Chapter 1

Some Good Coding Practices

In this chapter we present some simple and good practices that make your code often more efficient and avoid generating unnecessary garbage.

1.1 Avoid unnecessary string concatenations

In Smalltalk, the message , concatenates two strings. It is handy but this message is really costly since it copies the receiver. Therefore avoid it as much as possible and especially in loop since it multiplies the effect.

Prefer to use streamContents: , nextPut: and nextPutAll: since they avoid the duplication. The method streamContents: expects a block would argument is a stream on the receiver.

Let us have a look at the following code snippet.

```
String streamContents: [:s |
 #('Pharo''s' 'goal' 'is' 'to' 'deliver' 'a' 'clean,' 'innovative,' 'free' 'open−source' 'Smalltalk' '
       environment.' 'By' 'providing' 'a' 'stable' 'and' 'small' 'core' 'system,' 'excellent' 'dev' '
       tools,' 'and' 'maintained' 'releases,' 'Pharo' 'is' 'an' 'attractive' 'platform' 'to' 'build' 'and
       ' 'deploy' 'mission' 'critical' 'Smalltalk' 'applications.')
    do: [:each | s nextPutAll: each; nextPut: Character space].
    s contents]
```

Note that we took an example with many strings to stress the effect, but the bench results shows already a 4 factor.

```
[String streamContents: [:s |
 #('Pharo''s' 'goal' 'is' 'to' 'deliver' 'a' 'clean,' 'innovative,' 'free' 'open−source' 'Smalltalk' '
       environment.' 'By' 'providing' 'a' 'stable' 'and' 'small' 'core' 'system,' 'excellent' 'dev' '
       tools,' 'and' 'maintained' 'releases,' 'Pharo' 'is' 'an' 'attractive' 'platform' 'to' 'build' 'and
       ' 'deploy' 'mission' 'critical' 'Smalltalk' 'applications.')
```

```
    do: [:each | s nextPutAll: each; nextPut: Character space].
    s contents]] bench
        ⟶    '110,000 per second.'

[| s|
s := ''.
 #('Pharo''s' 'goal' 'is' 'to' 'deliver' 'a' 'clean,' 'innovative,' 'free' 'open−source' 'Smalltalk' '
       environment.' 'By' 'providing' 'a' 'stable' 'and' 'small' 'core' 'system,' 'excellent' 'dev' '
       tools,' 'and' 'maintained' 'releases,' 'Pharo' 'is' 'an' 'attractive' 'platform' 'to' 'build' 'and
       ' 'deploy' 'mission' 'critical' 'Smalltalk' 'applications.')
    do: [:each | s := s , each , Character space asString].
    s ] bench
        ⟶    '31,500 per second.'
```

Another example that may be a bit draft and fuzzy but gives an idea of the cost that you may gain.

```
[String streamContents: [:s |
    Smalltalk globals allClasses do: [:each |
                            s nextPutAll: each name].
    s contents]] bench
        ⟶    '246 per second.'

| s |
s := ''.
[Smalltalk globals allClasses
    do: [:each | s := s, each name]] bench
        ⟶    '2.01 per second.'
```

Preallocate when possible. Preallocate when possible the string used withnew:streamContents:. For example, the following method does not take advantage that the result is always a 8 characters string.

```
Time>>print24
    "Return as 8−digit string 'hh:mm:ss', with leading zeros if needed"
    ↑String streamContents:
        [ :aStream | self print24: true on: aStream ]
```

This version does the same but more efficiently, since the system does not have to reallocate the underlying buffer.

```
Time>>print24
    "Return as 8−digit string 'hh:mm:ss', with leading zeros if needed"
    ↑ String new: 8 streamContents: [ :aStream |
        self print24: true on: aStream ]
```

Use nextPut: when possible. When you can, use nextPut: use it instead of nextPutAll:. Indeed nextPutAll: requires that the argument is a container of elements. When you have already the element, no need to create an extra container.

For example better use the second form.

```
stream nextPutAll: '0'
```

```
stream nextPut: $0
```

1.2 Avoid creating temporary objects

The computation of minute in Time could be written as asDuration minutes . However, this solution creates a duration object that is only used to get the minutes. In addition to be slow such approach generates extra garbage which stresses the garbage collector.

```
Time>>minute
  ↑ self asDuration minutes

Time>>asDuration
  "Answer the duration since midnight"
  ↑ Duration seconds: seconds nanoSeconds: nanos
```

A much better solution is to use the encapsulation of the class Time and performs the computation locally as

```
Time>>minute
  "Answer a number that represents the number of complete minutes in the receiver,
  after the number of complete hours has been removed."
  ↑ (seconds rem: SecondsInHour) quo: SecondsInMinute
```

1.3 Avoid several iterations on the same collection

It may seem obvious but it is better to iterate once than two on the same collection, when we can do what we want in a single pass.

For example, the following code creates a first string then creates another one where the character $: is removed.

```
Time>>hhmm24
  "Return a string of the form 1123 (for 11:23 am), 2154 (for 9:54 pm), of exactly 4 digits
    "
  ↑(String streamContents:
```

```
    [ :aStream | self print24: true showSeconds: false on: aStream ])
        copyWithout: $:
```

Better implement it as

```
Time>>hhmm24
    "Return a string of the form 1123 (for 11:23 am), 2154 (for 9:54 pm), of exactly 4 digits
        "
    ↑ String new: 4 streamContents: [ :aStream |
        self hour printOn: aStream base: 10 length: 2 padded: true.
        self minute printOn: aStream base: 10 length: 2 padded: true ]
```