Taking Exception to Smalltalk

Bob Hinkle* and Ralph E. Johnson

University of Illinois at Urbana-Champaign

Part 1

Exception handling is an important part of many languages. Although not provided in the original Smalltalk-80 or in Smalltalk V, it is supported in the latest version of Smalltalk-80 from ParcPlace. This paper will show how to build an exception handler for any version of Smalltalk and will use Smalltalk V/286 as an example. Along the way, we'll show you why it's useful for languages to treat seemingly internal mechanisms like processes and contexts as first-class objects.

This exception handler was first built for an early version of Smalltalk-80 implemented by Tektronix. It was modelled after a version described in an article by Evelyn Van Orden [1], and we used it in the type inference system of Typed Smalltalk [2]. When we ported Typed Smalltalk to ParcPlace Smalltalk, we wanted to use their faster exception handler, so we modified ours to be compatible. Thus, our exception handler is similar to ParcPlace's, though less powerful. We then developed the V/286 version described here, both to test the generality of the solution and to make the work interesting to a wider audience.

A Quick Look at Exceptions

Briefly speaking, exception handling is the provision for non-lexical flow of control in a program when something out of the ordinary (i.e., exceptional) occurs. An exception handler is a part of the program, and in Smalltalk usually a block, that can deal with some possible but unlikely event such as reading past end of file, dividing by zero, or referencing out of bounds in an array. In the usual scheme, a program registers an exception handler for a particular kind of event and then continues with its normal

---

* Supported by a fellowship from the Fannie and John Hertz Foundation.

processing. If an exceptional event does occur, a signal is raised as a notification to the system. The system finds the last handler that was registered for that signal by searching down the context stack. If one is found, control passes into the exception handler. Depending on the system, the handler will have different options; the handler can usually make whatever changes are necessary, and then execution can resume where the signal was raised, resume where the handler was registered, or return from where the handle was registered.

This description shows that implementing an exception handler requires access to processes and their context stacks. An exception needs to search the context stack to find the correct handler for a given signal and to implement non-local control flow. As a result, exception handling could only be added to traditional languages by the language designer. However, in Smalltalk, where processes are objects and contexts can be objects, exception handling can be added by a programmer. Smalltalk's first-class treatment of contexts is one aspect of a concept called reflection, which is the idea that languages and systems should objectify their internal mechanisms to make them accessible to the programmer. In that way programs can monitor and change their behavior, in a sense reflecting on themselves. Our example of exception handling shows how some reflectiveness makes a language more adaptable.

This article and its sequel next month present a Smalltalk implementation of exception handling. In this month's article we describe the system's interface and the machine-independent aspects of its implementation. Next month's article completes the picture by describing the V/286-specific implementation.

The Exception Handling Interface

At the heart of the exception handling system are the classes Signal and Exception. An instance of Signal represents an exceptional event that might occur, and its most important methods are handle:do: and raise. Sending handle:do: to a Signal object registers a block that can be evaluated if that event occurs. For example, suppose OutOfBoundsError is a global variable that holds a Signal object. As the name implies, this signal is intended to signify out-of-bounds references in arrays, and it might be used in a method of class Array as follows:

```
checkFifthElement
    OutOfBoundsError
        handle: [ :exception | ^self handleException: exception]
        do: [ ^self at: 5]
```

The effect of the handle:do: is to evaluate the second parameter (the do: block), with the addition that a raised OutOfBoundsError will be handled by evaluating the first parameter (the handle: block). So, evaluating #(1 2 3 4 5) checkFifthElement will return 5, as you might expect, but evaluating #(1 2 3 4) checkFifthElement will cause the block [:exception | self handleException: exception] to be evaluated. What happens then depends on Array>>handleException:--it might define a default value for that array, or prompt the user for information, or form some other appropriate response.

For this scheme to work, the system must use OutOfBoundsError to signify the out-of-bounds condition. This can be done by sending the raise message to OutOfBoundsError in the midst of at: (and methods like it), as follows:

```
at: anIndex
    <primitive: 60>
    (self outOfBounds: anIndex)
        ifTrue: [^OutOfBoundsError raise]
```

One interesting aspect of the handleException: message is its parameter "exception," which is an instance of the class Exception. Each time a signal is raised, a new exception is created to objectify that fact. The exception is a convenient place to encapsulate information about both the signal and the context in which it was raised. Particular error information or a special error message can be associated with an exception by using variations of the raise message, in this case raiseWith: and raiseErrorString:, respectively. In this way, an exception handling block can learn a great deal about the error by accessing the exception, which allows it to respond more intelligently.

In addition, class Exception provides support for common exception-handling techniques, including the messages proceed, reject, restart, and return. When an exception proceeds, control resumes in the context where its signal was raised, and a value can be returned if desired. This is how a new default value can be defined for an array. Thus, handleException: could be implemented as:

```
handleException: anException
    anException proceedWith: 'Bob'
```

This will cause the string 'Bob' to be returned as the value for any index outside the array's bounds. In addition to proceed, you can send restart to an exception, which causes the handle:do: context to be restarted, or send return, which causes the handle:do: message itself to return, again with the option of returning a specified value. Finally, sending reject to an exception is a way of saying that the current handler can't solve the problem. The system looks for the next handle:do: context down the stack that can handle the signal and evaluates its handle: block. These different possibilities are illustrated in Figure 1.

For the purposes of this example, we assume that Array>>foo is implemented as

```
foo
    Transcript show: self checkFifthElement printString
```

Now, if #(1 2 3 4) foo is selected and evaluated, then when fetchHandlerBlock returns, the context stack will be as shown in Figure 1, with the exception's instance variables signalContext and handlerContext referring to the indicated contexts.

1	Exception propagatePrivateFrom:	
2	Exception propagateFrom:	
3	Signal raiseWith:startingAt: extraString:proceed:	
4	Signal raise	
5	Array at:	← signalContext
6	Signal handle:do:	← handlerContext
7	Array checkFifthElement	
8	Array foo	
	⋮	

Figure 1: Stack during exception handling

There are several ways to define Array>>handleException:. One possibility is for it to proceed from the exception, as in:

```
handleException: exception
    exception proceed
```

In this case, when the handle: block of handlerContext is evaluated, nil will be returned as the value of the Array>>at: message send, the fifth context on the stack, and execution will proceed in the sixth context. However, if handleException: is defined as

```
handleException: exception
```

```
        exception return
```

then nil will be returned as the value of the Signal>>handle:do: message send corresponding to the sixth context on the stack, and execution will proceed in the seventh context. Using restart, as in

```
handleException:  exception
        exception restart
```

will cause the handlerContext, the sixth context on the stack, to be restarted from the beginning, in effect re-evaluating the do: block. Finally, the exception handler may reject the Exception, as in

```
handleException:  exception
        exception reject
```

In this case Exception>>propagatePrivateFrom: will be called again, but this time the search for a handler will proceed downward from the context just below the handlerContext, in this case the seventh one on the stack.

There is one final part of the system that interacts with exception handling, though it's not implemented in either of the above two classes. This feature is something called an unwind mechanism, which is a way for a programmer to ensure that certain actions are performed, even if a context is skipped during exception handling. For example, when an exception does a proceed, a restart, or a return, the flow of control jumps into lower contexts on the procedure's stack, and any higher contexts are removed from the stack without ever returning to them. This creates a potential problem: these contexts that were skipped might have performed some clean-up actions, such as closing files or releasing semaphores, if they'd been allowed to finish execution and return normally. Skipping these contexts during exception handling means skipping those important clean-up jobs. The solution to this problem is to define a special method, whose purpose is to ensure clean-up blocks will be executed, even in the presence of exception handling. The name of this method in Smalltalk-80 is valueOnUnwindDo:. Assuming aCollection is defined, evaluating

```
[aCollection checkFifthElement]
        valueOnUnwindDo: [Transcript show: 'Time to clean up!']
```

will cause the first block, [aCollection checkFifthElement], to be evaluated. If aCollection has five or more elements, the value of the fifth element will be returned, and nothing more needs to be done. However, if aCollection has four or fewer elements, and if the exception handler for OutOfBoundsError would cause control to return past the context of the valueOnUnwindDo: method (in effect skipping it), the second block will be evaluated, allowing any clean-up or finalization to be done. In Smalltalk-80, unwind blocks are even executed if they're skipped by a normal method return, because up-arrow is treated just like a return from an exception. In V/286, though, the meaning of up-arrow is hardwired into the virtual machine, so we can't duplicate this behavior.

The Machine-independent Implementation

Though an implementation of exception handling inevitably delves into system-specific code, much of our solution is system-independent--in fact, the same implementation of class Signal is used for the Tektronix and Digitalk platforms (and it could also be used for ParcPlace), and most of class Exception is common as well. This section considers the system-independent aspects of the exception-handling package.

To begin with, there are a number of pre-defined signals, all of which are defined in the Signal class>>initialize method and accessible using messages to Signal. These basic signals include ones for unhandled exceptions and keyboard interrupts. In addition to these, there is a class variable called ErrorSignal that is added to Object (just be careful how you add it!) and is accessible using Object>>errorSignal.

To create a new signal, you send the message newSignal to an existing signal. So, for example, we could create the signal OutOfBoundsError by evaluating

```
OutOfBoundsError := ErrorSignal newSignal
```

either in a workspace or (more likely) in a class initialization method. The newSignal method creates the new object and sets its parent instance variable to the receiver. The parent variable in class Signal is used to provide more structure in signal handling--when a signal is raised, it can be handled by an exception handler for the signal, or by one for the signal's parent, or by one for any of the signal's ancestors. In this way a programmer can define some general response for a tree of signals by registering a handler for the signal at the root. This response can then be specialized by registering more specific handlers for the signals further down in the tree.

Once a signal has been defined, sending it handle:do: registers an exception handler for it. The code for handle:do: is:

```
handle: handlerBlock do: doBlock
    "Evaluate doBlock. If all goes well, return its value. If an
    exception occurs then the returned value could be generated
    by evaluating returnBlock."

    | returnBlock |
    returnBlock := [ :value | ^value].

    ^doBlock value
```

This method's most significant role is as a placeholder. Its basic function is simply to evaluate its second parameter, the do: block. But in addition it marks a place on the context stack, so the system can find an appropriate handler when an exception occurs. How this happens will be explained next month when we consider Exception>>fetchHandlerBlock:. The block stored in the returnBlock temporary variable is used to make implementing Exception>>return easier.

The only other method we mentioned for class Signal was raise. As we said before, there are actually many variations of the raise message, depending on whether the exception handler can proceed through the exception, whether there's a parameter or error string needed, and so on. All these various raise combinations end up calling the same private method, which is Signal>>raiseWith:startingAt:extraString:proceed:. This is implemented as follows:

```
raiseWith: parameter startingAt: context
extraString: str proceed: aBoolean
    "Create a new exception and have it look for handlers
    starting at context."

    | exception |
    exception := self newException
                    signal: self
                    parameter: parameter
                    extraString: str
                    proceedBlock:
                        (aBoolean
                            ifTrue: [[:value | ^value]]
                            ifFalse: [nil]).
    ^exception propagateFrom: context
```

This method creates a new instance of Exception, passing the signal as one of the parameters in the creation message. In addition, if aBoolean is true, the signal is proceedable, which means that the handler is allowed to send the exception the proceed message, in effect declaring the error completely resolved and causing a return from the raise message send. If it is proceedable, the new exception will be passed the block [:value | ^value]. Like returnBlock in the handle:do: method, the block here simplifies our implementation, in this case making Exception>>proceedDoing: much simpler. Finally, this new exception is sent the message propagateFrom: with the context passed in as a parameter. This begins the process of finding a handler for the exception.

Exceptions have five instance variables: signal, parameter, extraString, proceedBlock, and handlerContext. The first four are set by the signal:parameter:extraString:proceedBlock: message, which is sent by a signal when the exception is created. The value of proceedBlock, if it isn't nil, is the [:value | ^value] block we saw above. After creating a new exception, a signal sends the propagateFrom: message, which in turn calls the propagatePrivateFrom: method. In addition to error handling, propagatePrivateFrom: sends the message fetchHandlerBlock: to find the right handler for the exception (and in the process it sets the instance variable handlerContext to the appropriate handle:do: message's context), and then it evaluates that handler. The implementation of fetchHandlerBlock: is described in next month's system-dependent section, since it depends on the layout of contexts.

Once the handler block is found, it's evaluated with the exception as a parameter. This allows the handler block to send the proceed, reject, restart, and return messages to the exception, and also to query the exception for information about the error. Below are the implementations for proceed and reject--those for return and restart are in next month's article, since they depend on some specifics of the V/286 system.

Proceeding is simple: since we have the instance variable proceedBlock, all we need to do is evaluate it, perhaps with some meaningful parameter, as in:

```
proceedDoing: aBlock
        "Return the value of aBlock as the value of the raised
        signal. Unwind the stack up to that point and resume
        execution in the context that raised the signal."

        | answer |
        answer := aBlock value.
        signalContext unwindLaterContexts.
        proceedBlock value: answer
```

Evaluating proceedBlock will cause control to return into the context where the signal was first raised. The only subtle thing is to remember the unwind mechanism: before evaluating proceedBlock we call unwindLaterContexts, which will evaluate the unwind blocks of every context we'll skip by proceeding.

Implementing reject is also quite simple. The current handler context (as found by fetchHandlerBlock:) is stored in the exception's handlerContext instance variable. So, to find the next handler below the current one, we just need to look for some handler for the exception's signal that is below handlerContext, and we can do that by sending propagatePrivateFrom: to the receiver exception with handlerContext as the parameter.

At this point we have a system-independent implementation for much of our package. The class Signal is complete, and we need only three more methods for class Exception: return, restart, and the private method fetchHandlerBlock:. We also need to implement `unwindLaterContexts` to implement our unwind mechanism. Finally we need some extra functionality for class `Process`, and we will see the need to create a new set of context-related classes to make dealing with contexts in V/286 consistent and relatively trouble-free. Next month we will describe these final aspects of our system.

References

[1] Evelyn Van Orden. *Application Talk*, HOOPSLA! Vol. 1, Number 2, Jan. 1988.

[2] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1989.