execute the loop: it creates a block whose code is [supplier atEnd] ... . This block becomes the block variable of a new BlockWithExit as a result of the withExit message being sent. theLoop is set to the BlockWithExit just created. When theLoop is sent the message value, the value method in BlockWithExit first creates another block, the exitBlock, which, if evaluated, will return to the sender of value *regardless of how many other activations have intervened*. The value method in BlockWithExit then sends value to the original block, causing it to execute. If no exit is sent, the loop completes normally. If an exit is sent, the exitBlock is evaluated and control returns to the last statement of maxBefore1000, just as if the loop had completed.

Dynamic Binding

Another common kind of infrequent event is a request for information. For example, suppose we want to specify a default directory for disk files throughout some part of a program. We could pass this information as an argument through all intervening calls, but this would place an added burden (in time, space, and complexity) on many parts of the program that have no interest in this information. An alternative would be to set a global variable before starting the computation, and reset it afterwards; unfortunately, if the computation is interrupted (say by something like the loop exit construct we described earlier), this leaves the variable with the wrong value. Ideally, we would like to set up a structure that will get control if the default information is ever needed, without getting in the way of the rest of the program. Such an arrangement is called *dynamic binding*. We will illustrate how it can be used both for data and control.

Suppose we want to write something such as the following:

```
#defaultDirectory bindTo: 'Smith' in:
    [someComputation]
```

and then have the file system be able to ask for the current default directory by:

```
#defaultDirectory binding
```

Since we want the binding of defaultDirectory to 'Smith' to last only for the duration of someComputation, it follows that in order to find the binding of a dynamic variable, we must examine the data structures that Smalltalk uses to represent the state of a computation. In

class name	Binding
superclass	Association "Provides key and value variables, and messages for accessing them"
instance variable names	"none defined here"
class messages and methods	

creation
of: aSymbol to: aValue in: aBlock | |
↑ self new of: aSymbol to: aValue in: aBlock

instance messages and methods

initialization
of: aSymbol to: aValue in: aBlock | |
key ← aSymbol.
value ← aValue.
↑ aBlock value "Actually does the computation"

class name (existing)	Symbol
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	

"none added here"

instance messages and methods

binding
bindTo: value in: aBlock | |
↑ Binding of: self to: value in: aBlock

Table 9: *Templates showing creation of a class template for class* Binding *(9a) and additions to existing class* Symbol *(9b).*

class name (existing)	Symbol
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
binding
binding | context |
    context ← thisContext. "Start here. thisContext is a
    machine register"
    [context = nil] whileFalse:
        [((context receiver isMemberOf: Binding)
            and: [context selector = #of:to:in: "Is it a
            binding..."
            and: [context receiver key = self]]) "...of this'
            variable?"
                ifTrue: "Yes, return its value"
                [↑ context receiver value]
                ifFalse: "No, go on to the next context in the
                chain"
                [context ← context sender]].
    self error: ('No binding for' concatenate: self )
```

Table 10: *Template showing additions to existing class* Symbol.

particular, even though many messages may be sent in someComputation before the file system needs to find the binding of defaultDirectory, there must be some way to search the stack of methods that have been started but not completed, looking for whatever represents the binding of defaultDirectory. In Smalltalk, each element of this stack is a MethodContext object, and the variable in a MethodContext that refers to its caller is called its *sender*. So searching this stack just means checking the current context's sender, its sender, and so on, until we find a binding of the variable. We know we have found a binding when we recognize a MethodContext in which the receiver of the message is a Binding (see tables 9a and 9b), and which was created in response to a particular message. During this computation (↑ aBlock value in table 9a), a MethodContext will exist in which the receiver is the Binding and the message is of:to:in:. This is how we recognize a binding in the stack of MethodContexts. The searching process is shown in table 10.

Note that by combining dynamic binding with the ability to name exit points (eg: by doing #theExit bindTo: to create a BlockWithExit), we can arrange for dynamically bound exceptional events to stop a computation in midstream. More complicated arrangements that allow the parts of the computation being stopped to clean up after themselves are also easy to construct.

Coroutines

Generator loops are an example of *producer/consumer*