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
```

This behavior is 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 13/10

1.3 is represented in machine as

```
(1.3 significandAsInteger printStringRadix: 2) , '.0e' , (1.3 exponent
− Float precision + 1) printString.
−> '2r10100110011001100110011001100110011001100110011001101.0e−52'
```

Or if you prefer:

```
(1.3 asTrueFraction numerator printStringBase: 2) , '/' , (1.3
asTrueFraction denominator printStringBase: 2).
−> '10100110011001100110011001100110011001100110011001101/
10000000000000000000000000000000000000000000000000000'
```

As you can see, this is quite different from 13/10. However, you can test (13/10) asFloat = 1.3 and that happens to be true, but that won't always be true. In particular the inverse it not. So we are sure that if you compare floats using simple equality there is a high chance that you will get burned back. Again scaled decimal return correct and consistent behavior.

```
(13/10) asFloat = 1.3
   returns true

1.3 = (13/10).
   returns false

1.3s1 = (13/10).
   returns true

1.3s2*1.3s2 = 1.69s2.
   returns true

1.3 * 1.3 = 1.69.
   returns false

1.3 * 1.3 closeTo: 1.69
   returns true
```

1.2 Study of a simple example

While float equality is known to 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. Why because 2.8 is an inexact number and truncateTo: and roundTo: perform several operations on floats.

As soon as you write 0.01 instead of 1/100 or 0.01s2, the worm is in the fruit. Note that the same behavior can arrive with more elaborated numbers as 2.8011416510246336.

```
2.8011416510246336 roundTo: 0.01
   returns 2.8000000000000003
```

This again happens even if performed exactly (then rounded to nearest Float) (2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat

Once more, Floats are inexact

Stéf ►*should continue from here*◄

```
0.01 ∼= 0.01s2
```

The name absPrintExactlyOn:base: is a bit confusing, it does not print exactly, but it prints the shortest decimal representation than will be rounded to the same Float when read back.

To print it 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).

So try:

```
0.01 printShowingDecimalPlaces: 59
   returns 0.01000000000000000020816681711721685132943093776702880859375
```

You see that even if you try to execute the operation without rounding error, then convert it back to Float, you get the error:

```
(2.8011416510246336 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat
   returns 2.8000000000000003
```

When you perform the roundTo: operations in Float inexact arithmetic, you may accumulate more rounding errors, so the result may vary.

If you want to round to an exact hundredth, then use exact arithmetic and try:

```
2.8011416510246336 roundTo: 0.01s2
   returns 2.80s2
```

1.3 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
```

1.4 Conclusion

Floats are inexact numbers. Pay attention when you manipulate them.