

Benchmarking in Pharo

Damien Cassou, Stéphane Ducasse and Luc Fabresse

W5S09



<http://www.pharo.org>



Common Wisdom

If you did not profile your code you may have 40-50% speed up waiting for you.



Measuring Execution Speed

We create an expression and use [expression] timeToRun

```
[ 1000 factorial ] timeToRun
```

Comparing Two Executions

Is `select:, then collect:` **slower than** `select:thenCollect:`?



timeToRun example

```
| coll |  
coll := #(1 2 3 4 5 6 7 8 9 10) asOrderedCollection.  
[ 1000000 timesRepeat: [  
  (coll select: [:each | each > 5]) collect: [:i | i * i ] ]  
] timeToRun  
> "0:00:00:00.517"
```

```
| coll |  
coll := #(1 2 3 4 5 6 7 8 9 10) asOrderedCollection.  
[ 1000000 timesRepeat: [  
  coll  
  select: [:each | each > 5]  
  thenCollect: [:i | i * i ] ]  
] timeToRun  
> "0:00:00:00.362"
```



bench

- Returns how many times the code can get executed in 5 seconds
- Answer a string with meaningful description

```
[ 1000 factorial ] bench  
> '610.234 per second'
```

The higher the better!

```
[ 1234 factorial ] benchFor: 2 seconds
```

Time Profiler

- TimeProfiler: a sampling-based code profiler
- At regular interval, take information from the execution stack

TimeProfiler

```
spyOn: [ 20 timesRepeat: [  
    Transcript show: 1000 factorial printString ] ]
```

Time Profiler

The screenshot shows the Time Profiler window with a call tree. The root node is `63.4% [53ms] LargePositiveInteger(Integer)>>printString`. It is expanded to show `63.4% [53ms] LargePositiveInteger>>printOn:base:`, which is further expanded to show several sub-nodes. The most significant sub-nodes are `28.0% [24ms] LargePositiveInteger>>printOn:base:` and `36.6% [31ms] SmallInteger(Integer)>>factorial`. The `SmallInteger(Integer)>>factorial` node is expanded to show its implementation details.

Method Call	Percentage	Duration
<code>LargePositiveInteger(Integer)>>printString</code>	63.4%	53ms
<code>LargePositiveInteger>>printOn:base:</code>	63.4%	53ms
<code>LargePositiveInteger>>printOn:base:</code>	28.0%	24ms
<code>LargePositiveInteger>>printOn:base:</code>	13.4%	11ms
<code>LargePositiveInteger>>highBit</code>	6.1%	5ms
<code>LargePositiveInteger(LargeInteger)>>printOn:base:nDigits:</code>	3.7%	3ms
<code>LargePositiveInteger(LargeInteger)>>*</code>	2.4%	2ms
<code>LargePositiveInteger(LargeInteger)>>-</code>	2.4%	2ms
<code>LargePositiveInteger(LargeInteger)>>printOn:base:nDigits:</code>	19.5%	16ms
<code>LargePositiveInteger(LargeInteger)>>*</code>	13.4%	11ms
<code>LargePositiveInteger(LargeInteger)>>-</code>	2.4%	2ms
<code>SmallInteger(Integer)>>factorial</code>	36.6%	31ms

Code | **Statistics**

You can edit some code in the top editor and accept to profile it.
The result consists in a tree of method calls shown in the middle panel.
Select a method call to see or change its implementation in the bottom editor.

Tallies building:
The tallies are built according to the time threshold value that you can change at the top right (Treshold input field).

Tree expanding:
A tree node is expanded only if its duration percentage is greater that the minimum value that you can change at the top left (Min duration percentage input field).

Summary

- [anExpression] timeToRun
- [anExpression] bench
- TimeProfiler spyOn: [anExpression]
- Check *Profiling Applications* Chapter in **Deep into Pharo**
(at <http://books.pharo.org>)



A course by



and



in collaboration with



Inria 2016

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France

<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>