Handling Exceptional Conditions

All real programs have to deal with exceptional conditions. Following a summary of different approaches to exception handling in VisualWorks, this module concentrates on describing classes Exception and Signal, giving examples of their correct usage.

An exceptional condition is something that is expected to occur infrequently, and does not fit into the usual pattern. More often than not the condition is detected in a piece of code that cannot properly deal with the condition, because the context in which it has been detected is inappropriate. Hence, the condition must be signalled to a wider context, i.e., the calling code.

1. When to use Exception Handling

You should *not* try to use exception handling code to:

- Deal with (local) hardware faults. Possible exception: network failures.
- Work round errors in the program. Software can fail in an infinite number of different ways, and you can't anticipate them all.

However exception handling code should be used to deal with infrequent, but anticipated conditions, such as:

1. **User Errors**. Rather than testing for all the numerous ways a user can make a mistake (for example when prompted for a number), you can use exception handling as a catch–all.
2. **Violation of usual preconditions.** With every piece of code there should be two sets of preconditions:
   - Those which, if violated, will cause the code to fail in arbitrary and undefined ways. Guaranteeing that these preconditions are met is the responsibility of the caller, and is not normally checked by the called code. For example, when passing a cyclic structure to a tree–walking algorithm, it is impractical and inefficient to test for these sorts of errors.
   - Those which are tested for and dealt with by code, and cause an exception condition to be passed back to the caller.xample: looking up a non–existent key in a data structure.

2. Different Approaches in VisualWorks

2.1. Exceptional Value

If there is a distinguished value which cannot be an otherwise sensible answer (e.g. nil), return that. Examples: returning 0 as the array index of a value which is not present in the array (e.g., from findFirst:); superclass of a class returns nil for Object.

2.2. Test for exception as a separate message

Provide a test message to indicate that a precondition would be violated. Example: isEmpty in collections; asking for the first element in a list does not make sense if the collection is empty.

This approach can be used if the test is inexpensive to compute, and does not significantly overlap with the algorithm itself, and no suitable exceptional value exists. Example in which this approach is bad: testing to ensure that a tree contains a particular value; the test for the missing value is the same as the search itself.

2.3. Returning a pair (triple, etc.) of values

One value in the pair indicates if an exceptional condition has arisen (and the other is undefined), otherwise the other value(s) is(are) the answers. This approach is rather unusual in VisualWorks, and is more likely to be seen in Pascal or C, thus best avoided.

2.4. Exception block

An exception block is activated should an exception arise, and can use the calling context. Example:

```
coll at: key ifAbsent: [self notFoundError]
```

If the algorithm must terminate when an exception is detected, ensure that it cannot continue should the exception block return, thus:

```
^exceptionBlock value
```

Exception blocks are sometimes stored in instance or class variables to avoid being passed around all over the place.

2.5. Multiple exception blocks, exception blocks with parameters

If there is more than one way an exception can arise, then it may be desirable to separate these. This may be achieved by using multiple exception blocks, for example:

```
reactant
    changeTemp: delta
    ifFreezes: [reactant expand]
    ifBoils: [vessel explode]
```

Alternatively, it may be better to use a parameterize the exception block, as below:

```
reactant
    changeTemp: delta
    ifStateChanges:
      [:state |
        state == #gas ifTrue: [vessel explode]]
```

Make sure in the comment that it is clear the block should accept parameter(s).

2.6. Sending a message

To self

Useful if the error can be meaningfully reinterpreted by a subclass.

```
self error: 'error message'
self errorOrSomething
```

To an error handler

For example, the compiler must behave differently for errors in source when filing in, and when run from a browser or workspace. The user interface to the error handler is encapsulated in separate classes. Instances of subclasses of CompilerErrorHandler (an abstract superclass), respond differently to the following message:

```
requestor
    error: #syntax
    with: string, '->'
    at: location
```

3. Signals and Exceptions

The exception handling mechanism provided by classes Signal and Exception provide the most general solution, but it is also the most complex. Don't use a sledgehammer to crack a walnut!

A Signal represents a *kind* of error. Signals are arranged in a hierarchy so that more general signals can be used to cover different kinds of error. For example, the division–by–zero signal is below the general arithmetic error signal in the hierarchy. The signal hierarchy, like the class hierarchy, is basically static. An Exception represents the dynamic invocation of a signal, i.e. the specific occurrence of an error. It therefore knows which signal was raised, and in what context.

Suppose we are building a calculator application. If the user attempts to divide 42 by 0, a Notifier will appear, saying

Can't create a Fraction with a zero denominator.

This isn't terribly meaningful to the average user. Besides, we don't want to present the user with a notifier or debugger: we want the application to recover and continue.

This is one case in which we can use a signal. We set the signal to watch over the part of our application that performs arithmetic. When things are going smoothly, the signal just observes. When a specific error occurs, the signal is notified and an *exception* is created to represent the error. The exception travels back down the message stack looking for a matching *exception handler.* When a handler is found, it receives control. The handler can execute recovery code, and then direct flow back to the point at which the error was raised, or abandon the error code and proceed down a different path (see Fig.1).

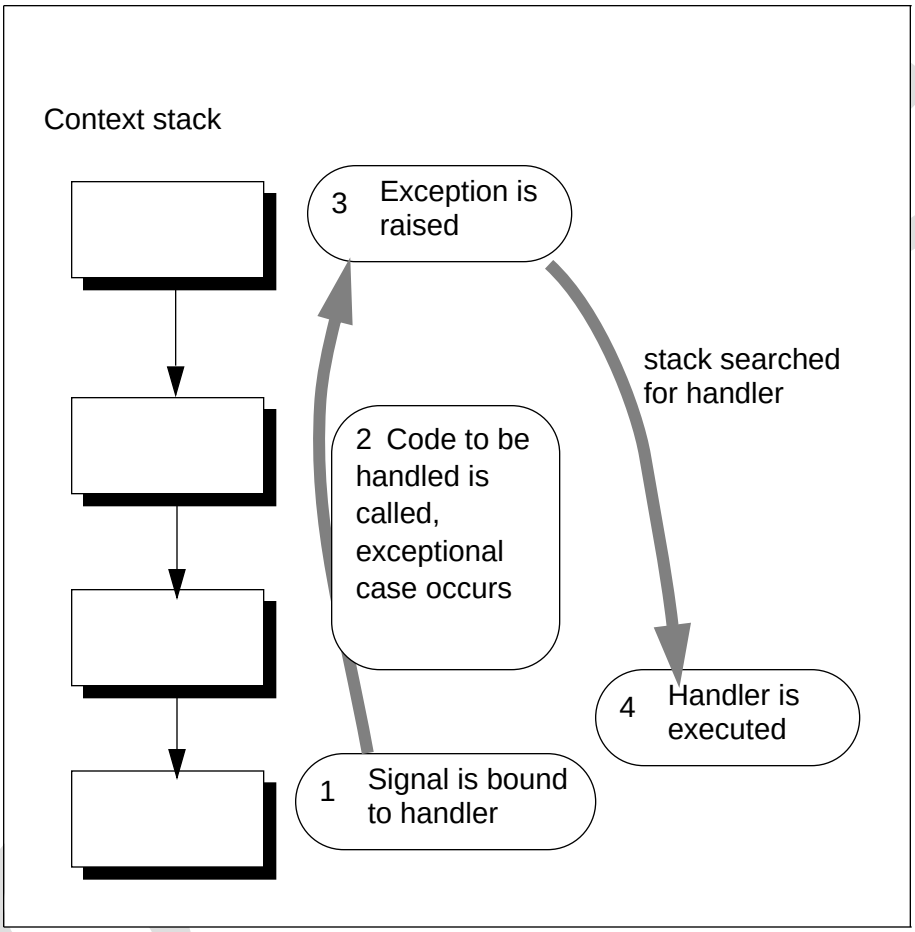

Figure 1: Exceptions and Signals

3.1. Handling Existing Signals

To use a signal, we must tell it what to *do* and how to *handle* an error. Hence, we send the handle:do: message to the appropriate instance of Signal. For example, to catch division–by–zero errors, send the message to

```
ArithmeticValue divisionByZeroSignal.
```

Signals are arranged in a hierarchy, with those higher up in the hierarchy covering all the error situations of those lower in the hierarchy. Hence, to catch all arithmetic errors, including division by zero, we could use the signal:

```
ArithmeticValue errorSignal
```

which is above divisionByZeroSignal in the signal hierarchy. See the VisualWorks manual for a list of predefined signals, or browse any 'Signal constants' class methods (e.g., in Object or Number).

The handle:do: message takes two blocks as arguments. The do: block is the code during which the exception handling takes place, the handle: block is the exception handler, and is passed the exception as an argument. For example:

```
result :=
    ArithmeticValue divisionByZeroSignal
      handle: [ :theException |
          Dialog warn: 'Division by zero!']
      do: [self doOperation]
```

3.2. Passing Control on from the Handler

The exception handler can deal with the exception in one of the following ways:

1. Each signal is either *proceedable* (i.e., the handler can instruct computation to continue), or *non–proceedable*. If the signal is proceedable, and is raised in a proceedable way (see later), then the handler can cause computation to proceed from the point of error by sending the exception one of the messages described in Table 1.

Message	Description
proceed	Return control to the point at which the error occurred.
proceedDoing: aBlock	Substitute aBlock, then proceed
proceedWith: aParameter	Return control to the point at which the error occurred, using aParameter as the returned value

Table 1: Proceed Messages

2. It can refuse to handle the exception, sending reject to the exception. The exception will continue its search down the message stack for a matching handler.

3. It can exit the handler block and the enclosing handle:do: by sending return to the exception (which returns nil from handle:do:). To return some other object, use returnWith: anObject.

4. The do: block can be restarted, by sending restart to the exception. To substitute the do: block with another block use restartDo: aBlock. Here's an example insists that the user supply a valid Date (by trapping all conversion errors in Date>readFrom:):

```
| date input |
input := Dialog request: 'Enter a date'.
Object errorSignal
  handle:
    [ :ex |
      input := Dialog request: 'Try again'.
      ex restart]
  do: [date := Date readFrom: input readStream].
Transcript show: date printString
```

If a handler doesn't explicitly specify one of these alternatives, it has the same effect as returnWith:, returning the value of the handler block.

Ex 1. Browse class PassingControl, and experiment with the class examples.

3.3. Using An Existing Signal

Sometimes you may need to produce a method that can create an exception directly. For example, if you create a new kind of indexed collection then you should create an exception if an access is made with an invalid index. In this case, there is an appropriate existing Signal we can use:

```
Object indexNotFoundSignal
```

To create an exception, send a Signal the message raise. If you wish the exception to be proceedable, use raiseRequest (or one of the request variants below). Alternatives are described in Table 2.

Message	Description
raiseErrorString: aString raiseRequestErrorString: aString	Provide a error string for the resulting Exception. If the first character of the string argument is a space then it is appended to the Signal's notifierString.
raiseWith: parameter raiseRequestWith: parameter	Provide a parameter for the resulting Exception. The parameter can be accessed by sending parameter to the Exception.
raiseWith: parameter errorString: aString raiseRequestWith: parameter errorString: aString	Combination of above. If the last character of the string argument is a space then it is prepended to the Exception's parameter.

Table 2: Raising Messages

If you want to find all the existing signals, evaluate:

```
Browser browseAllClassMethodsInProtocol: 'Signal constants'
```

Ex 2. Experiment with the examples in class RaisingExceptions.

3.4. Adding a new Signal

To create a new signal of your own, find an existing signal and make your signal a descendent of that. (Object errorSignal is the general "catch–all" error signal). This is typically done in a class initialization method, and the signal is stored in a class variable, for example:

```
MySignal := Object errorSignal newSignal.
```

If the original signal was proceedable, the new signal will also be. If you don't want the same proceedability, use newSignalMayProceed: aBoolean. You may also provide the signal with a notifier string, using the message notifierString:.

Ex 3. Browse class NewSignal, to see how a new signal may be created. What is the purpose of the nameClass:message: expression in the class initialization method?

Classes SignalCollection and HandlerList are provided to avoid lots of nested blocks. SignalCollection instances also understand handle:do:. HandlerList instances understand on: aSignal handle: aBlock (for each signal); the created instance should be sent the message handleDo: doBlock.

3.5. Example: Simulating a "Trigger"

VisualWorks doesn't have "triggers" as basic features, but you can implement them easier than in most other languages.

```
Object subclass: #TriggerObject
    instanceVariableNames: 'lowerLimit upperLimit value '
    classVariableNames:
      'TriggerUpperSignal TriggerLowerSignal'
```

Class methods

instance creation

```
newFrom: lower to: upper
    ^ self new initializeFrom: lower to: upper
```

class initialization

```
initialize
    "Create new signals"
    "TriggerObject initialize"

    TriggerUpperSignal := Object errorSignal newSignal.
    TriggerUpperSignal nameClass: self message: #upperSignal.
    TriggerUpperSignal notifierString: 'Upper Limit Failed '.
    TriggerLowerSignal := Object errorSignal newSignal.
    TriggerLowerSignal nameClass: self message: #lowerSignal.
    TriggerLowerSignal notifierString: 'Lower Limit Failed '
```

Signal constants

lowerSignal
^ TriggerLowerSignal

upperSignal
^ TriggerUpperSignal

Instance methods

initialize - release

initializeFrom: lower to: upper
lowerLimit := lower.
upperLimit := upper

public - value

value: aValue
aValue > upperLimit ifTrue:[TriggerUpperSignal raiseRequestWith: aValue].
aValue < lowerLimit ifTrue:[TriggerLowerSignal raiseRequestWith: aValue].
value := aValue

testing

test
"TriggerObject test"

| trigger |
trigger := TriggerObject newFrom: 0 to: 100.
^TriggerObject lowerSignal
handle: [:ex | Transcript cr; show: ex errorString.
ex proceed]
do: [trigger value: -10]

Ex 4. Which of the above approaches would you use to handle the following exceptional situations?

a. A menu pops up; the user selects none of the items.

b. A database is accessed for a record, by giving it a key; the key does not exist in the database.

c. The elements of an array are to be added together, returning the sum. What if the array is empty? What if one of the elements is not a number?

Ex 5. Add two extra testing methods to TriggerObject to handle both lower and upper limit range errors, using

a. a SignalCollection

b. a HandlerList.

Ex 6. Add a method to BlockClosure, logErrors, that executes a block (the receiver) and uses an exception handler to catch any exceptions, printing a message in the

Transcript when an exception occurs, and aborting the block. Test the code with the following expression:

```
['one' + 'two']
```

Ensure that the result of the block is returned if no error is encountered, nil otherwise. What might be the purpose of such a handler?

Ex 7. Define a new signal, Object loopBreakSignal, which is to be raised to exit from loops, in conjunction with a method, break, in Object, and a method, loop, in BlockClosure:

```
"returns an array with more than 14 elements"
[Array allInstances do:[:i | i size > 14 ifTrue:[i break]]] loop.
```

What should the signal's parent be? Write the loop and break methods. (We will return to this code in the optimization module.)