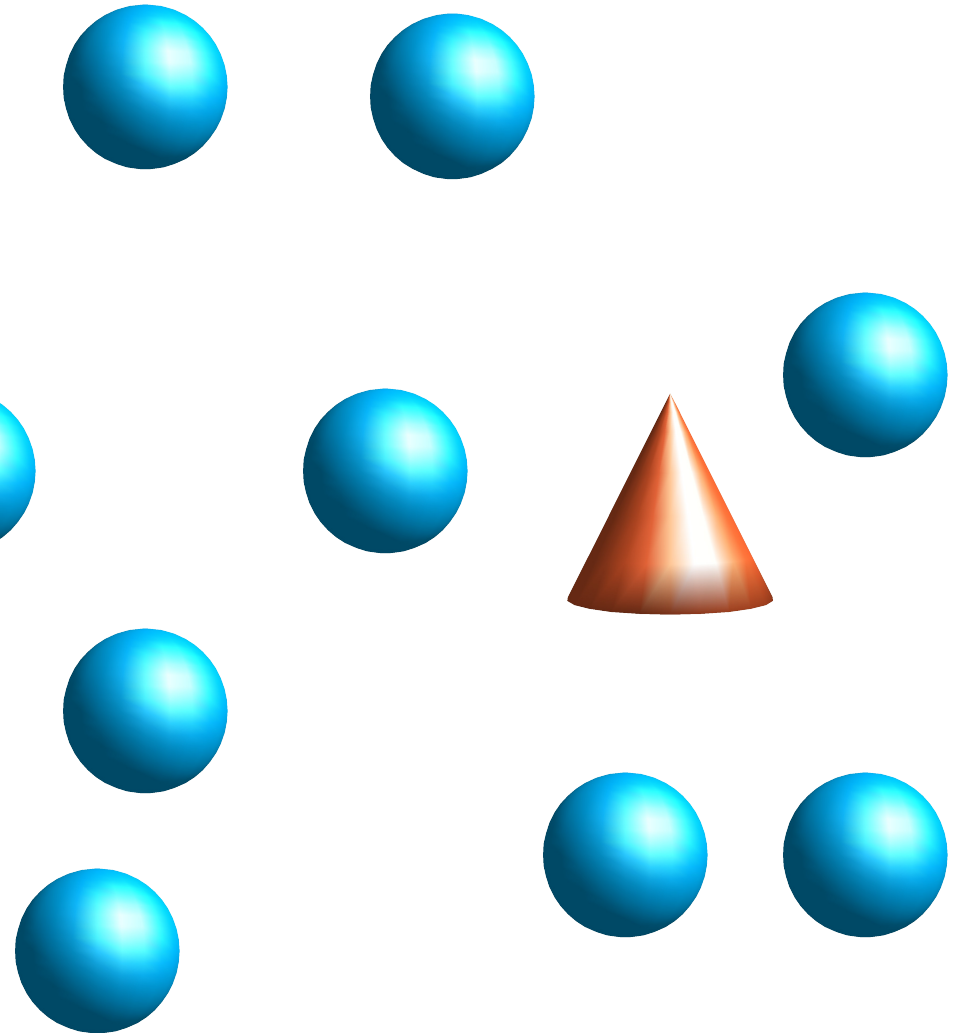


Taking Exception



Alex Kodat
Sirius Software Inc.



Sirius Software, Inc.

Stuff Happens

- Coding errors
- Environmental errors
 - Blivet situations
 - Hardware failures, including network errors
- User errors
- Unusual (unexpected?) data



Coding Errors

- The earlier caught, the better
 - Compile-time if possible
 - Request cancellation as soon as it's detected
- Recovery difficult
 - If code is incorrect, difficult to determine how incorrect it is
 - Can happen anywhere
 - All coding errors tautologically unanticipated
 - If the programmer anticipated it she wouldn't have code it
- So best response is usually terminate current request (compilation) and go to some error handler
 - APSY error proc, usually, under Model 204



Examples of Coding Errors

- Reference to undefined variable
 - Caught at compile-time
- Attempt to use null object reference
- Invalid array or arraylist item number reference
 - Non-positive or greater than number of items
- Assertion failures
- Many more



Environmental Errors

- Hardware failures
 - Application recovery can, at best, bypass the bad hardware
 - But, if critical, unrecoverable
- Resource shortages
 - Virtual storage full, CCATEMP full, record locking table full, etc.
 - Difficult to anticipate since can happen almost anywhere
 - Recovery likely to hit the same error
- So best response is to terminate the current request and **try** to run some error handler
 - If that fails, let environment provided error notification



Network Errors

- Special class of hardware error
 - Though could be caused by software errors in the “outside world”
- Relatively common “expected” error
 - Because of the complexity of the “outside” world
 - Systems can fail or be intentionally shut down
- Front-end error usually calls for request cancellation
 - If client gone, why bother continuing?
- Back end errors, harder to deal with
 - Probably want to notify user/client of server unavailability



User Errors

- The most common error
- Applications must be written to assume that everything the user enters might be incorrect
- Of course, only certain classes of user errors are detectable
 - Malformed values (like invalid dates, numbers, or CRNs)
 - Inconsistent data (like death date before birth date)
 - Crazy values (like birth dates in the future, hundreds of children)



Example of Detecting Bad User Entered Date

```
%startDate      is string len 32
...

%startDate = $web_form_parm('startdate')
if $sir_date2n(%startDate, 'YYYYMMDD') lt 0 then
  %errors = %errors + 1
  %error(%errors) = 'Invalid start date'
  %field(%errors) = 'startdate'
end if
```



Problem: User Error or Program Error

- Internal routines usually can't distinguish
 - For example, \$sir_date2n has no idea whether its date came from the database or was user entered
- Internal APIs might have the same issue
 - For example, a GetCrnData method might not know whether the CRN was entered by a user or came from a record in a file
 - If CRN not found when user entered, no surprise. If CRN not found when came from a file, probably indicative of an application/system error



How Should a Routine Indicate an Error?

- With a special return code
 - Like \$sir_date2n returns -1 to indicate an invalid date
 - Problem: no way to tell if caller actually checks for the return code
 - So special return code can end up causing incorrect application behavior
 - Problem: no (good) error return value available
 - Subroutines or methods that return strings
- Request cancellation
 - Like \$sir_date2n cancels request if the date format argument is invalid
 - Problem: no decent way to recover from this
 - So applications must pre-emptively check for possible errors (like with \$sir_datefmt). This can be quite difficult.



A New Way (for 204) of Indicating Errors - Exceptions

- Changes code path in caller of routine that experienced exception
 - Cancels request of exception not handled (caught)
 - Goes to exception handler (catcher) if exception handled
- So method return value (if any) not set
- Allows users of a routine to indicate that they expect an error
 - By providing *catch* code
- So code that expects possible errors provides a catch routine
- And code that doesn't doesn't so blows up on error



Example of Handling an Exception

```
%hexValue = $web_form_parm('hexValue')  
  
try  
    %string = %hexvalue:hexToString  
    call doSomething(%string)  
catch invalidHexdata  
    %errors:add('Invalid hexadecimal value entered')  
end try
```



What is an Exception

- A special kind of class/object
 - It can be *thrown* to indicate an error
 - When thrown, it affects a method caller's program flow
- Like any other object/class can contain variables and methods
 - For example, `InvalidHexData` has `Position` variable that indicates the position of the invalid hexadecimal character
- Supports inheritance (more on this later)
- The name of the class is what's specified on the catch statement
 - A target object can also be specified after the word `To`



Example of Catching an Exception Object

```
%invalidHex      is object invalidHexData

%hexValue = $web_form_parm('hexValue')

try
    %string = %hexvalue:hexToString
    call doSomething(%string)
catch invalidHexdata to %invalidHex
    %errors:add('Invalid hexadecimal value entered at position ' with -
                %invalidHex:position with ' in Hex Value')
end try
```



Exception Objects

- Treated just like any other object
 - So can be passed to/from methods
 - Can have methods applied to them
- Not required
 - Often all one cares about is whether an exception occurred
 - So class name is all that's required on Catch statement



Creating Your Own Exception Classes

- Declared just like any other other class
 - With the exception (hah, hah) that the *Exception* keyword is on the declaration
- Can extend other exception classes
 - But cannot extend non-exception classes
- Usually has constructor where all variables can be set
 - For reasons we'll see later



Examples of Exception Class Declarations

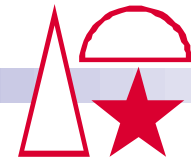
```
class badRgbCode exception extends invalidHexData inherit
  public
    variable alternative is string len 3
    constructor new(%position is float nameRequired, -
      %alternative is string len 3)
  end public
end class
```

```
class badCrn exception
  public
    variable reason is enumeration badCrnReason
    constructor new(%reason is enumeration badCrnReason nameRequired)
  end public
end class
```



Throwing an Exception

- Only thrown by methods
 - \$functions, non-OO subroutines and level-0 code need not apply
- System methods can throw system (class) exceptions
- User language methods can throw system or User Language exceptions
 - Each method must declare up-front what exceptions it might throw
 - Done with the Throws clause on method declaration/definition
 - Since exceptions are part of a method's interface to its callers



Example of A Throws Clause

```
class customer
  public shared
    function getCustomer(%crn is string len 9) is object customer -
      throws badCrn and invalidHexData
  end public shared
  function getCustomer(%crn is string len 9) is object customer -
    throws badCrn and invalidHexData
  ...
end function
end class
```



Throwing Exceptions

- Done with the Throw statement
- Always throws an instance of an exception class
- Must be a class declared in the method's Throws clause
 - Compile-time error if not declared
 - Though there's no requirement that a method throw an exception of every class in its Throws clause



Example of A Throw Statement

```
function getCustomer(%crn is string len 9) is object customer -  
    throws badCrn and invalidHexData
```

```
    %recset is object recordset in file crnfile  
    %badCrn is object badCrn
```

```
    find records to %recset  
        rectype = %crn  
    end find
```

```
    if %recset:count eq 0 then  
        %badCrn = new  
        %badCrn:reason = notFound  
        throw %badCrn  
    end if
```

```
    ...  
end function
```



But Usually You Don't Want to Hassle with Exception Class Variables

```
function getCustomer(%crn is string len 9) is object customer -  
    throws badCrn and invalidHexData
```

```
    %recset is object recordset in file crnfile
```

```
    find records to %recset  
        rectype = %crn  
    end find
```

```
    if %recset:count eq 0 then  
        throw %(badCrn):new(reason=notfound)  
    end if
```

```
    ...  
end function
```



Differences With Other Languages

Implementation of Exceptions - 1

- Catches not required for exceptions
 - Requiring this (as Java does) seems to defeat one of the purposes of exceptions of being used only if needed
- Exceptions can't be caught locally (inside method in which they were thrown)
 - Exceptions are an interface between methods and their callers
 - So no *Finally* clause
 - Only really makes sense for locally caught exceptions



Differences With Other Languages

Implementation of Exceptions - 2

- Exception classes don't all inherit from a single base class
 - Though you can pretend they do
 - But you can't generically catch all errors by catching the base exception class
 - A bad idea, anyway
- No automatic exception propagation



Automatic Exception Propagation

- Not provided by Janus SOAP ULI implementation
- Since usually a bad idea
 - Exposes innards of method to callers
 - Encourages sloppy programming
- Easy enough to provide “manually”
 - Called “throwing up”



Throwing Up

```
function getCustomer(%crn is string len 9) is object customer -  
    throws badCrn and invalidHexData  
  
    %invalidHex is object invalidHexData  
    ...  
  
    for each record in %recset  
        try %cust:favoriteColor = (color):hexToString  
        catch invalidHexdata to %invalidHex  
            throw %invalidHex  
        end try  
    ...  
end function
```



Inheritance

- Support for inheritance in Throws and Catch
 - Throws clause for a class allows throws of extension classes
 - But as the base class
 - Catch of base class will catch extension class exceptions
 - But avoid overdoing this
 - Janus SOAP ULI prevents this to a large degree by not providing **the** ultimate base exception class
- A moderately big and complex topic so details left for another day



OnThrow and OnUncaught Methods

- Have special semantics inside of exception classes
- OnUncaught called whenever exception thrown and not to be caught
 - OnUncaught cannot “save the day”
- OnThrow called whenever exception thrown and caught, or thrown and uncaught if no OnUncaught
- Must be subroutines
- Can be used to fix-up/validate exception object
 - Exception object is %this to these methods
- Can be used to standardize exception handling
- Must be subroutine with no parameters



Recommendations 1

- Don't throw exceptions for things that are clearly programming errors or unrecoverable environmental errors
 - Needlessly complicates a method's interface
 - Of course, there are border-line cases
 - So err on the side of providing exceptions
- Don't make Try blocks too big
 - You might accidentally catch an exception you didn't mean to
 - Unclear to person reading the code where the exception being caught might be thrown
 - Try to use single-line Try as much as possible



Recommendations

- Avoid exception genericity
 - Prevents code from accidentally handling one type of error when it gets another
- Always provide constructor that allows all/most variables in class to be set
- Make most/all variables ReadOnly
 - ReadOnly variables available in Sirius Mods 7.2



Summary

- New exception support provided powerful new capabilities for UL programmers
- Available in Sirius Mods 7.2
 - Now available in an Online near you
- Similar to other language's implementations of exceptions
 - But some slight differences that we feel make it better (naturally)



Let's Play Around with Exceptions a Bit



Sirius Software, Inc.