Chapter

20

Error Handling

The usual way of handling errors in a procedural language such as C is to return a status from a function call, then check the status in the calling function. If there is a problem, you may take some action in the caller, or more usually, you return from the caller, passing back the status code and possibly logging an error message. Typically, you end up with a lot of code whose sole purpose is to handle errors. It’s not uncommon for the majority of the code in a robust application to be associated with error handling.

Smalltalk makes it much easier to focus on the domain problem you are trying to solve by eliminating much of the error handling code. It does this by providing a wonderful mechanism called exception handling. We'll start by taking a look at basic exception handling, then, since there are a lot of additional capabilities, we'll look at these finer points later in the chapter.

The general idea behind exception handling is that you assume everything works, and so you don't check return values. Of course, sometimes things will go wrong. Perhaps you try to divide by zero, or perhaps you try to write to a read-only file. These events will raise an exception, which makes its way up the stack, looking for a signal that can handle it. Rather than checking a status at every level in the call stack (ie, in every method), you simply need to have an exception handler at the appropriate level in the code where you can take some action. This is illustrated in Figure 20-1.

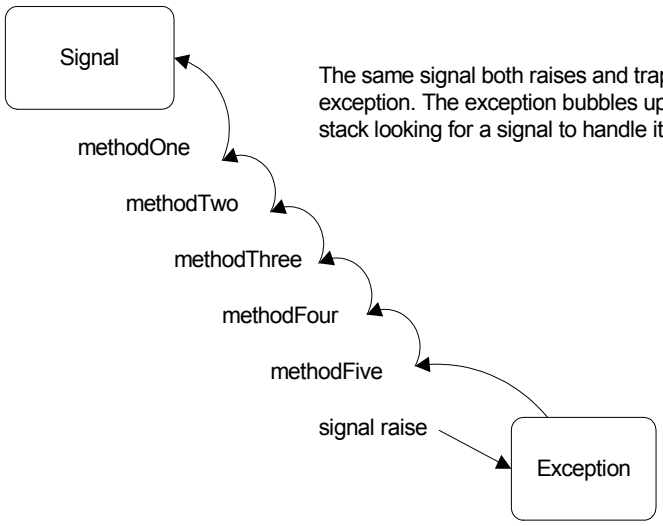

Figure 20-1.
The exception handling mechanism.

To trap, or handle, an exception, you send the `handle:do:` message to an instance of *Signal*. This message takes two blocks as parameters. The parameter to `do:` is the block that you want to execute, usually consisting of several lines of code. The parameter to `handle:` is the block that will be executed if an exception is trapped as a result of a signal being raised while executing the `do:` block. This block takes one parameter, the exception that was raised. The exception contains information about the error which you can use in the handler block. You'll usually see the exception parameter called `:ex`. Let's take a look at how code might look when using an exception handler.

```
MyRequest>>processYourself
   | tape drive |
   MySignals appSignal
      handle: [:ex | self myHandleFailure: ex]
      do:
         [self myCheckRequest.
         tape := self myReserveTape.
         drive := self myReserveDrive.
         tape mountYourselfOn: drive.
         self myRespondSuccess].
   self myReleaseTape: tape.
   self myReleaseDrive: drive

MyRequest>>myReserveTape
   self tapeName isReserved ifTrue:
      [MySignals appSignal raiseWith: #tapeBusy].
   ...code to reserve the tape...
```

In `myReserveTape`, we raise an exception if the tape is already reserved. In `processYourself` we have a signal to trap these raised exceptions. You'll see that we don't explicitly test the results of any of the message sends in `processYourself`, relying on the signal to trap the errors. This makes the code much more compact and easy to read. It allows us to focus on the problem we are trying to solve, rather than on the possibilities that things will go wrong.

Let’s now look at an example of how you would trap an exception raised by the system classes. The classic example is division by zero. You can find the signals in the *Signal constants* protocol on the class side of ArithmeticValue. (In Chapter 29, Meta-Programming we see how to find all the classes with a specified protocol.) If you evaluate the following code, you will get a message displayed on the Transcript.

```
ArithmeticValue divisionByZeroSignal
   handle: [:ex | Transcript cr; show: ex errorString]
   do: [ 4 / 0 ].
```

Signals and Exceptions

There are two classes that play a role in exception handling: *Signal* and *Exception*. The basic idea is that a signal both raises and traps an exception. When a signal raises an exception in a method, the exception code works its way up (unwinds) the message stack looking for a signal to handle it. Signals that can handle the exception are the signal that raised the exception, and any signal in the raising signal's parent hierarchy (more on this later). If the exception can't find a signal to handle it, it asks *Object noHandlerSignal* to raise an unhandled signal exception. If this new exception can't find a handler, it invokes the EmergencyHandler. The default EmergencyHandler puts up a Notifier window that displays information about the original exception and gives you the option to bring up a Debugger.

The beauty of this scheme of raised exceptions and exception handlers is that you don’t have to have lots of error handling code. Instead, at the level that knows what action to take in response to an error, you create the exception handling code. Then if a lower level method comes across an error, it simply raises an exception.

Creating your own signal

Exceptions can be raised by the system classes, and your code must be able to handle these. However, for your application, you should raise application defined signals and trap them with application defined signals. When you write your application, you may be several methods deep and realize that you can’t continue because of some error. Perhaps you have bad data, perhaps some resource you need is already in use, perhaps some condition has not been satisfied. In any of these cases, you will need to raise an exception that you will trap at a higher level. So you must first create a signal to use.

The easiest way to create a signal is to do `Signal new`, which creates a new signal with *Object errorSignal* as its parent. An alternative way to do the same thing (and, in fact, `Signal new` does this) is `Object errorSignal newSignal`, which we could also do with `self errorSignal newSignal` if our class is subclassed from *Object*. We'll talk more about the details of signal creation later. Now let's create a signal that we can use in our application. We have a couple of choices: we might create several signals, each appropriate for a specific situation, or we might create a more general signal that the whole application can use.

A single application signal

One approach is to create a single signal for the whole application. This works well in applications where the response to an error is the same, no matter what the error. I've used this approach in applications where we were processing requests from other computers, with no user interaction. If we detected an error, we simply rejected the request, passing back an error number and error message in the response. Here's an example of how we might create and access such as signal.

```
MySignals class>>setValues
   (AppSignal := Object errorSignal newSignal)
      notifierString: 'Application Error:';
      nameClass: self message: #appSignal
```

```
MySignals class>>appSignal
   ^AppSignal
```

Multiple application signals

Applications that are more user-interface intensive usually have a need for multiple application signals. For example, you might have one or more signals for file errors, one for validation errors, and so on. Then, for example, any time you try to validate something you assume that the validation works unless a validation signal is raised. To create and access these signals we might have something like:

```
MySignals class>>initialize
   AppSignal := Object errorSignal newSignal notifierString:
'Application Error:'.
   ValidationSignal := AppSignal newSignal notifierString:
'Validation Error:'.
   FileSignal := AppSignal newSignal notifierString: 'File Error:'.
   FileNotFoundSignal := FileSignal newSignal notifierString: 'File
Not Found:'.

MySignals class>>fileNotFoundSignal
   ^FileNotFoundSignal

MySignals class>>validationSignal
   ^ValidationSignal
```

Note the signal hierarchy we set up in the `initialize` method. AppSignal is our top level signal, with ValidationSignal and FileSignal as children. FileNotFoundSignal is a child of FileSignal. Thus we can trap file not found errors with any of FileNotFoundSignal, FileSignal, AppSignal, or Object errorSignal.

Overriding errorSignal

Sometimes you may decide to override `errorSignal` to provide different catchall error handling behavior for a class or for a hierarchy of classes. If you override `errorSignal`, you'd like your new behavior to be invoked rather than the default behavior. For example, suppose you have overridden `errorSignal` for *MyClass* and you are sending a message to an instance of MyClass that may generate any number of exceptions. You would ask your class rather than Object to handle the error. Your handling code will now look like the following.

```
anInstanceOfMyClass class errorSignal
   handle: [ :ex | ex return ]
   do: [ anInstanceOfMyClass processYourself ].
```

There are two points to note with this technique. First, you can use it as a standard technique because if a particular class hasn't overridden `errorSignal` it will inherit the default `errorSignal` from Object. Second, the behavior you write in your `errorSignal` method must be able to handle signals that are generated by any object since you can't guarantee that the instance of MyClass is the only object that will raise an exception.

Another approach is to always use `self errorSignal` as the catchall signal handler rather than `Object errorSignal`. This gives you the opportunity to override the catchall behavior in the class sending the

message rather than using the behavior inherited from Object. If you don't override `errorSignal`, then it is simply inherited.

```
MyClass>>doSomething
   self errorSignal
      handle: [ :ex | ex return ]
      do: [ self myDoSomethingElse ].
```

Central error messages

Let's take a look at what we might do in the `myHandleFailure:` method in the example at the beginning of the chapter. We have the exception since it was passed in. Now we need to report or log the error condition. How we do this is very application specific, but let's look at one way of transforming the exception into something a human would understand.

We do this with a central error message facility that contains the error number and error text. The text will directly tell what is wrong, and the number will allow the user to consult an error codes manual for more information about what this error means. In the class side initialization of *MyErrorMessages* we create an *IdentityDictionary* containing instances of *Association*, each association being a message number and the corresponding message text. The dictionary keys are the symbols that are passed as parameters when raising the signal. To retrieve the message number and the message text, we send `numberFor:` and `textFor:` respectively. (You can find code for this example in the file `errormsg.st`.)

```
MyErrorMessages class>>initialize
   "self initialize"
   (Messages := IdentityDictionary new)
      at: #tapeNotFound put: 1 -> 'The specified tape could not be
found';
      at: #tapeBusy          put: 2 -> 'The tape is in use'.

MyErrorMessages class>>numberFor: aSymbol
   ^(Messages at: aSymbol ifAbsent: [0 -> nil]) key.

MyErrorMessages class>>textFor: aSymbol
   ^(Messages
      at: aSymbol
      ifAbsent: [nil -> 'Code ', aSymbol printString, ' not found'])
value.
```

Now we just need to see how we would use this message facility when handling the failure. In `myHandleFailure:` we extract the error number and the error text from the centralized error message facility. Next, we append any additional text that was given when the signal was raised. Then we take some domain specific action to report the error.

```
MyRequestClass>> myHandleFailure: anException
   number := MyErrorMessages numberFor: anException parameter.
   text := MyErrorMessages textFor: anException parameter.
   anException localErrorString notNil ifTrue:
      [text := text, ' - ', anException localErrorString].
   .... now do something domain specific with the number and text
....
```

In this example, we generate the message text when we trap the exception. This is acceptable in situations where we don't need to add information about the values of the objects at the time of the error. In other situations we may need to also log or display the values of objects. To pass back information, we could create an *ExceptionParameter* class that contained the error symbol, a collection of parameters, and possibly an additional error string. We would then raise the exception with `raiseWith:`, passing our ExceptionParameter as the parameter, and have the exception handling code construct an error message from the ExceptionParameter. (An alternative approach is to use the mechanisms described below in the section on Parameter Substitution, and have the exception create a fully formed string.)

The scheme described above is fairly simple since we only have error numbers and error text. In a production system we would probably have a *MyMessages* superclass that defined most of the behavior, then *MyErrorMessages* and *MyInformationMessages* as subclasses. We would also store more information with each message, such as the message severity and message categorization to help route error responses and error reporting. One approach would be to code this information into the message string, but this would not be very object-oriented. A more powerful and more object-oriented approach would be to create *MyMessage* objects to put in the dictionary, rather than instances of Association. We might have something like the following. It would be straightforward to write methods to extract the message text and categorize and sort it for easier viewing.

```
MyErrorMessages class>>initialize
   "self initialize"
   (Messages := IdentityDictionary new)
      at: #tapeNotFound
      put: (MyMessage
         number: 1
         severity: #warning
         routing: 'TN'
         text: 'The specified tape could not be found');
      at: #tapeBusy
      put: (MyMessage
         number: 2
         severity: #warning
         routing: 'TN'
         text: 'The tape is in use').
```

The severity codes might include #information, #warning, #error, #fatal, while the routing information tells what type of message this is. In this example, T refers to tapes and N refers to the non-existence of things. We might further encapsulate information and create a specific routing object rather than code routing information into a string. We might also create subclasses of MyMessage for errors and warnings (for example, MyErrorMessage and MyWarningMessage). By having subclasses that encapsulate the information, we could later decide to encapsulate behavior specific to the message type.

The severity and routing information allows clients of the messaging system, such as other computers, the error log, and the system console, to register for the types of messages they want to receive. If an error message is created in response to a request, the message would be sent to the originator of the request. It would also be sent to a central message router that would distribute it to all message clients whose registration criteria matched the information in the message.

If you are using ENVY, you might consider using the provided error reporting mechanism, which uses the *ErrorReporter* class. There is more about ENVY in Chapter 34, ENVY.

More on Signal Creation

We've seen the basics of creating a signal, so now let's take a look at some of the details. Signals can be proceedable or non-proceedable, you can specify the string to be displayed in a Notifier window, and you can specify the message that will return the signal. Additionally, we'll take a look at parents of signals, and how signal trapping works within the parent hierarchy.

Specifying the proceedability of the signal

Signals can be proceedable or non-proceedable. A proceedable signal allows the exception handler to proceed from the point where the exception was raised. In general you won't do this and so won't care about this feature, but it's useful to know. When you send `newSignal`, you inherit the proceedability of the parent signal. `Object errorSignal` is proceedable, so `Object errorSignal newSignal` gives you a new proceedable signal with `Object errorSignal` as its parent. If you care about the proceedability, you can use the `newSignalMayProceed:` message with a Boolean parameter that specifies whether the signal is to be proceedable. For example, `Object errorSignal newSignalMayProceed: true` gives you a proceedable signal. If you want to be able to proceed from an exception, you must raise exceptions by sending your signal a message from the `raiseRequest` family rather than the `raise` family.

Notifier strings and inspector information

There are two other aspects to consider when creating a new signal. If you send `notifierString: aString` to an instance of Signal, you set the default string that is returned by the `errorString` message — the string that will be displayed at the top of any Notifier window raised by this signal. If you send `nameClass: aClass message: aSymbol` to the signal, you are by convention specifying that you can get this signal by sending the message specified by aSymbol to the class specified by aClass (this is really just a documentation aid; you'll still have to write the method that returns the signal). Additionally, aClass and aSymbol are stored in instance variables in the signal and give inspectors more information to display. For example, you might create a signal by doing the following, also remembering to write the `appErrorSignal` method.

```
(AppErrorSignal := self errorSignal newSignal)
   notifierString: 'Application Error: ';
   nameClass: self message: #appErrorSignal
```

Parents handling signals

Earlier we trapped a division by zero error using `ArithmeticValue divisionByZeroSignal`. Now let's trap the same error with the parent signal, `ArithmenticValue domainErrorSignal`. Then we'll trap the signal with its parent, `ArithmeticValue errorSignal`, then with its parent, `Object errorSignal`. You can see how the signals are created from their parents in the class initialization of ArithmeticValue.

```
ArithmeticValue domainErrorSignal
   handle: [:ex | Transcript cr; show: ex errorString]
   do: [ 4 / 0 ].
```

```
ArithmeticValue errorSignal
   handle: [:ex | Transcript cr; show: ex errorString]
   do: [ 4 / 0 ].

Object errorSignal
   handle: [:ex | Transcript cr; show: ex errorString]
   do: [ 4 / 0 ].
```

Since we also know how to create new signals of our own, let's see another example of how the parent of a signal can trap exceptions raised by the signal, but other signals derived from the same parent (ie, siblings of the signal) can't trap the exception. In our example, *parent1* and *parent2* are two signals with the same parent, Object errorSignal. *child2* has parent2 as its parent. If we ask parent1 to trap an exception raised by child2, it doesn't. If we ask parent2 to trap the exception, it does. Try the code below twice, once with parent1 handling the exception and once with parent2 handling it. Moral of the story: an exception can be trapped by the signal that raised it, or by its parent or grandparent, ..., up to Object errorSignal.

```
parent1 := Object errorSignal newSignal notifierString: 'Error1:'.
parent2 := Object errorSignal newSignal notifierString: 'Error2:'.
child2 := parent2 newSignal.
parent1
   handle: [:ex | Transcript cr; show: 'in handler']
   do: [child2 raise].
```

Signal collections

If your application can raise several different signals but the way of dealing with the exception is the same for all the signals, you have two alternatives. You could use a generic signal, such as *Object errorSignal*, or a common parent signal to handle all the possible exceptions. Or, if you know exactly which signals can be raised, you can use a *SignalCollection*. To do this, add the signals you want to trap to the SignalCollection, then send `handle:do:` to the SignalCollection. When a signal is raised, it looks at each signal in the collection until it recognizes one. You can build the SignalCollection in one place then reuse it in different places.

Try the following example twice, reversing the order of the statements in the `do:` block. You'll get a different exception reported on the Transcript.

```
sigCollection := SignalCollection
   with: Dictionary keyNotFoundSignal
   with: Dictionary valueNotFoundSignal.
sigCollection
   handle: [ :ex | Transcript cr; show: ex errorString]
   do:
      [Dictionary new at: 3.
      Dictionary new keyAtValue: 4].
```

How useful are SignalCollections? I've worked on applications where they were not very useful because it was easier to have the top level application signal or *Object errorSignal* do the trapping. I've also worked on applications where we used SignalCollections because we needed to be specific about the errors we were trapping. The VisualWorks system classes use a SignalCollection only two times, and one of those is an empty collection!

Handler lists / collections

If you have a set of signals you want to trap but you want to take different actions for each exception, you can use a set of nested handlers. For example,

```
signalOne
   handle: [:ex | some code]
   do: [signalTwo
      handle: [:ex | some code]
      do: [signalThree
         handle: [:ex | some code]
         do: [some application code]]
```

If there are a lot of signals you want to look for, the code will get pretty messy. The *HandlerCollection* (VisualWorks 2.0) or *HandlerList* (VisualWorks 2.5) class provides a shorthand way of doing the same thing. The handlers will still be nested when the code is executed, but the code is a lot tidier and is therefore easier to understand and maintain. Here's the above example coded using a HandlerList — note that since the handlers are really nested, the last signal added is the first signal that will get a chance to trap the exception. So put the most general signals first and the most specific last. Again, you can build the HandlerList in one place and reuse it in different places.

```
(handlerList := HandlerList new)
   on: signalOne handle: [:ex | some code];
   on: signalTwo handle: [:ex | some code];
   on: signalThree handle: [:ex | some code].
handlerList handleDo: [some application code].
```

How useful are HandlerLists? Again, I've worked on applications where we never used them. I've also worked on applications where we wanted to trap errors generated by the file system, and give the user a different message for each type of `error. In my experience, HandlerLists are most useful when handling errors generated by the interaction of the Smalltalk image with the operating system, where the action you take depends on the exact problem detected. The VisualWorks system classes use a HandlerList or HandlerCollection only once.

(In VisualWorks 2.5, HandlerCollection was replaced with HandlerList. Unless you have been subclassing from HandlerCollection, the behavior is identical.)

Passing information with an Exception

Let's now look at the various ways of passing information with the exception. We'll examine only non-proceedable signals; for proceedable signals you substitute `raiseRequest` for `raise` in all the examples.

To get a string describing the exception, you send the `errorString` message to the exception. The signal's notifier string and the parameters to the raise message are all used when creating the error string, and in the following examples, we will show how they affect the error string. Notifier windows display the value returned by `errorString`.

raise

The `raise` message simply raises an exception, passing back no information.

raiseWith: aParameter

The `raiseWith: aParameter` message passes back information in the parameter. To retrieve the parameter, send `parameter` to the exception. In our example above, we passed the symbol *#tapeBusy* as the parameter, and then used it to look up the error message.

raiseErrorString: aString

The `raiseErrorString: aString` message passes back a string. You can access the string by sending `localErrorString` to the exception. The value returned by `errorString` depends on spaces in aString. The rule is:

aString has no space in front - replace notifier string
aString has a space in front - append aString to notifier string

raiseWith: aParameter errorString: aString

The `raiseWith: aParameter errorString: aString` message passes back both a parameter and a string. You can access the parameter by sending `parameter` to the exception and the string by sending `localErrorString` to the exception. The value returned by `errorString` depends on spaces in aString. The rule is:

aString has no space in front - replace notifier string
aString has a space in front - append aString to notifier string
aString has a space at the end - append parameter to aString

Parameter Substitution

At the time you raise an exception you have access to all the information that is relevant to the error. VisualWorks provides a parameter substitution mechanism where you can raise an exception, passing a template string and a collection of parameters to be substituted. (If you are using a centralized messaging facility, you could get the message template from it, using a symbol as the lookup key.) The exception substitutes the parameters, generating a fully formed message. However, my preference would be to create the ExceptionParameter object that we discussed above since this provides more power and flexibility.

The parameter substitution mechanism is different in VisualWorks 2.0 and 2.5, so we'll look at both. In both messages below, the template is a string with embedded formatting information. The parameter is expected to be a collection, where each element of the collection corresponds to a formal parameter in the template string. Sending `parameter` to the exception will return the parameter collection, and sending `localErrorString` will return the expanded string. The `errorString` message conforms to the space rules shown in `raise:ErrorString:` above.

raiseWith:errorTemplate:

The `raiseWith: aParameterCollection errorTemplate: aString` message is the parameter substitution message used in VisualWorks 2.0. The template string contains embedded percent signs

(%). The string will be expanded before the handler gets to trap the exception, with the % being replaced by the `printString` representation of the appropriate collection element. For example, the following brings up a Notifier window with the message: *Unhandled exception: Received bad value 3 and bad value 'm:\temp'.*

```
Object errorSignal
   raiseWith: #(3 'm:\temp')
   errorTemplate: 'Received bad value % and bad value %.'
```

raiseWith:errorPattern:

The `raiseWith: aParameterCollection errorPattern: aString` message is the parameter substitution message used in VisualWorks 2.5. It provides a new, more powerful parameter substitution mechanism using a new family of messages (`expandMacros`). For more information on the parameter substitution options, see Chapter 12, Strings. For example, the following brings up a Notifier window with the message: *Unhandled exception: Received bad value 3 and bad value m:\temp.*

```
Object errorSignal
   raiseWith: #(3 'm:\temp')
   errorPattern: 'Received bad value <1p> and bad value <2s>.'
```

Handling the exception

There are several ways to handle the exception in the `handle:` block of the `handle:do:` message. Remember that the block will look something like `[:exception | some code]`, where the exception is passed in as the parameter. If you hit the right bracket of the handle block, the return value of the block (ie, the value of the last statement executed) will be the value returned from `handle:do:`. If there is no code in the block, the exception will be returned.

Alternatively, you can send a message to the exception as the last thing in the block — this is the preferred approach because it gives the exception a chance to gracefully unwind the stack. The messages you can send are `return`, `returnWith:`, `reject`, `restart`, `restartDo:`, `proceed`, `proceedWith:`, and `proceedDoing:`. We will look at some of these messages. For the others, you'll have to examine the system code and read the manuals.

return

The `return` message returns nil from the handle block. It's the same as doing `returnWith: nil`. Note that `return` leaves you in the method containing the exception handler. If you don't want to do anything in the handler block, it's good practice to do `ex return`. The following code prints *nil* to the Transcript.

```
Transcript cr; show:
   (self errorSignal
      handle: [:ex | ex return]
      do: [self errorSignal raise]) printString.
```

returnWith:

The `returnWith:` message allows you to set a value as the return value from `handle:do:`. The parameter to `returnWith:` will be the return value. Note that `returnWith:` leaves you in the method. The following code prints 3 to the Transcript.

```
Transcript cr; show:
   (self errorSignal
      handle: [:ex | ex returnWith: 3]
      do: [self errorSignal raise]) printString.
```

reject

The `reject` message allows you to take some action then reject the exception, or reject the exception based on some criteria. In the following example, in the handle block we print to the Transcript then reject the exception. Since the exception is rejected, it continues to look for a handler, but doesn't find another one, so it raises a Notifier window.

```
self errorSignal
   handle: [:ex |
      Transcript cr; show: ex errorString.
      ex reject]
   do: [self errorSignal raiseWith: 3 errorString: 'Error '].
```

restart

The `restart` message allows you to figure out what caused the exception to be raised, fix the condition, then restart the `handle:do:` expression. The following code gives you a chance to correct the error that you get if you divide by zero.

```
self errorSignal
   handle: [:ex |
      Dialog warn: 'You divided by zero'.
      ex restart]
   do: [| answer |
      answer := Dialog request: 'Divide 4 by what?' initialAnswer:
'0'.
      Transcript cr; show: (4 / answer asNumber) printString].
```

restartDo:

The `restartDo:` message allows you to figure out what caused the exception to be raised, possibly fix the condition, then restart execution with a different code block. While you can use `^` in a `handle:` block to return from the method, `restartDo:` is the recommended way of doing this. For example, `ex restartDo: [^nil]` exits the *method*, returning *nil*.

```
MyClass>>myMethod
   someSignal
      handle: [:ex | ex restartDo: [^nil] ]
      do: [some code that raises an exception]
```

self error:

Another way to raise an exception is to send the `error:` message, passing a string as a parameter. By default this raises a Notifier window and allows you to bring up a Debugger. To change the default behavior, you can override `error:`. If you are using the application signal mechanism described above and you have a single centralized message file in your image, you could write, for example, `Object>>error:args:string:` to take a symbol, an array of arguments, and an additional string as arguments. It would handle raising the exception in the appropriate way. To invoke it you would do something like

```
self error: #tapeNotFound args: (Array with: tapeName with:
driveName) string: nil.
```

A more elegant approach would be to have your own subclass of Object, say *MyObject*, and to defined `error:args:string:` on MyObject. All application objects that used to be subclassed from Object would now be subclassed from MyObject.

doesNotUnderstand:

If your code sends a message that is not understood, eventually the `doesNotUnderstand:` message will be sent. The default `doesNotUnderstand:`, implemented by Object, raises a Notifier window. You can override `doesNotUnderstand:` if you want to do anything specific such as logging an error.

valueNowOrOnUnwindDo:

Suppose you have code that should be executed both after normal completion of a sequence of statements, and if an exception is raised. You can wrap the regular code in a block and send it the message `valueNowOrOnUnwindDo:`, passing as the parameter a block containing the code you always want executed. A good example of this is closing a file. You'd like to close the file after successfully reading it, and also if an exception is raised while reading the file. Here's an example of how to use this technique.

```
stream := self myFilename readStream.
[ self readFile: stream ]
   valueNowOrOnUnwindDo:
      [ stream close ].
```

Emergency Exception Handler

If an exception can't be handled, the EmergencyHandler will be invoked. The default EmergencyHandler is defined during class initialization for *Exception*, and it raises a Notifier window. You can change the EmergencyHandler by sending *Exception* the class message `emergencyHandler:`, where the parameter is a block that takes two arguments: the exception and the context. For example, try the following (after saving any changes you want to keep).

```
Exception emergencyHandler:
   [:ex :context |
      Dialog warn: 'Quitting: ', ex errorString.
      ObjectMemory quit].
3 next.
```

Once the emergency handler has done its thing, it attempts to re-execute the code that caused the exception. So, unless the emergency handler quits the Smalltalk image or starts a new thread of execution, you will end up in an infinite loop.

noHandlerSignal

As we saw earlier, if an exception can't find a signal to handle it, it asks *Object noHandlerSignal* to raise an unhandled signal exception. If this new exception can't find a handler, it invokes the EmergencyHandler. However, the new *noHandlerSignal* exception is raised in the original context so we still have an opportunity to trap it. To see an example of how this works, create a class with the following methods.

```
MyClass>>methodOne
   self methodTwo

MyClass>>methodTwo
   Signal noHandlerSignal
      handle: [:ex | Transcript cr; show: 'methodTwo - ', ex
parameter errorString]
      do: [self methodThree]

MyClass>>methodThree
   self foo
```

If you now execute `MyClass new methodOne`, you will see the following on the Transcript:

```
methodTwo - Message not understood: #foo
```

The way this works is that `methodThree` sends the message `foo`, which is not implemented and thus raises a *messageNotUnderstood* exception. The exception cannot find a handler for itself anywhere in the stack, so it raises a *noHandlerSignal* exception, with the original exception stored in the *parameter* instance variable of the new exception. In `methodTwo` we have a handler for the new exception, so we trap it and print out some information. Note that since the *noHandlerSignal* exception contains the original exception in its *parameter* variable, we have to print `ex parameter errorString` to get information about the original error.

Since the *noHandlerSignal* exception is raised only if no handler is found for the original exception, we should be able to add an exception handler in `methodOne`, and have this invoked instead of the handler in `methodTwo`. Modify `methodOne` to look like the following:

```
MyClass>>methodOne
   Object messageNotUnderstoodSignal
      handle: [:ex | Transcript cr; show: 'methodOne - ', ex
errorString]
      do: [self methodTwo]
```

If you now execute `MyClass new methodOne`, you will see the following on the Transcript:

```
methodOne - Message not understood: #foo
```

This technique gives us a way to create a "last resort" exception handler. By wrapping our code in a *Signal noHandlerSignal* exception handler, we in effect say "I'll give someone else an opportunity to handle any exception that is raised, but if no one else wants to handle it, I'll take care of it myself."

Exceptions and Processes

Unfortunately, Smalltalk doesn’t do a good job of handling exceptions raised in forked processes. The problem is that the when a process is forked, it gets its own stack context. When an exception is raised in a forked process, it bubbles up the stack until it is either handled or it raises a Notifier window. Because the forked process has a different context than the process that forked it, the exception never gets into the stack of the forker.

This means that each forked process must have its own exception handling code if you want to trap exceptions. What you do next, after trapping the exception, depends on the application and how it is structured. If you have an application where there is a central dispatcher that gets requests and runs each request in its own forked process, each request could have an instance of SharedQueue which it uses to communicate back to the forker. For an example of this, see the section named *Running tests that time out* in Chapter 30, Testing.

For a more complete discussion of handling exceptions in a multi-process environment, see the two part article, Cross-Process Exception Handling, by Ken Auer and Barry Oglesby in The Smalltalk Report, January 1994 and February 1994 (the first part is mis-titled, being called Cross-Purpose Exception Handling).

When to look for errors

There are different philosophies on when to look for error conditions. The idea has made it into software engineering folklore that you should trap errors as soon as possible. This makes sense when we are dealing with user interfaces because it's reasonable to let users know about errors as soon as possible (of course, it's even better to write applications that prevent users making errors).

On the other hand, leaving aside user interface code, there's no reason to trap an error before it absolutely needs to be detected. There's no reason to have a whole hierarchy of methods all checking the validity of the same parameters. Most of them won't even care what the value is and will simply pass it on. It makes a cleaner application to check the validity of things only when it's vital that they are valid. Then, and only then, raise an exception.