Blocks – Advanced Use

Blocks are a powerful tool in Smalltalk. Their effective use can improve the readability, reusability and efficiency of code. However, because most programmers are weaned on languages with no equivalent, many programmers do not make good use of blocks.

This module includes hints on how to make good use of blocks, and describes some advanced features.

1. Review of Mechanism

Blocks are instances of class BlockClosure. A block may be evaluated by sending it one of the messages in Table 1, depending on the number of arguments the block expects. The corresponding methods are all primitives — hence, sending the wrong message will lead to an error.

message	# arguments
value	0
value:	1
value: value:	2
value: value: value:	3
valueWithArguments:	any number (in an Array)

Table 1: The value messages

Alternatively, blocks with zero arguments may be sent one of the messages repeat, whileFalse:, whileFalse, whileTrue: or whileTrue — they in turn rely on the use of the value message.

2. Variable Scope and Lifetime

In addition to arguments, a block may also have its own temporary variables. It can also refer to self, and any instance variables of self, and any shared (class, pool or global) variables accessible in the method in which the block is defined.

The Virtual Machine represents the state of execution as Context objects, one per method or block in progress. Method activations are represented by MethodContexts, block activations by BlockContexts. Each context contains a reference to the context from which it is invoked, the receiver, arguments and temporaries in that context, and the method or block being executed.

Private variables (such as self, instance variables, temporaries and arguments) are *lexically* scoped in Smalltalk[1]. Therefore, these variables are *bound* in the context in which the block is defined (its *home* context), rather than the context in which the block is evaluated.

As an example, consider the following methods (assume that they are implemented in some arbitrary class):

```
testScope
    | t |
    t := 42.
    self testBlock: [Transcript show: t printString]

testBlock: aBlock
    | t |
    t := nil.
    aBlock value
```

1. Also known as *static binding of variables* — not to be confused with static/dynamic binding of procedures/methods.

Evaluating the testScope message will cause 42 to be displayed in the Transcript. However, if Smalltalk was not a dynamically–scoped language (i.e., *statically*–scoped), the Transcript would display nil. See Fig.1.

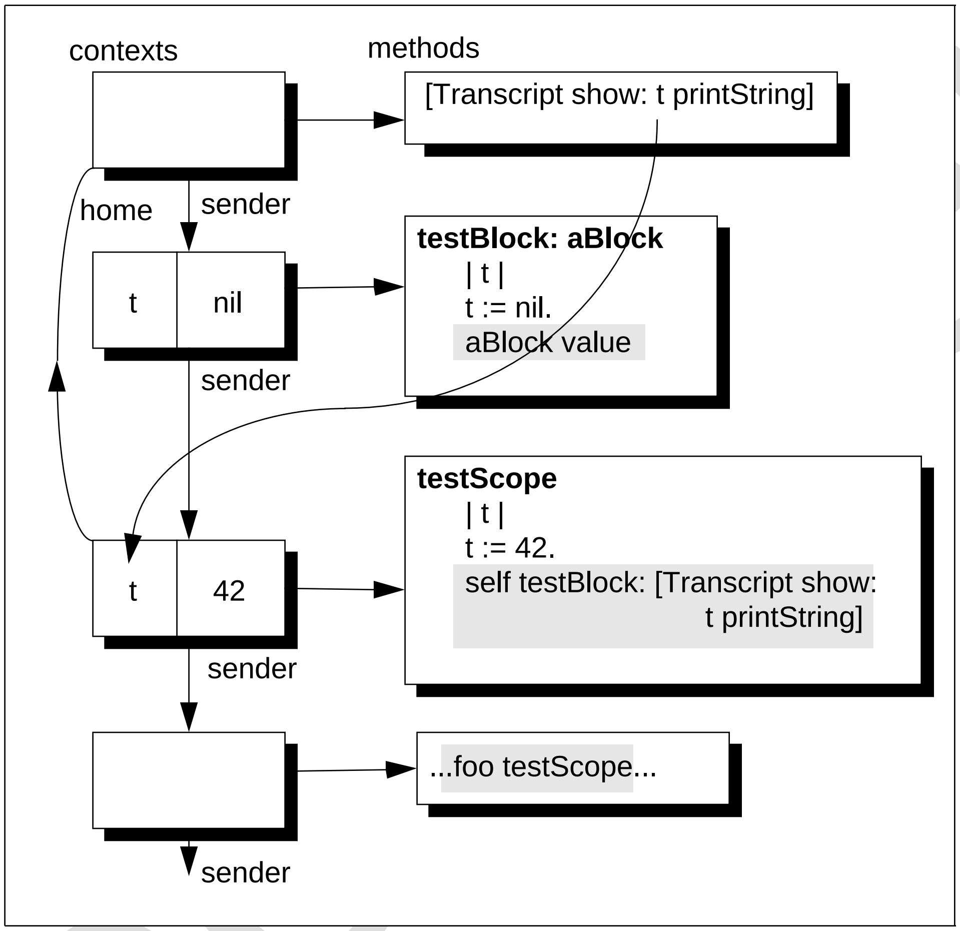


Figure 1: Lexical scope of variables

3. Returning from a Block

It is useful to distinguish between two kinds of blocks:

1. Those that end with a return expression. We will call these *continuation* blocks (there is no standard, concise term). For example:

   ```
   [:x :y | ^x + y]
   ```

2. Those not ending in a return expression, which we will call *simple* blocks.

When a *simple* block completes evaluation, it returns its *value* (the result of evaluating the last message expression in the block) to the method that sent it the value message (the *sender*).

When a *continuation* block completes evaluation it returns its result to the method that activated its *home context*.See Fig.2.

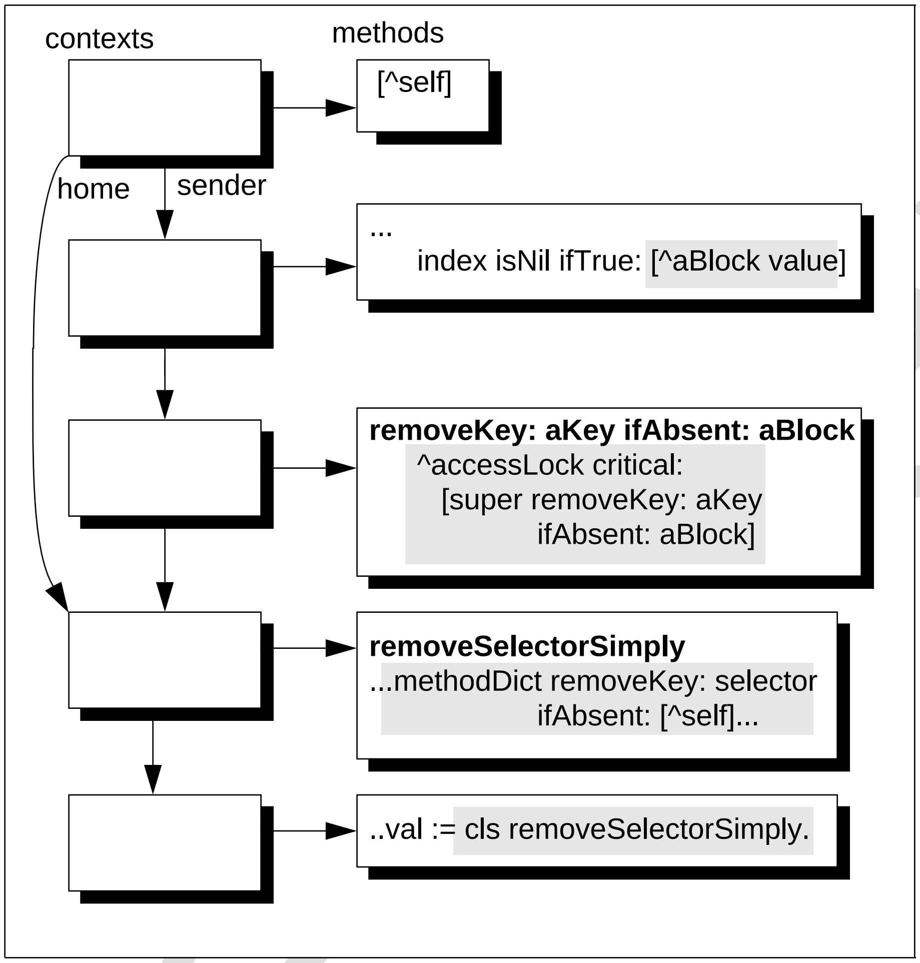


Figure 2: Continuation blocks

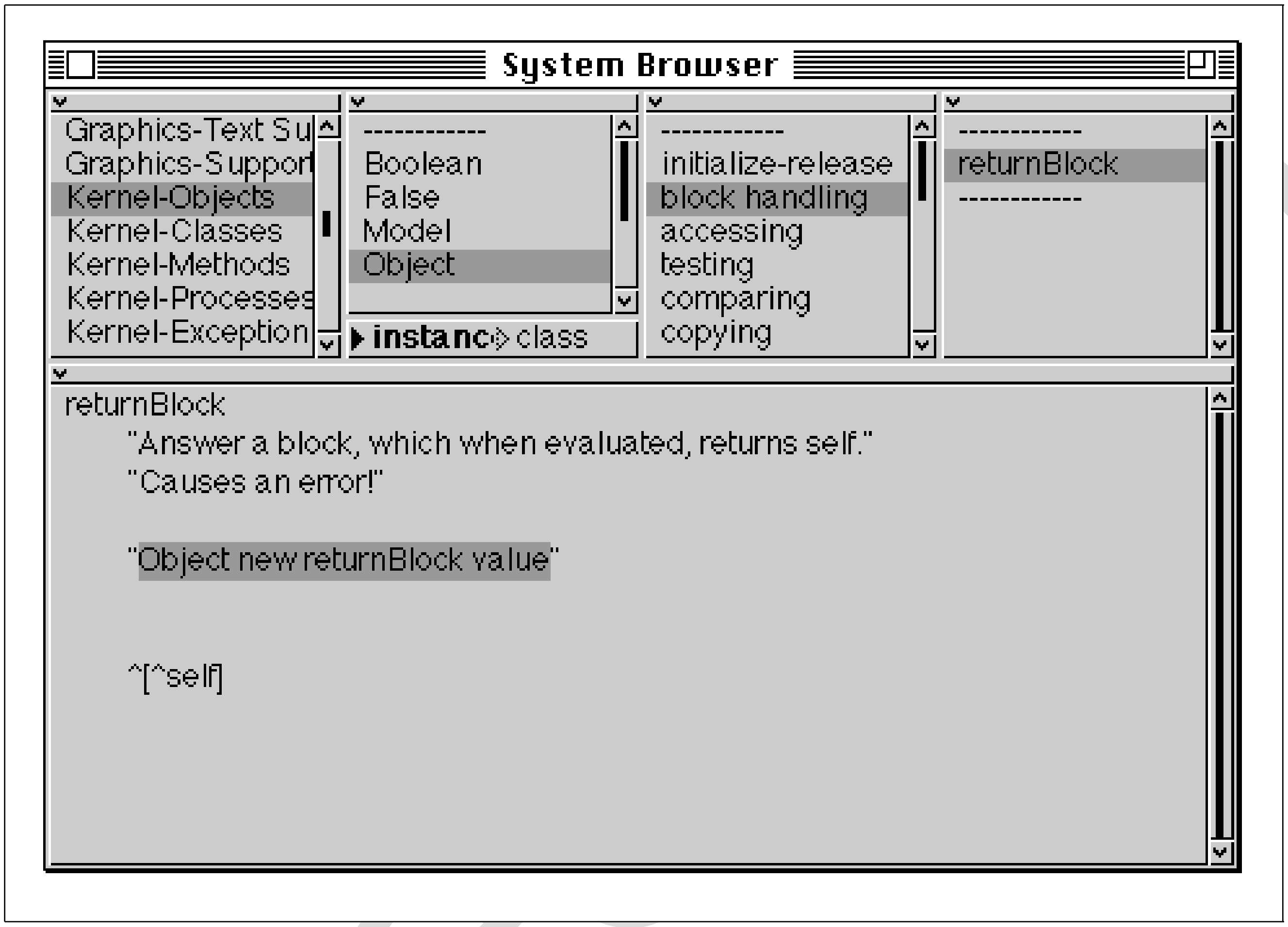
Thus, a block is always evaluated in the context in which it was defined. This means that it is possible to attempt to return (using a return expression inside a block, i.e. '^') from a method which has already returned using some other return expression. This run–time error condition is trapped by the virtual machine, and an error Notifier displayed. For example, Fig.3 shows the method returnBlock which we have added (for sake of brevity) to class Object. This method simply returns a block containing an expression to return self. Evaluating the expression

```
Object new returnBlock value
```

causes an error Notifier to appear (Fig.4).

Ex 1. A simple block can be evaluated many times; a continuation block can be evaluated at most once. Why? Examples:

```
| b |
b := [ :x | Transcript show: x. x ].
b value: 'a'; value: 'b'.
b := [ :x | Transcript show: x. ^x ].
b value: 'a'; value: 'b'
```

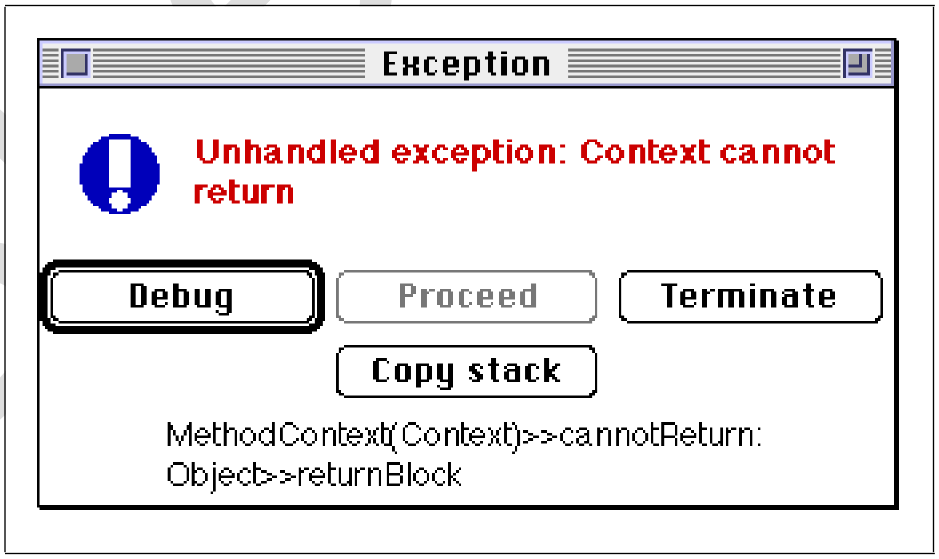


```
returnBlock
	"Answer a block, which when evaluated, returns self."
	"Causes an error!"

	"Object new returnBlock value"


	^[^self]
```

Figure 3: An example of a block expression that causes an error



Unhandled exception: Context cannot return

MethodContext(Context)>>cannotReturn:
Object>>returnBlock

Figure 4: A block cannot be evaluated once its has returned

Ex 2. In Smalltalk you can immediately return from a method using a continuation block:

```
exampleAbort
  self someTest ifTrue: [^nil].
  "rest of method"
```

However, there is no equivalent mechanism for exiting from a block without evaluating its last statement. Can you think how to build one?

Here is an example use which may help you to discover a solution:

```
| val |
val := [ :exit || goSoon |
    goSoon := Dialog confirm: 'Exit now?'.
    goSoon ifTrue: [exit value: 'Bye!'].
    Transcript show: 'Thank you for not exiting!'.
    'and have a nice day!!!!'] valueWithExit.
Transcript show: val
```

The idea is that the block is passed an *exit object* which, when sent the message value:, causes the block to immediately return, with the value passed to the exit object. (We will return to this code in the optimization module.)

4. Control Structures

Blocks are already used to implement the basic control structures in Smalltalk such as ifTrue:ifFalse:, whileTrue:, etc., and can be used to create new control structures. However, this is one place where it is *not* a good idea to experiment too wildly:

- The basic control structures are treated specially by the current implementation of VisualWorks (see Appendix D of the VisualWorks User's Guide). New looping control structures will not be so treated, and may be incredibly inefficient (unless implemented using the existing structures). This may be ameliorated should better Smalltalk compilers become available, but this seems unlikely in the near future.
- Thankfully, it is better to build control structures that are appropriate to the nature of the objects being manipulated, and these can usually be constructed using the basic, efficient mechanisms.

4.1. Example: A case Statement

It is relatively straightforward to create a new control structure similar to the *case* statement common in other programming languages. However, it is important to note that case statements should only be used to switch between different *values*, not *classes*. To create a simple case statement, use a Dictionary to represent the mapping between a value and the block to be evaluated for that value.

For example, the code below demonstrates how to associate a cursor key (represented by a Symbol) with a block of code.

```
| dictionary sensor |
dictionary := Dictionary new.
dictionary at: #Down put: [Transcript cr; show: 'Down!'].
dictionary at: #Up put: [Transcript cr; show: 'Up!'].
dictionary at: #Left put: [Transcript cr; show: 'Left!'].
dictionary at: #Right put: [Transcript cr; show: 'Right!'].
sensor := ScheduledControllers activeController sensor.
[sensor redButtonPressed]
        whileFalse:[sensor keyboardPressed
                              ifTrue:[| char block |
                              char := sensor keyboardEvent keyValue.
                              block := dictionary at: char ifAbsent: [].
                              block notNil ifTrue:[block value]]]
```

4.2. Iterators

Perhaps the best, and most powerful use of blocks is in building iterators. Iterators can provide control structures that are tailored to an object, easy to understand and use, and that do not violate that object's encapsulation.

Examples: do:, collect:, reject:, select:, inject:into:, in class Collection; findFirst:, findLast:, reverseDo: in class SequenceableCollection.

Ex 3. Browse the enumerating methods in Collection and SequenceableCollection.

Ex 4. Write methods allSatisfy: and anySatisfy: in Collection that take a block and return true (resp. false) if all (resp. if any) elements of the collection, when passed to the block, evaluate to true. For example:

```
#(1 2 3 4 5) anySatisfy: [:n | n even]
```

should return true, while

```
#(1 2 3 4 5) allSatisfy: [:n | n even]
```

should return false. Ensure that your method is efficient, and returns immediately in this case:

```
(1 to: 100000000) allSatisfy: [:n | n even]
```

When using iterators, take care not to modify the collection being iterated over; few (none?) of the system–provided collections can cope with this, and will fail in unexpected ways.

Something *not* to do:

```
aSet do: [:elem |
    …aSet remove: elem…]
```

Proof: Evaluate the following expressions a few times (with different classes instead of Array, such as Set or Dictionary):

```
| set |
set := IdentitySet new.
set addAll: Array allInstances.
set do: [:elem | set remove: elem].
set isEmpty
```

Ex 5. Can you think of a simple way to modify a collection class so that misuse of iterators (i.e., when the underlying collection is modified during an iteration) is automatically detected?

5. Unwind Blocks

There are occasions when it is important to guarantee that a sequence of message expressions will be evaluated in a method, to undo the consequences of earlier message expressions in the same method. For example, it is important to close a file if an error occurs whilst writing to it. Example:

```
| f |
f := 'foo' asFilename writeStream.
self writeMyContentsOn: f.
f close
```

If the stack is cut back or the process terminated, the close message will not be sent, possibly losing data.

VisualWorks provides a mechanism for handling this:

```
| f |
f := 'foo' asFilename writeStream.
[self writeMyContentsOn: f]
        valueNowOrOnUnwindDo: [f close]
```

The receiver of valueNowOrOnUnwindDo: is a zero–argument block, which is executed immediately, as if it were sent value. The argument to valueNowOrOnUnwindDo: will be evaluated after the receiver, even if the stack is cut back or *unwound* through this method context (see Fig.5).

It is a good idea to provide an evaluation mechanism that does this automatically, for example, try

```
Cursor wait showWhile: [(Delay forSeconds: 5) wait. ^nil]
```

Then browse Cursor>>showWhile:.

Ex 6. Add a class method

```
time: aBlock value: valueBlock
```

to Time that measure the time taken to evaluate aBlock (using the code from Time millisecondsToRun:), passing the *time* as an argument to valueBlock.

For example:

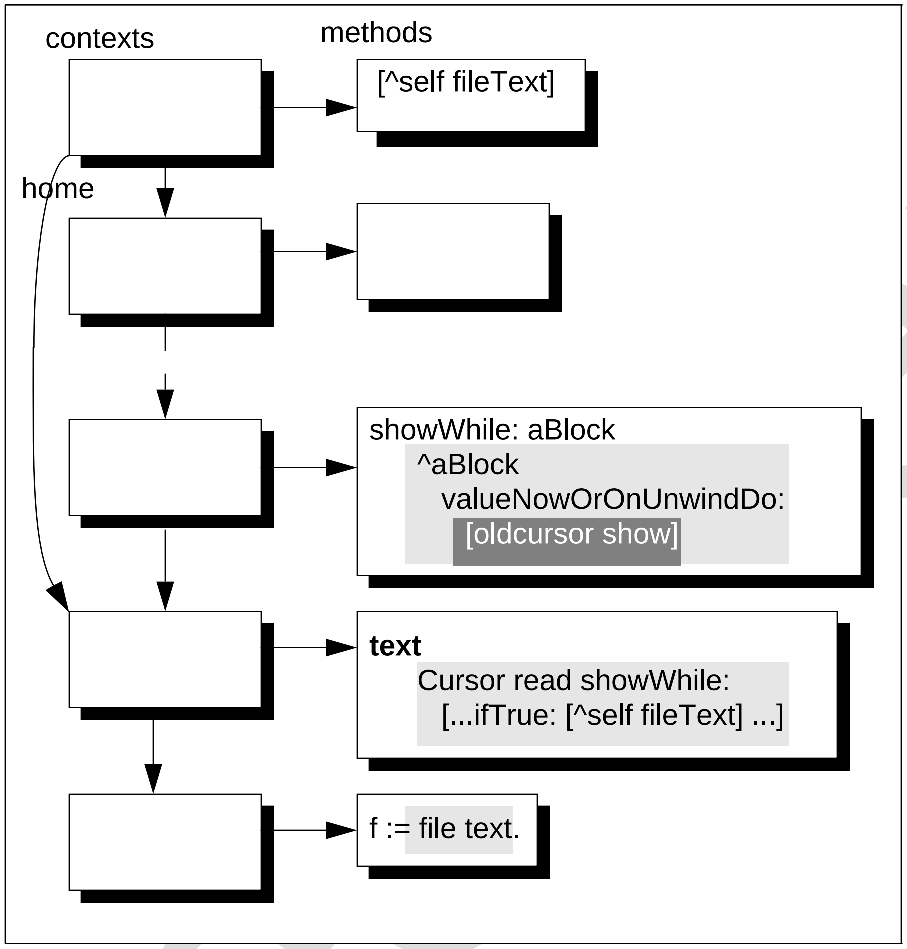


Figure 5: Unwinding

```
Time
  time: [1000 factorial]
  value: [ :t | Transcript show: t printString; cr]
```

should return 100! and print in the Transcript the time it takes to calculate the factorial. Modify your method to use an unwind block to ensure that the value: block is executed even if the time: block does a non–local return. For example:

```
Time
  time: [^1000 factorial]
  value: [ :t | Transcript show: t printString; cr]
```

It is also sometimes necessary to ensure that a a sequence of message expressions is evaluated *only if* an error occurs. In this case, there is a similar message valueOnUnwindDo: which will only evaluated the argument block if the stack is cut back because of an error, or if the process is terminated.

The use of these two messages highlights the problems described earlier regarding the inclusion of return expressions in blocks. For example, when evaluated, the message expression

```
^['receiver block'] valueNowOrOnUnwindDo:['unwind block']
```

returns 'receiver block', as expected. However, the inclusion of a return expression inside the unwind block will cause an error. Thus, the following expression fails:[1]

```
^['receiver block'] valueNowOrOnUnwindDo:[^'unwind block']
```

1. Note that although it is possible to include a return expression in the receiver block, it is considered bad style.