Chapter 1

Fun with Floats

Hope it helps to consider Float for what they are, inexact but fast. Don't put too much expectations on them.

1.1 Never test equality on floats

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

This behavior is normal since floats are inexact numbers. We can see that we are in presence of two different numbers

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

The method storeString conveys that

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

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

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

Analysing 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).
−> '
    10100110011001100110011001100110011001100110011001101/1000000000000000000000000
    '
```

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.

```
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
```

Stéf ►*add a word on closeTo:*◄

1.2 Study of a simple example

```
2.8011416510246336 roundTo: 0.01
−> 2.8000000000000003

2.8 truncateTo: 0.01
−> 2.8000000000000003
```

Yes nice example to exhibit in school

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

As soon as you write 0.01 instead of (1/100) or (0.01s2), the worm is in the fruit.

Once more, Floats are inexact

```
0.01 ~= 0.01s2
```

The name absPrintExactlyOn:base: is lying, 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: indeed. As every finite Float is a 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
-> 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
->  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
```

1.3 Fun with Inexact representations

Pour enfoncer le clou, let's play a bit more with inexact representations:

```
{
((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
   0.0->1
```

you get `0.0->1`, which mean 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
-2.0816681711721685e-16
```

1.4 Conclusion