Try-Catch FAQ

Version 2019.4
2020-01-28

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Try-Catch FAQ
InterSystems IRIS Data Platform Version 2019.4 2020-01-28

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

Try-Catch FAQ

General Questions

What is Try-Catch?

Try-Catch is a language construct in ObjectScript that allows applications to handle exceptional conditions, called *exceptions*. **Try** defines a block of code for which exceptions are handled by a paired **Catch** block. Exceptions include all ObjectScript system errors such as <DIVIDE> and <UNDEFINED>, which are thrown implicitly when the language encounters an error; they also encapsulate other types of exceptional conditions, which can be thrown explicitly by the application with the **Throw** command. If an exception is thrown in the **Try** block, control is transferred to the **Catch** block, and execution resumes there.

Exceptions can be thrown from code that is not in a defined **Try** block. When that happens, the next exception handler on the stack catches the exception, unwinding the stack as necessary. The exception handler that catches the exception may be a **Catch**, but it may alternatively be a **$ZTRAP** handler (more on this below).

An exception, when thrown, causes the application to deviate from the normal flow of control and resume execution at the first available exception handler on the stack (the deepest stack level), unwinding the stack if necessary until one is found. When using **Try-Catch**, typically the first available exception handler would be a **Catch** block. The exception object is available to the **Catch** block, and can be inspected to recover information about the exception.

What is the difference between an exception and an error?

The term "error" can have multiple meanings, so this article avoids using it as a technical term. An exception is an object that is a subclass of the %Exception.AbstractException class. Several types of exceptions are modeled as subclasses of %Exception.AbstractException.

ObjectScript system errors, such as <DIVIDE> and <UNDEFINED> are exceptions of the class %Exception.SystemException. Exceptions of this form are automatically instantiated and thrown by the system when such errors occur. There are other classes of exceptions that can be instantiated by the application and thrown using the **Throw** command. %Exception.StatusException is an exception class to model %Status errors, and %Exception.SQL models SQLCODE errors. You can also create your own exception class by extending these exception classes.

I called a method that returned an error %Status value. How do I throw it as an exception?

%Exception.StatusException has a method, CreateFromStatus, to create an exception object that can then be thrown with the **Throw** command. For example, if the variable *sc* contains a %Status value, the following code will throw it as an exception:

```
if $$$ISERR(sc) {
  throw ##class(%Exception.StatusException).CreateFromStatus(sc)
}
```

Note: This functionality is also accessible from the macros **$$$ThrowStatus** and **$$$THROWONERROR**.

How do I throw an exception from an error SQLCODE?

The %Exception.SQL class has a method, **CreateFromSQLCODE**, to create an exception object that can then be thrown with the **Throw** command. For example, if the variable *SQLCODE* contains an SQLCODE value and *%msg* its message, the following code throws it as an exception:

```
if SQLCODE<0 {
  throw ##class(%Exception.SQL).CreateFromSQLCODE(SQLCODE,%msg)
}
```

What happens if an exception occurs inside my Catch block?

Exceptions that are thrown inside a **Catch** block are just like exceptions that occur anywhere else outside of the **Try** block – the next available exception handler on the stack handles them. You can nest another **Try-Catch** within the **Catch** block itself in order to catch additional exceptions within your exception handling code.

Can I convert from an exception to a %Status or SQLCODE

Yes, exception objects have methods **AsStatus** and **AsSQLCODE** that do just that.

What can I do with an exception when I catch it?

The **Catch** block is a fully functioning ObjectScript environment and you can use any commands you need. There are some things that you may typically want to do in order to process the exception, which are described here. These actions need not be entirely contained within the **Catch** block; they can be done in code following the **Catch** block if desired.

First, because **Catch** handles multiple kinds of exceptions, your application may want to distinguish among different exceptions in order to determine what to do. You can use the $classname function or the %IsA method (inherited from the InterSystems IRIS® %Library.Base class) to determine the class or superclass of the exception object. You can inspect the Name and Code properties of the exception object to determine the type of error.

You often want to undo work that has been done prior to the exception, release a lock or other resource, and/or roll back a transaction.

You may want to log it to the standard application error log by calling **LOG^%ETN**. If the exception is not a %Exception.SystemException, set **$ZERROR** to a meaningful value prior to calling **LOG^%ETN**; this value will be used as the Error Message field in the log entry. (The application error log is visible in the Management Portal's **Application Error Log** page.) Additionally, you can get a summary of the exception to display to the user using the **DisplayString** method of the exception object.

Upon completion of all the above you would typically do one of several things:

- Continue processing or return from the current procedure
- Re-throw the exception to the next exception handler on the stack
- Throw a new exception
- Halt the process

Here's an example that illustrates some of these concepts:

```
func(id) public {
  Try {
    ; Flag indicates if we locked the global
    Set locked=0
    ; If we cannot get the lock, throw a user-created
    ; exception with the information we need
    Lock +^mygbl(id):0 If '$test {
      Throw ##class(Exception.MyException).%New("Unable to
```

```
          lock",$name(^mygbl(id)))
    }
    Set locked=1
    Set sc=$system.OBJ.Compile("MyClass")
    If $$$ISERR(sc) {
      Throw ##class(%Exception.StatusException).CreateFromStatus(sc)
    }
    ; Some further processing which may throw exceptions
    ; ...

    If locked { Lock -^mygbl(id) }
  }

Catch exception {
    ; Release the lock resource before doing anything else
    If locked { Lock -^mygbl(id) }
    ; First determine what sort of exception this is
    If exception.%IsA("%Exception.SystemException") {
      ; Log error in error log
      Do BACK^%ETN
      ; Throw my exception class rather than the system exception
      Throw ##class(Exception.MyException).CreateFromSystemException(exception)
    } ElseIf $classname(exception)="Exception.MyException" {
      ; Ignore this sort of exception and just return to code
      ; after the catch block
    } Else {
      ; We will just throw these to outer error handler
      Throw exception
    }
  }
}
```

I use Try-Catch in an outer-level procedure that will call other procedures, which in turn call other procedures. At some deep stack level, an exception occurs that gets caught in my outer-level Catch. How do I recover the call stack where the exception occurred?

For exceptions of the class %Exception.SystemException (such as the <UNDEFINED> ObjectScript system error), you can use the $stack function to inspect the error stack. For other exception classes, the code that throws the exception needs to be modified to allow the exception handler to recover the stack.

The following example shows how to use the $stack function for system exceptions and one way to capture the stack for other classes of exception. It comes in two parts: a custom exception class to extend %Exception.StatusException with stack information, and an example routine that both logs and displays the captured information, for both system exceptions and other types of exceptions.

The exception class:

```
Class MyException.Status Extends %Exception.StatusException
{

  Property Stack [ MultiDimensional ];

  /// Convert a %Status into an exception
  ClassMethod CreateFromStatus(pSC As %Status)
      As %Exception.AbstractException
  {
    // You could choose to override %OnNew and put this code that
    // captures the stack there instead of here in CreateFromStatus.
    // We put it here because we only need to capture the stack in
    // the outer exception, and it is more simply insulated from
    // future changes in the superclasses.

    // First, call CreateFromStatus in the superclass to instantiate
    // the object and fill in the standard exception information.
    set exc=##super(pSC)

    // Clear $ecode so that $stack() refers to the current stack,
    // not the error stack.
    set $ecode=""

    // Subtract one level because we don't need
    // to see this method itself in the stack.
    set exc.Stack=$stack-1

    for i=1:1:exc.Stack {
      set exc.Stack(i)=$stack(i)_
```

```
          " "_$stack(i,"PLACE")_" "_$stack(i,"MCODE")
    }
    quit exc
  }
}
```

The example routine that both logs and displays the exception information:

```
#include %occInclude
testexc(throwsystemexception) {
  try {
    do sub1($g(throwsystemexception))
  } catch exc {
    if exc.%IsA("%Exception.SystemException") {
      set stack=$stack(-1)
      // For System Exceptions, get the stack from the
      // built-in error stack using $stack().
      for i=1:1:stack {
        set stack(i)=$stack(i)_
            " "_$stack(i,"PLACE")_" "_$stack(i,"MCODE")
      }
    } else {
      if $extract($classname(exc),1,12)="MyException." {
        // Exceptions from package MyException will carry the
        // stack of the exception in the multidimensional
        // Stack property.
        merge stack=exc.Stack
      }
      // Set $ze explicitly because it's needed by BACK^%ETN
      // and only SystemExceptions set it implicitly.
      set $ze=exc.DisplayString()
    }
    do BACK^%ETN
    write !,"Exception occurred: ",exc.DisplayString()
    write !,"  class: ",$classname(exc)
    write !,"  name: ",exc.Name
    write !,"  code: ",exc.Code
    if $data(stack) {
      write !,"  stack:"
      for i=1:1:stack {
        write !,"    ",stack(i)
      }
    }
    write !
  }
}
sub1(throwsystemexception) {
  if throwsystemexception {
    do systemexception
  } else {
    do myexception
  }
}
myexception() PUBLIC {
  set sc=$$$ERROR($$$GeneralError,"this is my status code")
  throw ##class(MyException.Status).CreateFromStatus(sc)
}
systemexception() PUBLIC {
  // get a <DIVIDE> error
  set x=1\0
}
```

The output from the test routine:

```
USER>do ^testexc(1)

Exception occurred: <DIVIDE> 18 systemexception+2^testexc
  class: %Exception.SystemException
  name: <DIVIDE>
  code: 18
  stack:
    DO +3^testexc +1     do sub1($g(throwsystemexception))
    DO +40^testexc +1     do systemexception
    DO systemexception+2^testexc +1   set x=1\0

USER>do ^testexc(0)

Exception occurred: ERROR #5001: this is my status code
  class: MyException.Status
  name: 5001
  code: 5001
  stack:
    DO +3^testexc +1     do sub1($g(throwsystemexception))
```

```
   DO +42^testexc +1     do myexception
   DO myexception+2^testexc +1   throw ##class(MyException.Status).CreateFromStatus(sc)

USER>do ^%ERN


For Date: T  16 Feb 2012   2 Errors

Error: ?L

 1. <DIVIDE>systemexception+2^testexc  at  1:25 pm.   $I=/dev/ttys001   ($X=0  $Y=299)
    $J=8225  $ZA=0   $ZB=$c(13)   $ZS=16384 ($S=16504448)
          set x=1\0

 2. ERROR #5001: this is my status code  at  1:25 pm.   $I=/dev/ttys001   ($X=0  $Y=310)
    $J=8225  $ZA=0   $ZB=$c(13)   $ZS=16384 ($S=16504336)
```

Try-Catch and Older Error-Handling Mechanisms

InterSystems IRIS supports other mechanisms for handling exceptions, such as $ZTRAP. Which should I use?

Use **Try-Catch**. It's the recommended exception handling mechanism in InterSystems IRIS for several reasons:

1. In most cases, **Try-Catch** allows you to create more readable and elegant code, which makes it easier to maintain your application.
2. It has no runtime performance cost for activities that succeed (that is, where there is no exception). This generally leads to a performance benefit.
3. Because it's easier to use, **Try-Catch** code is less prone to error. (For example, it helps avoid the construction with **$ZTRAP** that can create an infinite loop.)
4. For existing applications, it can provide a path to a consistent exception handling interface by encapsulating code with other InterSystems mechanisms to handle exceptions.
5. **Try-Catch** gives you access to the exception object and therefore allows you to recover all information about the exception that was thrown, regardless of what type of exception occurs.

How do Try-Catch and exceptions interact with the older ObjectScript error handlers?

If an exception is thrown, and an older error handler is the first available exception handler on the stack, control is passed to that error handler in the normal way. The exception object, however, will not be available. If the exception thrown was a system exception (%Exception.SystemException), the **$ZERROR** value will be set as expected; for other exception classes caught by **$ZTRAP**, the **$ZERROR** value will be set to <NOCATCH>. In either case, the flow of control is the same.

If code in a **Try** block calls a procedure, method, or subroutine that sets **$ZTRAP** and then an exception occurs inside that procedure, the **$ZTRAP** catches the exception because it's at a deeper stack level. If code is using **$ZTRAP** and calls a procedure, method, or subroutine that uses **Try-Catch** and an exception occurs within the **Try**, then the **Catch** catches it (again, because it's at the deeper stack level). In short, exception handling uses whatever is at the deepest stack level.