CINCOM® Technical Note

Implementation Limits

This document describes aspects of the VisualWorks inner workings that impose restrictions on certain implementations.

This version of the document has been updated to reflect changes in VisualWorks 7.4.

Size Limitations

The following table gives the size limitations for various aspects of the system. A limit of “None” implies that no hard limit exists, though available address space is an upper bound in every case.

Unit	Limit	Comment
Number of objects	None	Objects are limited by address space. (See below)
Object size	256 MB for byte objects 1 GB for pointer objects	32-bits: 256Mb for byte objects/2^{28} bytes for byte objects. 1 Gb for pointer objects/2^{28} slots for pointer objects. 64-bits: $7.2x10^{16}$ bytes for byte objects/2^{56} bytes for byte objects. $5.8x10^{17}$ for pointer objects/2^{56} slots for pointer objects.
Named instance variables	4096 per class	Includes inherited instance variables
Method variables	255	Includes arguments, named temporary variables, unnamed temporary variables (needed to implement to:do: loops, etc.). Also includes pushes and pops, so the effective limit may be a little less.
Block variables	255	Includes block arguments and temporaries; in some circumstances, it also includes arguments and temporaries from outer scopes to which the block refers. Also includes pushes and pops (see above).
Method literals	16,777,216	Includes ordinary literals (strings, numbers, etc.), message selectors (other than about 200 of the most common selectors), static variables (global, pool and class) that are referenced, and one for each block.
Block nesting	16,777,215 levels	The maximum level of block nesting, if the innermost block does a ^ return.
Method branches	67,108,863 bytes	The VM may impose a limit lower than this, but the byte codes do not.

Regarding the number of objects, in a 32-bit system the average object size is on the order of 64 bytes, so max objects limit is approximately 2^32 / 64 = 67108864 when 4 GB of memory is available. Some 32-bit OSs limit the total per-process address space to 3/4 or 1/2 of the full 32-bit space. In the 64-bit system average object size is on the order of 96 bytes, so max objects limit is approximately 2^32 / 96 = 44739244 when 4 GB of memory is available, or around 1.9x10^17 objects if a full 2^64 bytes of memory is available.

Changes to the bytecode set for 7.4 have forced changes to subclasses of InstructionStream, or to code that creates an InstructionStream on a method in order to use, e.g., scanFor:. If interested, browse references to InstructionStream in both 7.3 and 7.4 to see how the code has changed.

Open-coded Blocks

All control constructs in Smalltalk are defined as operations involving blocks, but, in the interest of performance, certain of these messages are optimized by the compiler by in-lining in the compiled method. Ordinarily, literal blocks create BlockClosure instances when compiled that reference separate methods, and create separate contexts when invoked. By contrast, in-line optimized or "open-coded" blocks have their code generated inside the defining method and do not have associated BlockClosures; their arguments and temporaries are merged into the enclosing method's context as "compiler-generated temporaries."

The following constructs are subject to block open-code optimization by the compiler:

```
BlockClosure>>whileTrue
BlockClosure>>whileTrue:
BlockClosure>>whileFalse
BlockClosure>>whileFalse:
BlockClosure>>repeat

Boolean>>and:
Boolean>>or:
Boolean>>ifTrue:
Boolean>>ifFalse:
Boolean>>ifTrue:ifFalse:
Boolean>>ifFalse:ifTrue:

Number>>to:do:
Number>>to:by:do:
Number>>timesRepeat:
```

For the most part, blocks that are open-coded by the compiler have the same semantics as ordinary blocks. But there are some distinctions that should be noted with respect to context materialization and visibility.

Restrictions on optimization are noted below, under "Special Treatment Only at Compile Time". In general, if the receiver and/or argument is expected to be a literal block but is not, or if the block takes an unexpected number of arguments, the message is not optimized.

Note that the Boolean and Number messages are optimized even when it's uncertain whether the receiver will be of the right type. So, if the receiver is a Foo, and Foo understands #to:by:do: to mean something else, the compiler will still optimize the message, and the code will break. To avoid this, do not redefine these messages in your classes.

Shared Context

True closed blocks are invoked in a separate Context. Therefore, the following code evaluates to false:

```
| outerContext answer |
outerContext := thisContext.
(1 to: 1) do: [:i | answer := thisContext == outerContext].
answer
```

On the other hand, the following evaluates to true:

```
| outerContext answer |
outerContext := thisContext.
1 to: 1 do: [:i | answer := thisContext == outerContext].
answer
```

Of course, one typically doesn't need to know the current context (and the compiler *could* refuse to open-code blocks that referred to thisContext directly). But code that inspects the context stack (such as the debugger, the profiler or the exception-handling facility) can see the difference. This invariably is not a problem in practice.

Browser Visibility

Because the compiler rewrites the methods for open-coded blocks, the methods that use these messages do not register as senders of the message. For example, the expression 1 to: 5 do: [:i |] is compiled as if it were:

```
| t1 |
t1 := 1.
[t1 <= 5] whileTrue: [ti := t1 + 1]
```

Hence, it isn't thought to be a sender of to:do: at all, (or even whileTrue: because that is also open-coded), but it *is* considered to be a sender of +. All this means in practice is that you can't use the browser to find the senders of the 12 messages listed in "Open-coded Blocks."

Block Optimization

There are three common patterns of usage for blocks in Smalltalk programs:

- Arguments to ifTrue:, ifFalse:, or:, etc. When these blocks are in-lined by the Compiler's open-coded block optimization facility, they are not blocks as far as the execution machinery is concerned, and are not further discussed here.
- Exception blocks (for example, passed as an argument to ifAbsent:). These blocks are created fairly frequently and almost never executed.
- Iteration blocks (for example, passed as an argument to do:). These blocks are executed many times.

Because close-coded blocks are so common, both block creation and block invocation must be supported efficiently.

The [ ] language construct creates a BlockClosure object (except for open-coded blocks as mentioned above), which contains only the starting point and a reference to the "home" (the enclosing method). Invoking a block (sending a value message to a BlockClosure) creates a BlockContext, which is almost exactly the same as a MethodContext; it has local arguments and temporaries of its own, and a receiver which is the BlockClosure. To reference variables in enclosing scopes (outer blocks, the enclosing method, the method's receiver, or the receiver's instance variables), the enclosing method or block must arrange to share this data with the block by storing the value in an Array that is used both by the block and by the enclosing code to access the variable.

Because allocating these extra arrays, and using the extra indirection to access a variable, is somewhat more expensive than accessing the variable directly, we adopt a slightly more sophisticated implementation based on experience from the lexically scoped Lisp world. If the code in a block refers to variables from outer scopes, but it can be determined at compile time that the variables can't change their values after the block has been created, then the values can be copied into the BlockClosure without creating a shared Array to wrap them.

In order to support explicit returns within blocks, such a block needs an explicit pointer to its enclosing context. Blocks that include an explicit return (and therefore have a pointer to their outer context) are called “full blocks,” and those that do not are called “clean blocks.” Full blocks that refer to outer scope variables are called “full copying blocks,” while clean blocks that refer to outer scope variables are called simply “copying blocks.” If there is only a single copied value, it is stored in the closure directly; otherwise an Array is allocated to hold the copied values. The compiler generates different code for creating the block at runtime, depending on whether it is full or not, and whether it uses outer scope variables. Clean BlockClosures (assuming they do not use outer scope variables) can and are created at compile time rather than runtime.

It is a requirement of the current generation of Smalltalk that activations be accessible as first class objects. However, for efficiency, Contexts are represented as frames on a stack, and rarely instantiated as objects unless some code needs to access them as objects. Frequently, even when they are stored as obects, the object can be nothing more than a proxy for the stack frame, using specialized primitives to emulate the behavior of a more conventional object.

Apart from this optimization, Smalltalk supports the semantics of blocks in a straightforward way. It has a BlockClosure class with three instance variables:

copiedValues	nil for full or clean blocks, a single value or an Array for copying blocks
method	a CompiledBlock
outerContext	a MethodContext or BlockContext for dirty blocks, nil for clean blocks

Note that creating a full block involves not only allocating a BlockClosure object, but also materializing the outer activation as a Context object. This introduces an additional delayed cost, in that this Context object must remain synchronized with the stack frame until the frame is destroyed. However, this cost is substantially less than in previous versions of VisualWorks.

The Debugger

Some block contexts are more informative than others, and so the Debugger can provide more information in some context than in others.

For example, assume there’s a method SequenceableCollection>>foo. Array is a descendant of SequenceableCollection, so you can send Arrays the message foo. If you sent an Array the message foo, and an error occurs inside a block that is enclosed by the foo method, what the debugger shows depends on the block type.

If the block is clean or copying, you would see:

```
optimized [] in SequenceableCollection>>foo
```

This shows which method has been invoked, but not that the receiver is an Array.

If the block is full or full copying, you would see:

```
[] in Array(SequenceableCollection)>>foo
```

The debugger can give the additional information that the receiver (self) is an Array.

If the receiver’s class is the same as the class in which the method is defined, the class name is not duplicated inside the parentheses. The block would look the same in the debugger as a clean block, except that the word “optimized” does not appear.

Performance

Try to make *clean* blocks, especially if you care about performance. Blocks with ^ returns are full blocks. Blocks that reference outer-scope variables (even self or one of its instance variables) will be at least copying blocks.

Some examples:

```
| t |
[:x | t := x frobnicate] value: 1.
t := t * 2.
^t
```

The reference to t inside the block makes it a copying block, which incurs overhead. Even worse in this example, the change in t's value after the block is created causes t to be represented as an Array that is shared between the method and block, incuring even more overhead. Instead, you might write the following, which leaves the block clean:

```
| t |
t := [:x | x frobnicate] value: 1.
t := t * 2.
^t
```

Non-overridable Methods

Some methods are treated specially by the execution machinery and cannot be overridden. These optimizations are done for performance reasons. For example, the + method in SmallInteger is hard-wired because changing its definition would very probably break the system, for obvious reasons.

In releases before VisualWorks 3.0, the == method was always hard-wired. In releases 3.0 and beyond, while it is still hard-wired by default, the primitive

```
Behavior>>querySetNonArithmeticSpeicalSelectorInlining:
```

can be used to control the in-lining of == and ~~. Some transparent persistence schemes may profit from not in-lining == or ~~, and may use this primitive to do so. (See these entries in the list under "Special Treatment at Compile Time and Translation Time" below.)

Special Treatment Only at Compile Time

The following messages are treated specially by the compiler only at compile time:

anObject **and:** aBlock0
anObject **or:** aBlock0
anObject **ifTrue:** aBlock0
anObject **ifFalse:** aBlock0
anObject **ifTrue:** aBlock0 **ifFalse:** anotherBlock0
anObject **ifFalse:** aBlock0 **ifTrue:** anotherBlock0
anObject **timesRepeat:** aBlock0
aBlock0 **whileTrue:** anotherBlock0
aBlock0 **whileTrue**
aBlock0 **whileFalse:** anotherBlock0
aBlock0 **whileFalse**
aBlock0 **repeat**
anObject **to:** anotherObject **do:** aBlock1
anObject **to:** anotherObject **by:** aNumber **do:** aBlock1

If the receiver and/or argument(s) meet certain syntactic requirements, these messages are compiled open; otherwise, they are compiled as ordinary message sends. The following example is compiled as an ordinary message (and will cause a messageNotUnderstood error when executed).

```
1 and: 2
```

The requirements for open compilation are indicated in the message descriptions above. *aBlock0* or *anotherBlock0* means the receiver or argument must be a literal 0-argument block. *aBlock1* means the argument must be a literal 1-argument block. *aNumber* means the argument must be a literal number.

The effect of open compilation is that adding, removing or changing definitions of these messages in certain classes will have no effect on the execution of expressions that meet the open compilation requirements, specifically:

Any class
and:
or:
ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:

BlockClosure
whileTrue:
whileTrue
whileFalse:
whileFalse
repeat

Any class
to:do:
to:by:do:

Note that if the receiver and arguments do *not* meet the open compilation requirements, the expression is compiled as an ordinary message send. For example, the user can define a method in Integer for and:, and the method will be invoked as expected for:

```
1 and: 2.
```

Note that the requirements are syntactic, not semantic. For example,

```
something and: [some other thing]
```

compiles open, but

```
something and: someOtherThing
```

compiles as an ordinary message send, which is executed even if the value of someOtherThing turns out to be a BlockClosure at execution time. This is why correct definitions for the above-mentioned selectors must exist in classes True and False (for the first list above), BlockClosure (for the second list), and Number (for the third).

The user can change this state of affairs in a fairly straightforward way by modifying the compiler. MessageNode class>>initialize constructs the dictionary that determines what messages should be compiled open; each message has a corresponding transformation method defined in class MessageNode. Note that one must recompile the entire system for the changes to take full effect. Removing the conditionals (ifTrue/False or whileTrue/False:) from the list is likely to produce unacceptable system performance.

Note: Senders of these messages cannot be found using the **Browse → References To…** command.

Special Treatment at Compile Time and Translation Time

Many messages are compiled using “special selector” opcodes. These opcodes have two different functions: they eliminate the need to store copies of the selector in the sender’s literals, and the translator knows how to compile some of them specially.

The translator treats the following selectors specially by generating machine code that performs certain explicit class checks before (or instead of) sending a message. As long as these selectors are compiled as “special selectors,” their definitions for the given classes are fixed and cannot be modified by the user.

```
(SmallInteger) + (SmallInteger)
(SmallInteger) - (SmallInteger)
(SmallInteger) < (SmallInteger)
(SmallInteger) > (SmallInteger)
(SmallInteger) <= (SmallInteger)
(SmallInteger) >= (SmallInteger)
(SmallInteger) = (SmallInteger) literal
(SmallInteger) ~= (SmallInteger) literal
(Object) == (Object)
(Object) ~~ (Object)
```

Note that = and ~= are only translated specially if the argument is a literal number. Note also that if an addition or subtraction overflows, the expression is handled as a normal message send.

If the receiver or argument doesn't meet the listed criterion, the expression is executed as a normal message send. Note that this is a *semantic* check carried out at runtime, not a syntactic one.

The special selectors are defined in DefineOpcodePool class>>initializeSpecialSelectors and extendedSpecialSelectors. While it is permissible to modify the set of special selectors, it is never (and never has been) a good idea to do so. Changing the set of special selectors will invalidate BOSS and parcel files, including VisualWorks system parcels. BOSS and parcel files do not contain any record of the set of special selectors in effect when the files were created, even though the bytecodes in methods in those files implicitly refer to the special selectors.