Miscellaneous Tricks

In this module we explain several tricks that we have found of use to the advanced VisualWorks developer. These include: multiple dispatching, the perform: message, the become: message, the doesNotUnderstand: message, and using nil as a superclass to build an *Encapsulator*. The module contains a description of each trick, combined with examples and exercises.

Double Dispatching

Occasionally, behavior is required which is dependent on the classes of more than one object, e.g. a + b, or messages to display items on a GraphicsContext.

```
2 + 4.0

2 + 4

4.0 + 2

4.0 + (2/3)

aGraphicsContext display: aLine at: 100@100.
aGraphicsContext display: aText at: 10@10
```

In such cases, *double dispatching* can be used to choose the right method.

1. General Approach

In addition to the general message foo:, a special case version is made for each possible kind of argument, fooClass1:, fooClass2: etc.

Then, in each of those classes, foo: invokes the message naming that class, reversing the receiver and argument. Example (from class Integer):

```
+ aNumber
    ^aNumber sumFromInteger: self
```

This technique is used extensively in the arithmetic methods in the ArithmeticValue hierarchy, and in the display:at: method in class GraphicsContext. However, it does give rise to several problems:

- For *n* classes, n^2 classes are required. For triple dispatching (dispatching on two arguments), n^3!
- The method selectors appear clumsy for non–commutative operators and triple dispatching (example from class Integer):

```
- aNumber
    ^aNumber differenceFromInteger: self
```

- The flow of evaluation is difficult to follow, as methods are scattered across several classes.

The perform: primitive

The perform: primitive(s) can be used to dynamically determine the selector used in a message.

message	# message arguments
perform:	0
perform:with:	1
perform:with:with:	2
perform:with:with:with:	3
perform:withArguments:	any number (in Array)

Table 1: The perform: messages to send messages indirectly

Example using perform	Equivalent message
x perform: #foo	x foo
x perform: #foo: with: y	x foo: y
x perform: #foo:bar:baz: withArguments: (Array with: y with: z with: w)	x foo: y bar: z baz: w

Table 2: Some examples of perform:

It is used in “pluggable” classes which can be customized to different objects.

On the whole, perform: can be replaced with blocks. For example, assume that a class contains a method doUsing:, as follows:

```
doUsing: aSymbol
    ^bar perform: aSymbol
```

Then if the message doUsing: #doThis is sent to an instance of that class, it is more or less equivalent to an alternative implementation using a block argument:

```
doUsing: aBlock
    ^aBlock value: bar
```

using the message expression doUsing: [:x | x bar]

When choosing between the two approaches, consider the following points:

- There is little to choose between them in terms of performance.

- Use of either makes code harder to understand (e.g., tracing through Browsers is trickier, and requires understanding of the data flow), hence use with discretion.
- Blocks can be more flexible, using local context if necessary.
- Blocks are better when the code to be executed is not obviously associated with one object (use of local context).
- A block is usually created during an initialization phase, and hence cannot be modified directly in the Debugger when it is being evaluated.

The only *essential* use of perform: is in a doesNotUnderstand: method (if deciding to proceed). See Fig.1.

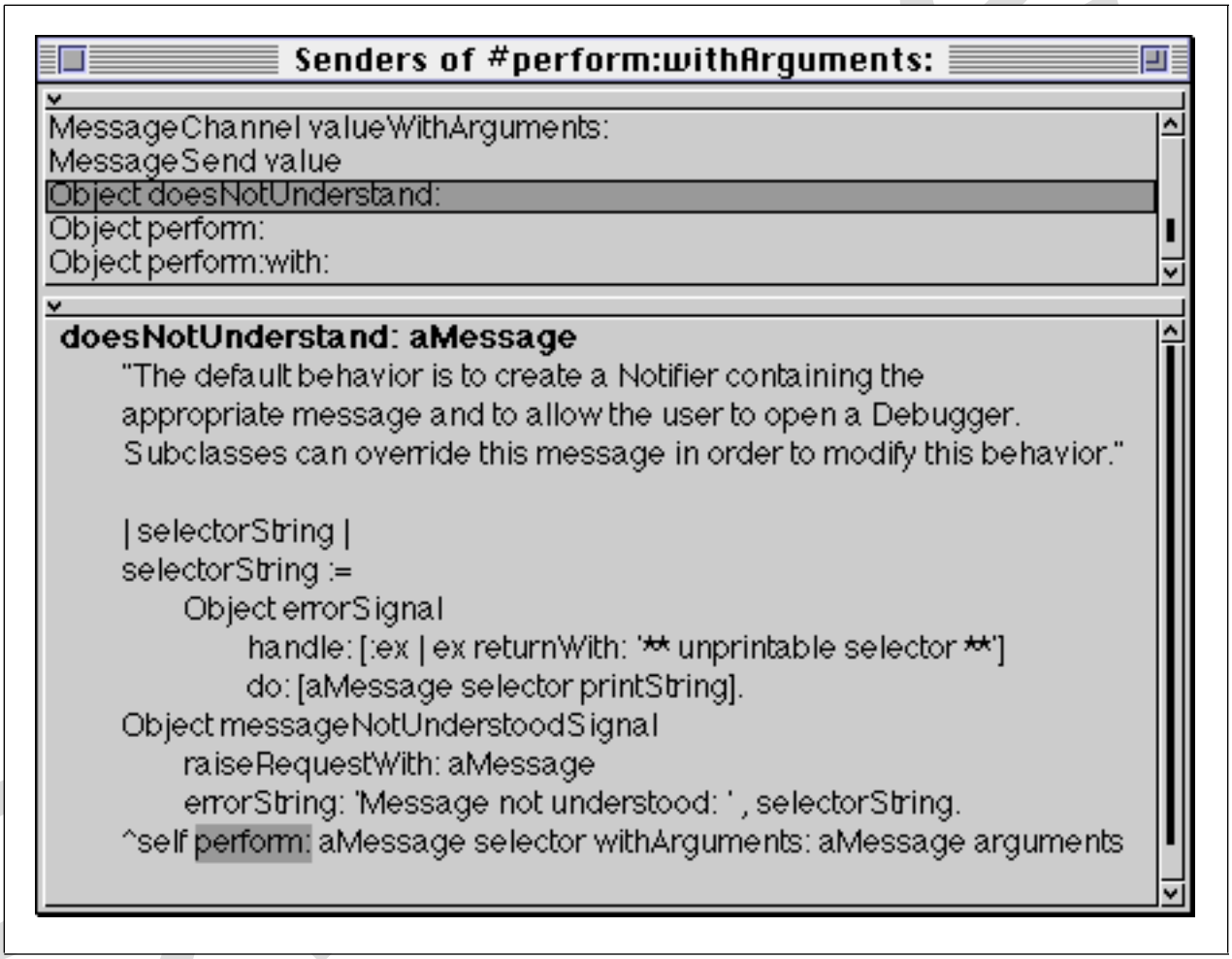

Figure 1: Using perform:withArguments: in doesNotUnderstand:

Ex 1. What is the effect of this code (save your image before trying it!)? Can you explain what happens?

```
| a |
a := Array with: #perform:withArguments: with: nil.
a at: 2 put: a.
Smalltalk perform: a first withArguments: a
```

(Due to Eliot Miranda)

Ex 2. Add a value: method to class Symbol which uses the receiver as the argument to a perform: message sent to the method argument. Test the code using the following expression:

```
#size value: Smalltalk
```

Similarly, add a value:value: method, which takes its second argument as the second argument to perform:with: message. Test the code using the following expression:

```
#at: value: Smalltalk value: #Processor
```

The become: primitive

The become: primitive exchanges the identities of the receiver and argument. For example, after

```
| a b c |
a := 3@4.
b := Rectangle fromUser.
c := a.
a become: b.
```

a and c refer to a Rectangle, b refers to a point.

```
Transcript show: a printString ; cr.
Transcript show: c printString.
b x: 3.
```

See Fig.2

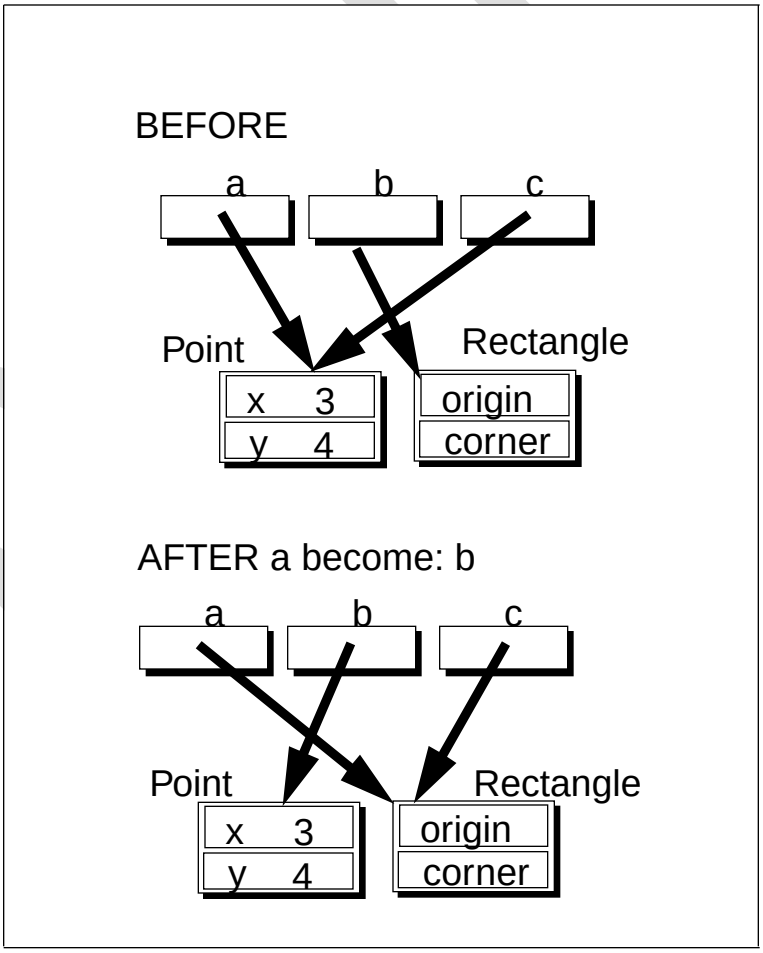

Figure 2: The effects of using become:

1. Existing uses of become:

1. To grow collections (in method changeCapacityTo:). Remember to redefine copyEmpty: when subclassing collections!

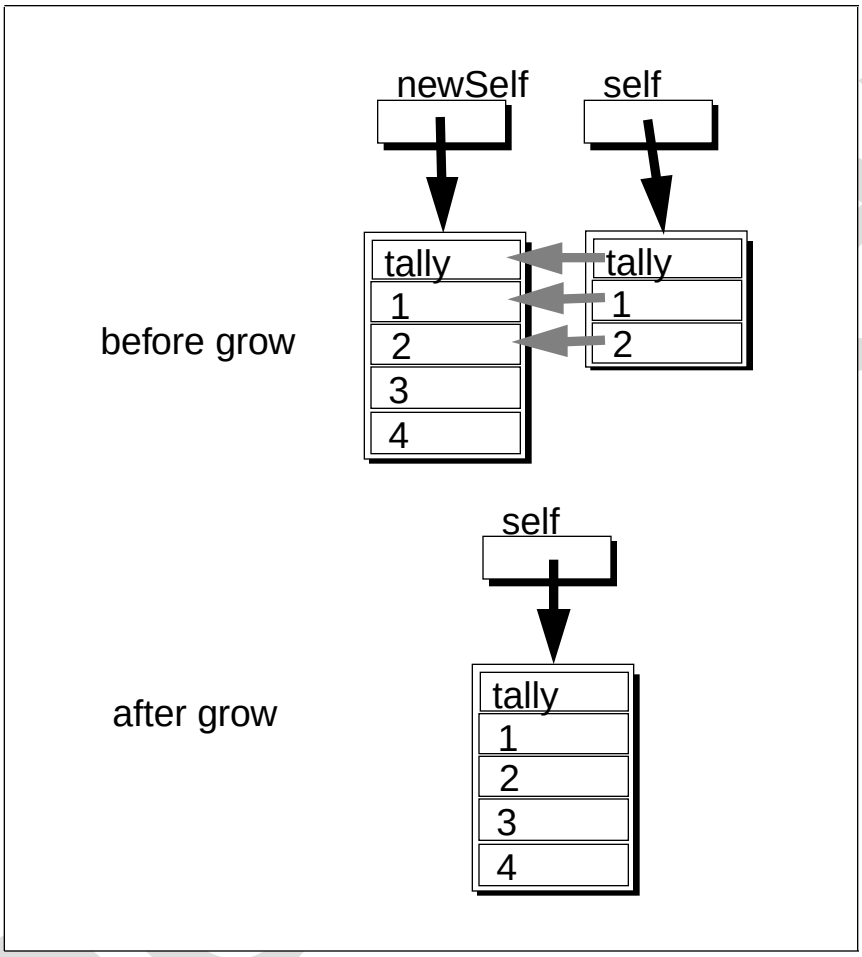

Figure 3: Growing an object

2. Mutating objects after class changes (e.g., when an instance variable has been removed).

3. Updating old instances recovered using BOSS.

Note that become: cannot be used for immediate objects (SmallIntegers and Characters in VisualWorks). VisualWorks also provides a primitive to change the class of an object: changeClassToThatOf: (in Object; see comment in method for restrictions).

2. Good reasons not to use become:

1. Dangerous and confusing. Hard to reason about. *Don't* use become: on system objects or pseudo–variables.

2. Not in the spirit of object–oriented programming, because the argument is not sent a message.

Always ensure that any use of become: with self is always the last expression in a method (to avoid potentially dangerous accesses to instance variables).

Ex 3. (For C/C++ programmers) Add methods inc and dec (to mimic ++ and --) to Number, to allow you to increment and decrement numbers. What are the restrictions? Why is this not a good idea?

Encapsulators

Name (which is awful!) was introduced in: *Encapsulators: A New Software Paradigm in Smalltalk-80*, by Geoffrey A. Pascoe, OOPSLA'86, ACM, pp.341–6. The basic idea of an encapsulator (aka a proxy, or surrogate) is that it sits in front of another object, passing messages through, but also performing some other function.

An example application might be an encapsulator to log messages to the Transcript.

1. Implementation in VisualWorks

An encapsulator understands no messages except:

- doesNotUnderstand:, which is used to capture every message, interrogate it somehow, and then forward it the encapsulated object (using perform:), and
- those messages used for its own operation (which should be unique to that class).

This is usually achieved by setting the encapsulator class's superclass to nil (like Object) — crude, but it works! When attempting this, you will get a notifier warning you that a class is being defined with a nil superclass — proceed through this. Also, it is not possible to redefine the structure of a class (e.g., the number of instance variables) if it has no superclass.

2. Possible problems

Errors—Much of the system (e.g., the debugger) depends on messages like printOn: and class working. If the encapsulator is faulty, and these do not work, things will go badly wrong, perhaps requiring a restart of the system. While debugging the encapsulator, make it a subclass of Object. Note that VisualWorks adds some methods (like class) automatically when the superclass is set to nil.

Incomplete encapsulation—If the encapsulated object passes a reference to itself to another object, or returns a reference to itself, then messages may reach the object bypassing the encapsulator. This is even more likely if the encapsulator is created well after the encapsulatee. The use of become: may help to capture references, but beware of attempting to encapsulate system objects in this way!

Primitive failure— Some primitives will fail if an argument is not of a specific class (e.g., primitive addition between integers). In this case an encapsulator will not work.

Ex 4. (Easy) File in `MessageTracer.st`, browse the implementation, and try the examples.

Ex 5. (Moderate) Build an encapsulator class Lazy that encapsulates a block. When the encapsulator receives a message, it should evaluate the block, holding on to the result object, and passing the message and all future messages on to the result object. For example,

```
f := [100 factorial] lazyValue.
```

should build the encapsulator, but the factorial should only be computed when f is later sent a message, e.g.:

```
Transcript show: f printString
```

Ex 6. (Harder) Build a similar class Future that commences evaluation of the block immediately, and in parallel (by sending the block fork). Should a message arrive before evaluation of the block has completed, suspend the sending process, and forward the message when the block has exited. (See *Writing Concurrent Object-Oriented Programs using Smalltalk-80,* by Trevor P. Hopkins and Mario I. Wolczko, Computer Journal, **32**:4, 1989, pp.341–50, for a description of the future evaluator.)