Article
Timothy Leavitt · Jan 15, 2020 9m read

Robust Error Handling and Cleanup in ObjectScript

Introduction and Motivation

A unit of ObjectScript code (a ClassMethod, say) may produce a variety of unexpected side effects by interacting with parts of the system outside of its own scope and not properly cleaning up. As a non-exhaustive list, these include:

- Transactions
- Locks
- I/O devices
- SQL cursors
- System flags and settings
- $Namespace
- Temporary files

Use of these important language features without proper cleanup and defensive coding can lead to an application that normally works correctly, but that may fail in unexpected and difficult-to-debug ways. It is critical for cleanup code to behave correctly in all possible error cases, especially since error cases are likely to be missed in surface-level testing. This post details several known pitfalls, and explains two patterns to accomplish robust error handling and cleanup.

Semi-related shameless plug: want to be sure that you're testing all of your edge cases? Check out my Test Coverage Tool on the Open Exchange*!*

Note: I originally published this content internally within InterSystems in June 2018. Posting it on the Developer Community has been on my to-do list for a year and a half now. You know what they say...

Pitfalls to Avoid

Transactions

A natural and simplistic approach to transactions is to wrap the transaction in a try/catch block, with a TRollback in the catch, as follows:

```
Try {
    TSTART
    // ... do stuff with data ...
    TCOMMIT
} Catch e {
    TROLLBACK
    // e.AsStatus(), e.Log(), etc.
}
```

In a vacuum, provided the code between TStart and TCommit throws exceptions rather than issuing Quit early when an error occurs, this code is perfectly valid. However, it is risky for two reasons:

- If another developer adds a "Quit" within the Try block, a transaction will be left open. Such a change would be easy to miss in code review, especially if it is not obvious that there are transactions involved in the current context.
- If the method with this block is called from within an outer transaction, the TRollback will roll back all transaction levels.

A better approach is to track the transaction level at the beginning of the method, and to rollback to that transaction level at the end. For example:

```
Set tInitTLevel = $TLevel
Try {
    TSTART
    // ... do stuff with data ...
    // The following is fine now; tStatus does not need to be thrown as an exception.
    If $$$ISERR(tStatus) {
        Quit
    }
    // ... do  more stuff with data ...
    TCOMMIT
} Catch e {
    // e.AsStatus(), e.Log(), etc.
}
While $TLevel > tInitTLevel {
    // Just roll back one transaction level at a time.
    TROLLBACK 1
}
```

Locks

Any code that uses incremental locks should also be sure to decrement the locks in cleanup code when they are no longer needed; otherwise, such locks will be held until the process terminates. Locks should not leak outside a method unless obtaining such a lock is a documented side effect of the method.

I/O Devices

Changes to the current I/O device (the $io special variable) similarly should not leak outside a method unless the purpose of the method is to change the current device (e.g., enabling I/O redirection). When working with files, use of the %Stream package is preferred over direct sequential file I/O using OPEN / USE / READ / CLOSE. In other cases, where I/O devices must be used, care should be taken to restore the original device at the end of the method. For example, the following code pattern is risky:

```
Method ReadFromDevice(pSomeOtherDevice As %String)
{
    Open pSomeOtherDevice:10
    Use pSomeOtherDevice
    Read x
    // ... do complicated things with X ...
    Close pSomeOtherDevice
}
```

If an exception is thrown before pSomeOtherDevice is closed, then $io will be left as pSomeOtherDevice; this is likely to cause cascading failures. Furthermore, when the device is closed, $io is reset to the process's default device, which may not be the same as the device prior to when the method was called.

SQL Cursors

When using cursor-based SQL, the cursor must be closed in the event of any error. Failure to close the cursor may result in resource leaks (according to the documentation). Also, in some cases if you run the code again and try to open the cursor, you'll get an "already open" error (SQLCODE -101).

System Flags and Settings

Rarely, application code may need to modify process- or system-level flags - for example, many of these are defined in %SYSTEM.Process and %SYSTEM.SQL. In all such cases, care should be taken to store the initial value and restore it at the end of the method.

$Namespace

Code that changes namespace should always New $Namespace at the beginning to ensure that namespace changes do not leak outside the scope of the method.

Temporary Files

Application code that creates temporary files, such as with %Library.File:TempFilename (which, on InterSystems IRIS in particular, actually creates the file) should take care to also remove the temporary files when they are no longer needed.

Recommended Pattern: Try-Catch (-Finally)

Many languages have a feature where a try/catch structure can also have a "finally" block that runs after the try/catch is complete, whether an exception has occurred or not. ObjectScript does not, but it can be approximated. A general pattern for this, demonstrating many of the possible problem cases, is:

```
ClassMethod MyRobustMethod(pFile As %String = "C:\foo\bar.txt") As %Status
{
    Set tSC = $$$OK
    Set tInitialTLevel = $TLevel
    Set tMyGlobalLocked = 0
    Set tDevice = $io
    Set tFileOpen = 0
    Set tCursorOpen = 0
    Set tOldSystemFlagValue = ""

    Try {
        // Lock a global, provided a lock can be obtained within 5 seconds.
        Lock +^MyGlobal(42):5
        If '$Test {
            $$$ThrowStatus($$$ERROR($$$GeneralError,"Couldn't lock ^MyGlobal(42)."))
        }
        Set tMyGlobalLocked = 1

        // Open a file
        Open pFile:"WNS":10
        If '$Test {
            $$$ThrowStatus($$$ERROR($$$GeneralError,"Couldn't open file "_pFile))
        }
        Set tFileOpen = 1

        // [ cursor MyCursor declared ]
```

```
        &;SQL(OPEN MyCursor)
        Set tCursorOpen = 1

        // Set a system flag for this process.
        Set tOldSystemFlagValue = $System.Process.SetZEOF(1)

        // Do the important things...
        Use tFile

        TSTART

        // [ ... lots of important and complicated code that changes data here ... ]

        // All done!

        TCOMMIT
    } Catch e {
        Set tSC = e.AsStatus()
    }

    // Finally {

    // Cleanup: system flag
    If (tOldSystemFlagValue '= "") {
        Do $System.Process.SetZEOF(tOldSystemFlagValue)
    }

    // Cleanup: device
    If tFileOpen {
        Close pFile
        // If pFile is the current device, the CLOSE command switches $io back to the
 process's default device,
        // which might not be the same as the value of $io was when the method was ca
lled.
        // To be extra sure:
        Use tDevice
    }

    // Cleanup: locks
    If tMyGlobalLocked {
        Lock -^MyGlobal(42)
    }

    // Cleanup: transactions
    // Roll back one level at a time up to our starting transaction level.
    While $TLevel > tInitialTLevel {
        TROLLBACK 1
    }

    // } // end "finally"
    Quit tSC
}
```

Note: in this approach, it is critical that "Quit" and not "Return" is used in the Try ... block; "Return" will bypass the cleanup.

Recommended Pattern: Registered Objects and Destructors

Sometimes, the cleanup code can get complicated. In such cases, it may make sense to facilitate reuse of the cleanup code by encapsulating it in a registered object. System state is tracked when the object is initialized or when methods of the object that change state are called, and reverted to its original value when the object goes out of scope. Consider the following simple example, which manages transactions, the current namespace, and the state of $System.Process.SetZEOF:

```
/// When an instance of this class goes out of scope, the namespace, transaction leve
l, and value of $System.Process.SetZEOF() that were present when it was created are r
estored.
Class DC.Demo.ScopeManager Extends %RegisteredObject
{

Property InitialNamespace As %String [ InitialExpression = {$Namespace} ];

Property InitialTransactionLevel As %String [ InitialExpression = {$TLevel} ];


Property ZEOFSetting As %Boolean [ InitialExpression = {$System.Process.SetZEOF()} ];


Method SetZEOF(pValue As %Boolean)
{
    Set ..ZEOFSetting = $System.Process.SetZEOF(.pValue)
}

Method %OnClose() As %Status [ Private, ServerOnly = 1 ]
{
    Set tSC = $$$OK

    Try {
        Set $Namespace = ..InitialNamespace
    } Catch e {
        Set tSC = $$$ADDSC(tSC,e.AsStatus())
    }

    Try {
        Do $System.Process.SetZEOF(..ZEOFSetting)
    } Catch e {
        Set tSC = $$$ADDSC(tSC,e.AsStatus())
    }

    Try {
        While $TLevel > ..InitialTransactionLevel {
            TROLLBACK 1
        }
    } Catch e {
        Set tSC = $$$ADDSC(tSC,e.AsStatus())
    }

    Quit tSC
}

}
```

The following class demonstrates how the above registered class could be used to simplify cleanup at the end of the method:

```
Class DC.Demo.Driver
{

ClassMethod Run()
{
    For tArgument = "good","bad" {
        Do ..LogState(tArgument,"before")
        Do ..DemoRobustMethod(tArgument)
        Do ..LogState(tArgument,"after")
    }
}

ClassMethod LogState(pArgument As %String, pWhen As %String)
{
    Write !,pWhen," calling DemoRobustMethod("_$$$QUOTE(pArgument)_"):"
    Write !,$c(9),"$Namespace=",$Namespace
    Write !,$c(9),"$TLevel=",$TLevel
    Write !,$c(9),"$System.Process.SetZEOF()=",$System.Process.SetZEOF()
}

ClassMethod DemoRobustMethod(pArgument As %String)
{
    Set tScopeManager = ##class(DC.Demo.ScopeManager).%New()

    Set $Namespace = "%SYS"
    TSTART
    Do tScopeManager.SetZEOF(1)
    If (pArgument = "bad") {
        // Normally, this would be a big problem. In this case, because of tScopeMana
ger, it isn't.
        Quit
    }
    TCOMMIT
}

}
```

#Best Practices #Error Handling #ObjectScript #Caché #InterSystems IRIS

Source URL:https://community.intersystems.com/post/robust-error-handling-and-cleanup-objectscript