Protecting Against Bad Parameter Counts

By David Adams

Technical Note 07-14

Overview

Calling a custom method with an incorrect parameter list may lead to error displays, crashes, or miscalculations in data. Fortunately, all of these problems are easily avoided with some simple defensive code. This technical note describes the range of behaviors 4th Dimension exhibits when methods do not protect against bad parameter counts, and then provides a generalized method for detecting and managing bad parameter lists.

Example Method and 4th Dimension Behavior

Before discussing code to solve the problems associated with bad parameter lists, it is worth exploring the full range of possible behaviors 4th Dimension may exhibit when missing parameters are addressed. For this discussion, consider the method listed below called *DisplaySum*:

```
C_LONGINT($1;$firstValue_l)
C_LONGINT($2;$secondValue_l)

$firstValue_l:=$1
$secondValue_l:=$2

C_LONGINT($sum_l)
$sum_l:=$firstValue_l+$secondValue_l
ALERT(String($sum_l))
```

The code adds two longints passed as parameters and displays the result. The example is deliberately trivial for clarity. Below is a sample of a call made to *DisplaySum* with the correct number of parameters:

```
DisplaySum (1;2)
```

The *DisplaySum* method adds the values in $1 and $2 and displays their total, as pictured below:

What happens when not enough parameters are passed? For example, what does 4th Dimension do when only one parameter is passed, as in the example below:

```
DisplaySum (1)  ` Only one parameter is passed while two are required.
```

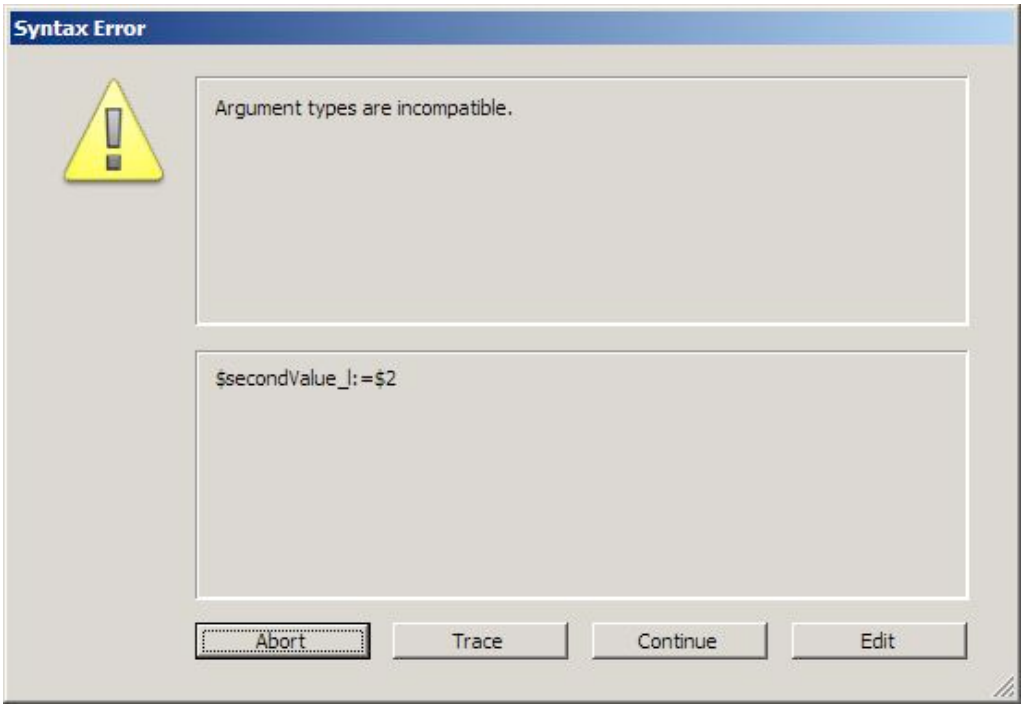
In such a situation, the following line of code within *DisplaySum* is invalid:

```
$secondValue_l:=$2
```

What happens at this point in the execution of *DisplaySum* depends on the operating system, version of 4th Dimension, and if and how the code is compiled. We will now review how 4th Dimension reacts to the problem code in various situations.

Bad Method Calls: Interpreted

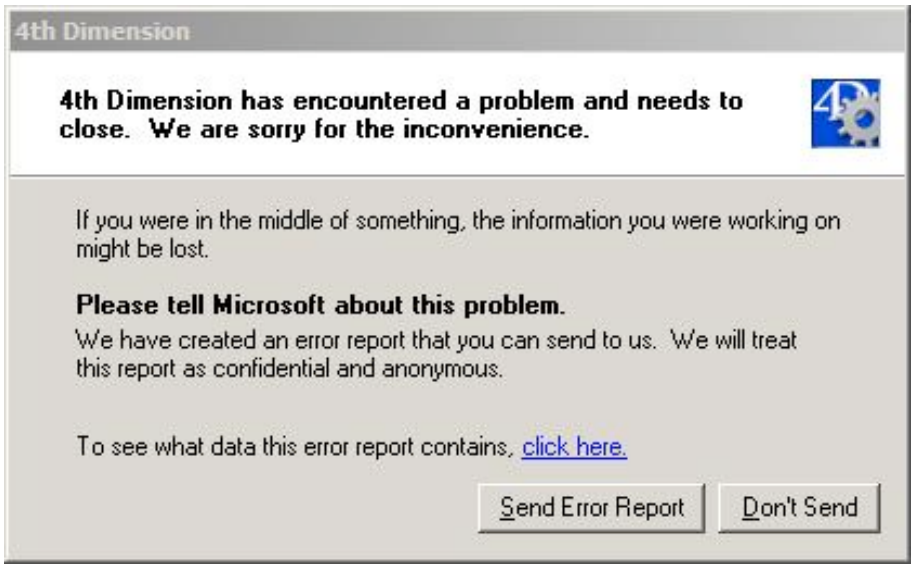
When run in an interpreted environment, the problem code very consistently leads to a syntax error dialog, like the one pictured below:



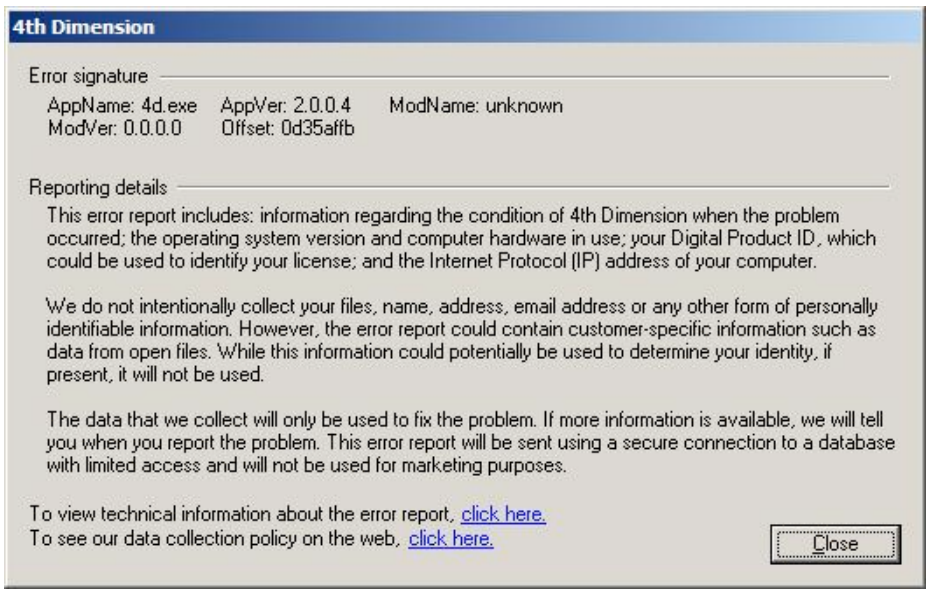
The error shown above is exactly what should be hoped for as no value was passed for $2. Ideally, test code and test procedures exercise all of the relevant code in a system to flush out errors of this sort. However, even with good testing procedures in place, it is relatively easy to write method calls with bad parameter lists.

Bad Method Calls: Compiled without Range Checking on Windows

The consequences of a bad parameter list can be more severe in a compiled database than in an interpreted one. While the exact behavior in a compiled program is variable, it is never desirable. For example, below is the outcome from running the problem code in a database without range checking on Windows XP:



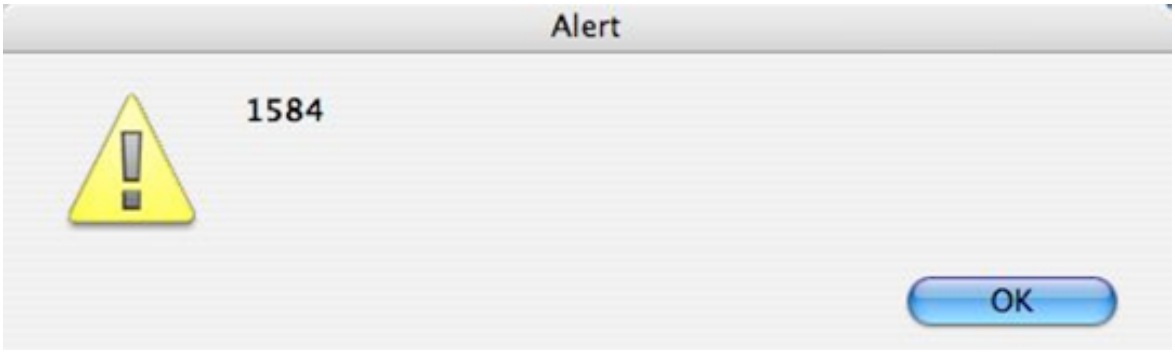
The dialog above should give anyone pause. Given that *one* bad parameter assignment in a program with hundreds of thousands of lines of code can crash a system, it is worth spending some effort to defend against such problems. This is particularly important as crash screens, like the one above, provide little or no information about the source of the problem. There is no way to know if there is a bug in 4th Dimension, a plug-in, or custom code. For example, the image below shows the technical information offered when selecting the `click here` link:



Unfortunately there is nothing in the error screen above, or in the more detailed report it links to, that identifies what the problem is.

Bad Method Calls: Compiled without Range Checking on OS X

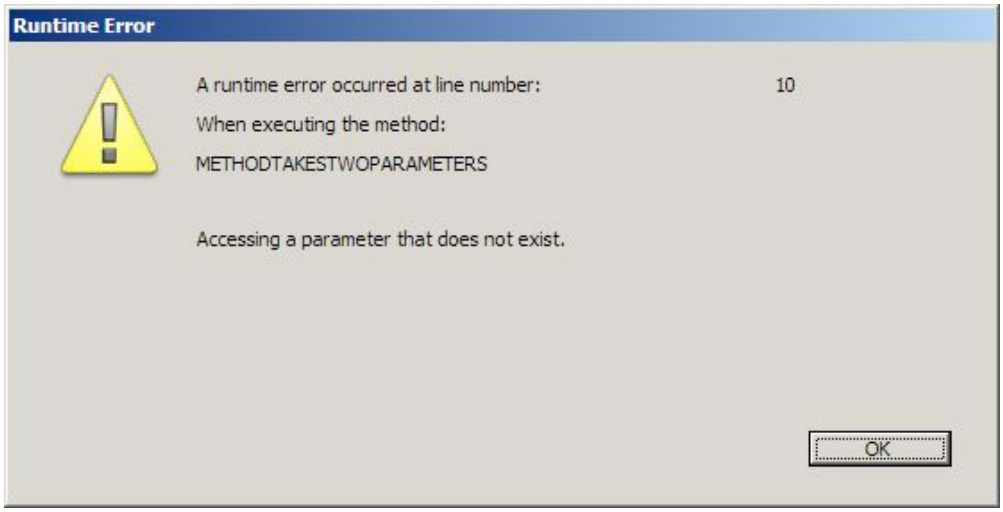
As disturbing as a crash may be, at least it is clear the system has a problem. The results of running the same code in a database compiled without range checking on OS X are arguably more insidious:

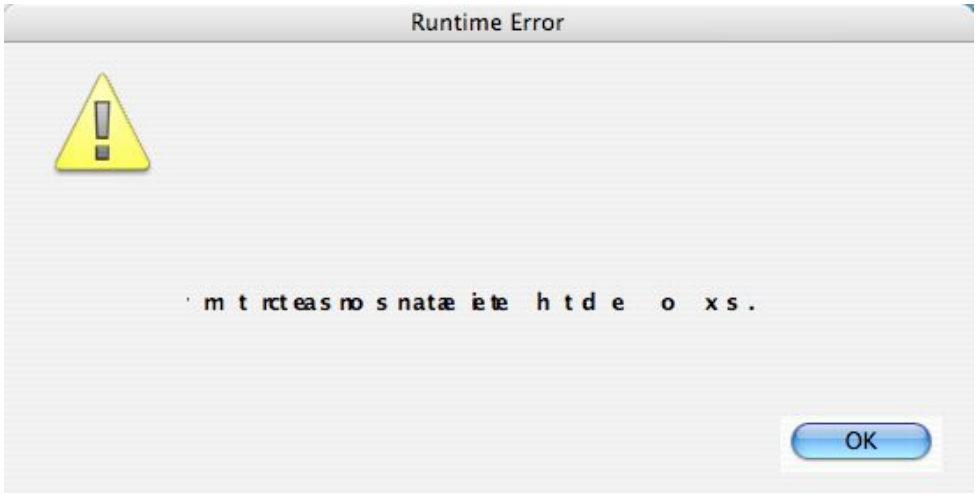


In this trivial example, it does not matter that the alert shows that 1 = 1584 (which is only true for very large values of 1), but what if the routine returns results used in calculations? Data can be subtly distorted without detection. In a way, a hard crash is more desirable as the problem is more likely to get immediate attention.

Bad Method Calls: Compiled with Range Checking

With range checking enabled, 4th Dimension displays, or attempts to display, a dialog similar to the syntax error dialog seen in interpreted mode:





Alerts, such as the ones above, are more helpful than crashes or bad results. Even the impossible to read OS X dialog makes it obvious something is wrong with the program. Such alerts improve the chances of a bug being discovered before harm is done to the data.

Tip *Compile with range checking turned on. The runtime speed cost of range checking is too small to be worth measuring or worrying about outside of rare situations.*

About the Behavior Documented Above

The behavior described above showing how 4th Dimension reacts to a bad parameter list in various situations should not be depended upon. Future versions of 4th Dimension may behave differently than what has been described here. Regardless, there is close to no desirable behavior available. At most, 4th Dimension should fail over gracefully when passed a bad parameter list. Regardless, a bad parameter list is a developer bug and should be detected and fixed.

Detecting and Stopping Bad Parameter Lists

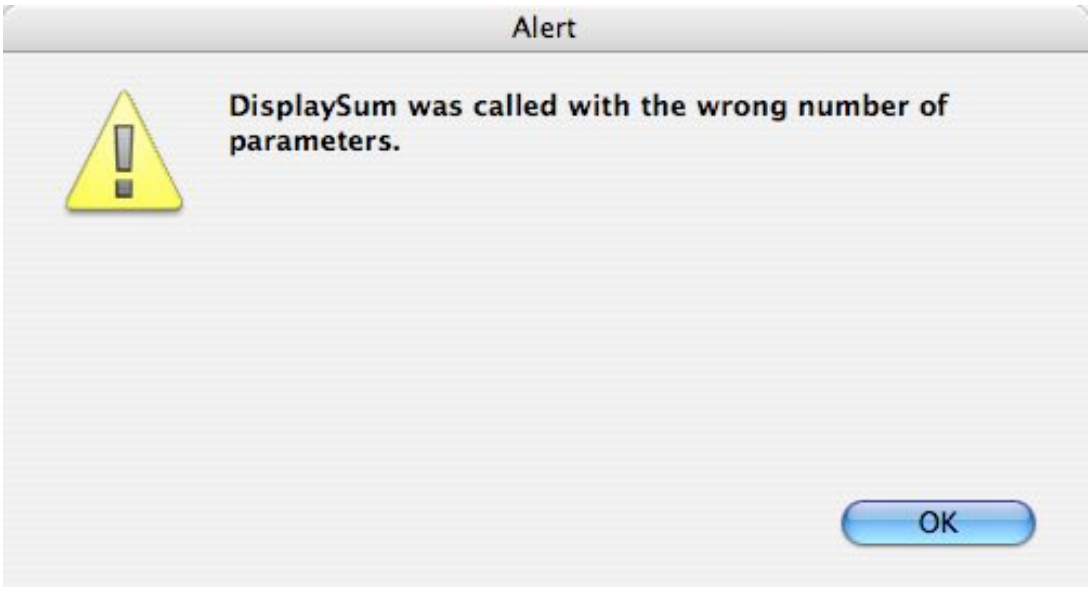
Counting Parameters within Each Method

Avoiding problems from reading or assigning missing parameters is as simple as counting the parameters before using them, as illustrated in the code below:

```
If (Count parameters=2)
  $firstValue_l:=$1
  $secondValue_l:=$2

  C_LONGINT($sum_l)
  $sum_l:=$firstValue_l+$secondValue_l
  ALERT(String($sum_l))
Else
  ALERT("DisplaySum  was called with the wrong number of parameters.")
End if
```

When called with anything other than two parameters, the routine now shows an alert like the one pictured below:



A simple **If** test and an **ALERT** are superior to any of the earlier approaches. Now there is no possibility of a crash or bad results, and the exact location of the problem is indicated on screen.

Tip *It is not necessary to use the* **ALERT** *command when encountering an error. This command is undesirable when, for example, code is called in triggers, over the Web, or through SOAP. Instead errors can be logged, displayed, or returned, as appropriate for the current context.*

Using a Centralized Parameter Tester

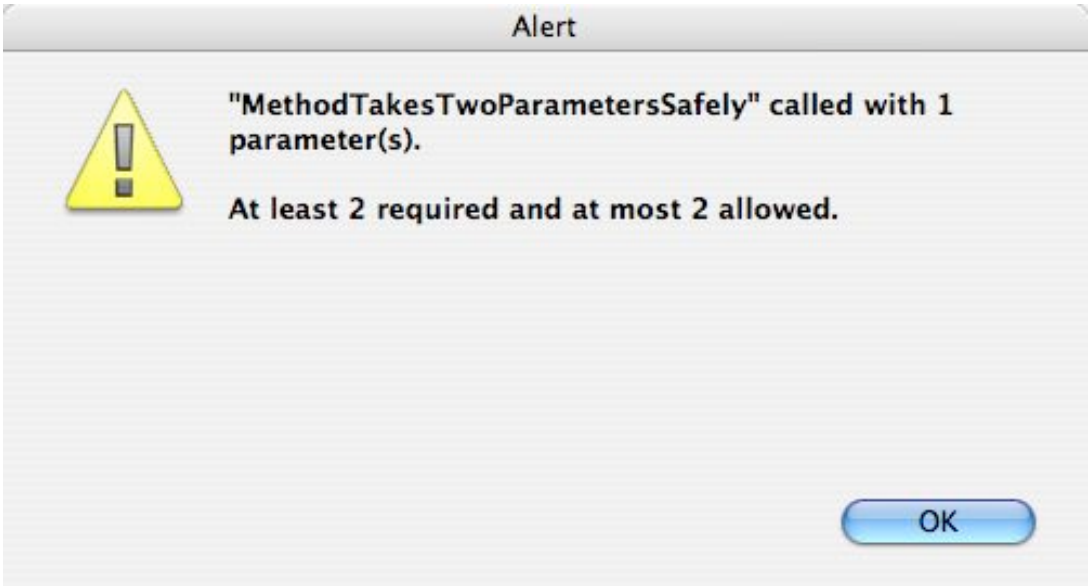
While the code shown above solves the problems associated with addressing missing parameters, it embeds the error management logic directly in the method. For many developers, it is more convenient and flexible to keep the error management behavior slightly separate from working code. For example, consider the revised method below:

```
C_LONGINT($1;$firstValue_l)
C_LONGINT($2;$secondValue_l)

If (ParameterCountIsOkay (Current method name;2;2;Count parameters))
  $firstValue_l:=$1
  $secondValue_l:=$2

  C_LONGINT($sum_l)
  $sum_l:=$firstValue_l+$secondValue_l
  ALERT(String($sum_l))
End if
```

Instead of counting the parameters directly, a function named *ParameterCountIsOkay* is called. The subroutine takes the calling method's name, the minimum number of parameters expected, the maximum number of parameters expected, and the actual number of parameters received. The *ParameterCountIsOkay* routine tests that the parameter count is within range and, if not, displays an error like the one pictured below:



The code for the *ParameterCountIsOkay* routine is listed below:

```
C_BOOLEAN($0;$countIsOkay_b)
C_TEXT($1;$methodName_text)
C_LONGINT($2;$min)
C_LONGINT($3;$max)
C_LONGINT($4;$count)

$countIsOkay_b:=False` Default.

If (Count parameters>=4)` Test that this routine has enough parameters!
   $methodName_text:=$1
   $min:=$2
   $max:=$3
   $count:=$4
```

```
   If (($count<$min) | ($count>$max))` Not enough parameters | Too many parameters
     $countIsOkay_b:=False` This is a good place to log or display an error.

     C_TEXT($error_text)
     $error_text:=""
     $error_text:=$error_text+Char(Double quote)+$methodName_text+Char(Double quote)
     $error_text:=$error_text+" called with "+String($count)+" parameter(s). "
     $error_text:=$error_text+Char(Carriage return)+Char(Carriage return)
     $error_text:=$error_text+"At least "+String($min)+
     $error_text:=$error_text+" required and at most "+String($max)+" allowed."
     ALERT($error_text)

   Else
     $countIsOkay_b:=True
   End if

Else ` The ParameterCountIsOkay routine didn't get enough parameters.
   ` This is a good place to log or display an error.
   C_TEXT($error_text)
   $error_text:=""
   $error_text:=$error_text+Char(Double quote)+Current method name+Char(Double quote)
   $error_text:=$error_text+" called with "+String($count)+" parameter(s). "
   $error_text:=$error_text+Char(Carriage return )
   $error_text:=$error_text+"Four parameters are required"
   ALERT($error_text)

End if `(Count parameters>=4)

$0:=$countIsOkay_b
```

There are several advantages to moving the logic into a function like *ParameterCountIsOkay*, including:

- Coding for individual methods is as easy as adding an **If** test before using parameters. Individual routines do not need to handle parameter count errors themselves. Therefore, each individual method is slightly simpler and smaller than it would otherwise have to be.
- Testing, debugging, and maintaining the code is easier.
- Centralizing the behavior simplifies adding new features, such as adding errors to a log, emailing errors, or suppressing error dialogs when within a trigger or Web/SOAP process.

Regarding ON ERR CALL

The **ON ERR CALL** command provides a way of inserting a custom method into the error management process. Ideally, when an error is encountered, the custom error handler method installed with **ON ERR CALL** runs, and the program does not crash, show a range check dialog, or syntax error dialog. Unfortunately, **ON ERR CALL**'s exact limits are not documented and the feature does not trap all errors. For example, **ON ERR CALL** does not meaningfully improve management of bad parameter list errors in compiled code.

Summary

Reading missing parameters in custom methods may cause error dialogs, program crashes, or distorted results. All of these problems can be avoided with simple defensive coding. This technical note explores some of the various behaviors of overly trusting code with bad parameter lists and then shows how to prevent problems 100% of the time.