VisualWorks Optimisation: Tips & Techniques

Bernard Horan and Laura Hill
JP Morgan

Collections

When it is known that a collection is going to become quite large, create it using new:, supplying a guess at its final size. (In the case of a dictionary, multiply your guess by four.) The default new only allocates between 2 and 10 elements (depending on the class), which can cause the collection to waste a lot of time growing (copying) itself.

Streams and Strings

Use a stream protocol rather than the concatenation operator to build a large collection from multiple subcollections.

NOT:

```
| s |
s := String new.
$a asInteger to: $z asInteger do: [ :c |
s := s, (String with: c asCharacter)]
```

BUT:

```
| s |
s := (String new: 26) writeStream.
$a asInteger to: $z asInteger do: [ :c | s nextPut: c
asCharacter].
s contents
```

Create streams outside of loops and reset rather than creating them inside loops. If a stream–based message gets hits a lot, consider retaining the stream as an instance variable and resetting it rather than creating a new one each time. This also avoids the problem of deciding a good initial size for the stream contents since it will grow to the maximum required by history.

Avoid creating Large, Short–Lived Objects

There is always a tension when iterating over some aspect of a structure between building a collection of objects for the projection of that aspect, or building a special–purpose iterator. Example, in Behavior:

```
Object selectors do: [ :selector | ...]
Object selectorsDo: [ :selector | ...]
```

(The latter does not exist.)

The first of these constructs a Set which is discarded after the iteration is finished. This may be inefficient (especially as the selectors are already held as a separate Array). However, it is probably more flexible and reusable. Unfortunately there is no easy answer here – treat each case on its merits.

Similarly, use keysDo: instead of keys do: and associationsDo: instead of associations do:. The latter versions make a copy before doing the enumeration, whereas the former enumerate over the elements directly.

Avoid Recomputations – General

General principle: avoid repeatedly computing the same result, but keep the previously computed result. This is a space–time trade–off. This is especially easy in an object–oriented language, as you can often hide the cache inside an object or class, e.g., an instance variable, or a dictionary held by a class variable.

Often, use an instance variable which is either a useful recently–computed value, or nil. If the variable is nil, compute the value instead and retain in the variable. If the cached value becomes inappropriate, set the instance variable back to nil.

Examples:

SystemDictionary>>classNames

BorderedWrapper>>insetDisplayBox

CompositePart>>preferredBounds

Browser menu class variables.

Avoid Recomputations – Displays

If a view is composed of only a small number of different Images, generate them all, once, and retain them using an instance or class variable. Example: class variables in LabeledBooleanView.

Remember that the pixels in a Pixmap is stored externally to the object memory, whereas that in an Image is held by Smalltalk. This means displaying a Pixmap is likely to be much faster than an Image (especially if using X on a remote display).

Class CachedImage is provided to switch between the two on demand.

General

1) In nested conditionals, put the most likely case first.
2) Don't use isKindOf:. Besides being slow, it represents bad object–oriented style.
3) Use self class == aClass instead of self isMemberOf: aClass. Better still, rewrite your code so you don't need to know the class of an object.
4) Don't use respondsTo:. Besides being slow, it represents bad object–oriented style (indicates the sender is taking responsibility for something that should be handled by the receiver).
5) Unless you are concerned about numerical accuracy (e.g., in monetary calculations), convert Fractions and FixedPoints to Floats before performing mathematical operations.
6) Use the following special selectors, which are optimised by the compiler:

   to:do:, ifTrue:ifFalse:, whileTrue:, and:, or:

7) and: is more efficient than & because it does not evaluate the argument if the receiver is false. Similarly, or: is more efficient than | because it does not evaluate the argument if the receiver is true. Both and: and or: are inlined by the compiler, so that no objects are created to represent the literal block arguments. So, unless evaluating the argument has side effects (which is, perhaps, bad style), use and: and or: instead of & and |.

   NOT:

   ```
   self sensor blueButtonPressed not & self
   viewHasCursor
   ```

   BUT:

   ```
   self sensor blueButtonPressed not and: [self
   viewHasCursor]
   ```

8) If a method requires repeated use of Character cr or Character space (for example), use the variables defined in pool dictionary TextConstants or its IOConstants subset to avoid repeated message sends. To gain access, list the pool dictionary in the class description.
9) Send self changed: nil with: nil rather than the more general self changed, which simply builds the same message for you. Similarly, implement update:with:from: rather than update:.
10) If the same message is being sent repeatedly inside a loop to access a constant, assign it to a temporary variable outside the loop.

    NOT:

    ```
    quantities inject: 0
          into: [ :tot :qty | tot + qty * self getPrice]
    ```

    BUT:

    ```
    | price |
    price := self getPrice.
    quantities inject: 0
          into: [ :tot :qty | tot + qty * price]
    ```

11) Avoid creating Symbols. Avoid sending asSymbol.
12) Use Symbols as dictionary keys in preference to Strings.
13) Sending a message via perform: is up to three times as slow as a direct message send.

    NOT:

    ```
    self perform: #messageSend
    ```

    BUT:

    ```
    self messageSend
    ```

Specialised Objects

Use of specialised subclasses of collection classes can give dramatic performance improvements.

Example: use of RunArray when Array would be inefficient (Primarily saves memory, but may improve speed if memory is tight).

Example: specialised Dictionary subclasses, optimised for storage space, insertion/removal, or search time.

A useful optimisation is to specialise on the contents of a collection: (e.g., String is an optimised Array).

Encapsulate Complex Processes in Objects

This is partly a style issue, but can also impact on performance.

If you are implementing a complex algorithm, operating on many objects, with many intermediate states, you should consider encapsulating and controlling the algorithm within a single object.

This can save much parameter passing and accessing of shared variables; both can lead to uglier code.

Examples: Compiler, scanners of all sorts.

Use Object Identity

Testing for object identity is very fast: the current compiler inlines the test, and uses no messages at all (this also means redefining == is completely ineffective). Hence, use == (and ~~) rather than = (and ~=) where safe to do so.

Much more effective is the use of identity–based collections (IdentitySet and IdentityDictionary). When building new keyed collections, consider providing equality– and identity–based versions. Alternatively, use objects whose definition of equality is identity (e.g., Symbols).

Use == instead of = when comparing Symbols, Characters and SmallIntegers.

NOT:

```
x = 3 ifTrue: [...]
```

BUT:

```
x == 3 ifTrue: [...]
```

Also, when testing if a variable is nil, use == rather than isNil.

Blocks

A simple block that makes no references to private variables other than its own arguments or temporaries, is called a clean block. A simple block that makes no references to private variables other than its own arguments or temporaries, or self, instance variables, or arguments to any surrounding blocks or method is called a copying block.

Clean blocks are bound at compile time, and are the fastest kind. Copying blocks are slower, but still faster than the most general kind of simple block (known as dirty blocks) and continuation blocks. In general, move the declarations of temporaries to the innermost possible block.

The special selectors mentioned above are inlined if literal blocks are used, so no block objects are created, nor are messages sent to evaluate the blocks, hence for those messages there is no need to worry about the clean/copying/dirty distinction.

Clean:

```
[:i | i sendMessage]
```

Copying:

```
[:i | self sendMessage: i]
```

Dirty:

```
| temp |
[:i | temp + i sendMessage]
[ ... ^nil]
```

Exceptions and Contexts

The exception–handling mechanism is mostly implemented in Smalltalk itself, with a little primitive support. It works by using the thisContext pseudo–variable to access the current context, and thence to access the stack of Contexts (MethodContexts and BlockContexts) in the current process's stack.

Hence, whenever an exception is raised, the stack has to be converted into object form. This can take a considerable amount of time. Hence it is advisable to only use exceptions for genuinely exceptional cases. Also, avoid using thisContext in performance–critical code.

Contexts also have to be converted to object form whenever a process is suspended (either due to suspend or a semaphore wait). This puts a minimum overhead on process switching.

With thanks to Mario Wolczko and John Nolan