 successor asTrueFraction))
      asFloat
   returns −2.0816681711721685e−16
```

1.5 Conclusion

Floats are inexact numbers. Pay attention when you manipulate them. There are much more things to know about floats, and if you are advanced enough, it would be a good idea to check this link from the wikipedia page "What Every Computer Scientist Should Know About Floating-Point Arithmetic".