Computer Programming Solutions
TECHNICAL INFORMATION SHEET

Handling Exceptions - Delphi, CPPB, IBO and Firebird/InterBase

Abstract: *It's a common misconception that Murphy's Law was invented by programmers. In fact, it goes back much further than programmer culture. It was born at the US Air Force's Edwards Base in 1949. It was named after Capt. Edward A. Murphy, an engineer working on Air Force Project MX981, [a project] designed to see how much sudden deceleration a person can stand in a crash.*

One day, after finding that a transducer was wired wrongly, he cursed the technician responsible and said, "If there is any way to do it wrong, he'll find it."

The contractor's project manager kept a list of "laws" and added this one, which he called Murphy's Law. Shortly afterwards, the Air Force doctor, Dr John Stapp, who rode a sled on the deceleration track to a stop, pulling 40 Gs, gave a press conference. He said that their good safety record on the project was due to a firm belief in Murphy's Law and in the necessity to try and circumvent it.

Exception Handling - Beating Murphy!

Exception handling in Delphi is about circumventing Murphy's Law. Delphi provides excellent support for trapping both anticipated and unexpected errors and providing ways to handle them elegantly without crashing your program or leaving your users wondering what to do next.

Many of the following notes came from the Delphi 5 VCL help text. The topic of exception handling is thoroughly documented there, although its organisation is rather confusing. Charlie Calvert and Marco Cantù both do it much better in their respective books "Kylix Developer's Guide" (SAMS, 2001) and "Mastering Delphi 3" (Sybex, 1997). I'm sure they wouldn't mind being quoted here at times.

Delphi Exception Objects

Delphi raises an exception (by creating an Exception object) whenever an error or other event interrupts normal execution of a program. The exception transfers control to an exception handler, thus separating normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code.

An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application **uses** the SysUtils unit, all runtime errors are automatically converted into exceptions. Errors that would otherwise terminate an application—such as insufficient memory, division by zero, and general protection faults—can be caught and handled.

C++ Builder Exception Objects

C++ also raises an exception by creating an Exception object whenever an error or other event interrupts normal execution of a program.

Note *:: in C++ there are two different exception types, VCL exceptions which are based on the Delphi Exception class and C++ exceptions. In the current context, we consider just the VCL exceptions.*

The Exception Constructs in your Code

An "exception block" consists of a block of protected code, in which you need to have errors detected, followed by a response block in which you handle the error. Place any block of code that can potentially cause one or more errors in a *protected block* bounded by the keywords **try**...**except**. The *response* code follows immediately after it and finishes with an **end** statement. After that point the flow of execution resumes at the next statement.

This simple example is from the Delphi help:

```
...

try                          { begin the protected block }
  Font.Name := 'Courier';    { if any exception occurs... }
  Font.Size := 24;           { ...in any of these statements... }
  Color := clBlue;
except                       { ...execution jumps to here }
  on Exception do MessageBeep(0); { this handles any exception by beeping }
end;
...                                 { execution resumes here }
```

This all very well, but it doesn't do much. Sure, it traps up to three errors but the user has no way to know why there was a beep. It won't fix the problem that caused the error, either.

In CPPB this translates to the following :

```
...

try {                         // begin the protected block
  Font->Name := "Courier";    // if any exception occurs...
  Font->Size := 24;           // ...in any of these statements...
  Color = clBlue;
} catch(...) {                // ...execution jumps to here
  MessageBeep(0);             // this handles any exception by beeping
}
...                           // execution resumes here
```

This code differs from the Delphi code above in that the **except** keyword is replaced by **catch (**...**)** which catches every exception. In CPPB you have to specify the exception type in the **catch** and not in an '**on … do …'**.

As with the Delphi code, this fragment will simply beep when any exception is raised by the code in the **try** block.

Here is a more meaningful example (from Marco Cantù):

```
function Divide (A, B: Integer): Integer;
begin
  try
  { the following statement is protected because
    it can generate an error if B is 0 }
    Result := A div B;
  except
    on EDivByZero do
      Result := 0;
```

```
  end;
end;
```

In this case, EDivByZero is a class defined in SysUtils to catch division-by-zero errors. When the error occurs, Delphi creates an Exception object of this class. Marco's code catches the error and fixes it silently by ensuring that the function will return 0 instead of reporting the error.

Both of these examples serve to show that the **on** SomeException **do**.. construct provides a place in your code where you can do whatever it takes to resolve the exception. The first example uses the generalized Exception object - "something went wrong but we can't tell what it was" - and delivers a Beep to the user's ear when any Exception occurs.

In the second case, the code detects a specific type of error, for which an exception type is specifically declared, in this case, in SysUtils, but it might just as well be an exception type which your code declares and, when it is detected, you have custom code somewhere to handle it.

An exception block can have as many of these **on** SomeException **do**.. blocks inside the except block as you need. The program flow will work through them one by one, much like a **case** statement, so you can use the try.. block to identify a number of exception conditions and then handle them all in the ...**except** block.

Marco's example, when translated into C++ looks like this :

```
int Divide(int A, int B)
{
   try {
        return (A / B);
   }  catch (EDivByZero &Oops) {
      return 0;
   }
}
```

This time, you will notice that the catch block is specifically looking for an EdivByZero error.

Custom Exception classes in Delphi

It's easy! All that is needed is to *first* declare an Exception class and afterwards **raise** the exception anywhere in your code where the error condition is detected.

In this example, a function will return a string which is the result of shortening a supplied string to a supplied length. The process could throw an error if the string is already shorter than the length requested. The function tests the length of the string and, if the error would occur by processing the supplied arguments, it will **raise** a custom exception instead of trying to process them.

```
type
  TMyException = class (Exception);

function ShortenString (S: string; Len: Integer):  String;
begin
  if Length(S) > Len then
    Result := Copy (S,1,Len)
  else
    raise TMyException.Create( 'String is too small to shorten.');
end;
```

Now, the custom exception will be raised if the length of the the input string is equal to or

shorter than the Len argument:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S: string;
begin
  S := Edit1.Text;
  try
    Edit1.Text := ShortenString(S, 5);
  except
    on E: TMyException do
      Edit1.Text := E.Message;
  end;
end;
```

The line

```
    on E: TMyException do
```

creates an exception object named E of the class TMyException. The object's Message property, which was assigned in its constructor, can be accessed in the handler:

```
      Edit1.Text := E.Message;
```

So, the nuts and bolts are:

1. Declare the custom exception class
2. Write code in your procedure to raise an instance of that exception in case an anticipated specific error occurs
3. Provide code to handle that specific exception when it actually occurs

Custom Exception classes in CPPB

It's also easy! All that is needed is to *first* declare a new **typedef** based on the existing Exception class and afterwards **throw** the exception anywhere in your code where the error condition is detected.

Using the same example above, we now have the following code :

```
typedef Exception TMyException ;

AnsiString ShortenString (AnsiString S, int Len)
{
  if (S.Length() > Len)
    return S.SubString(1, Len);
  else
    throw  TMyException("String is too small to shorten.");
}
```

The above should be typed just after the constructor for the form. In the event handler for a button, we put the following :

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    AnsiString S;

    S = Edit1->Text;
    try {
```

```
        Edit1->Text = ShortenString(S, 5);
    } catch (TMyException &E) {
        Edit1->Text = E.Message;
    }
}
```

The exception is created within the function ShortenString at the line :

```
   throw  TMyException("String is too small to shorten.");
```

This is trapped by the **catch** statement and it is processed accordingly. The Exception's message, assigned when it was created, can be accesed within the **catch** block :

```
      Edit1.Text := E.Message;
```

So, the nuts and bolts are:

1. Declare the custom exception class with a simple **typedef**
2. Write code in your procedure to raise an instance of that exception in case an anticipated specific error occurs
3. Provide code to handle that specific exception when it actually occurs

Protecting Allocated Resources

Under normal circumstances, you can ensure that an application frees allocated resources by including code to free resources that you allocate in your methods. Because exception handling passes control out of the protected block at the statement where the error occurs, you need to ensure that the application will execute the resource-freeing code, regardless of whether the code runs with or without errors.

Some common resources that you should always be sure to release are disk files, memory, Windows resources and any objects that your methods create within their own scope.

For example, the following event handler creates a stringlist, then generates an error. Control jumps out to the response block and so the code to free the stringlist never executes.

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  MyStringlist: TStringlist;
  ii: integer;
begin
  MyStringlist := TStringList.Create;
  with MyStringlist do
    try
      for ii := 1 to 20 do
        Add(EditText1.Text[ii]);
       with IB_Query1.FieldByName('BLOBCOL') as TIB_ColumnBlob do
         Assign(MyStringList);
       Free;
     except
        on E: Exception do
          E.Message := 'Your text must be at least 20 characters long.')
     end;
end;
```

Although most errors are not that obvious, the example illustrates how, when the Index Out of Bounds error occurs, execution jumps out of the protected block, so the Free method never gets called for the stringlist.

The solution is to nest the **try**…**except**…**end** block inside a **try**…**finally** resource-protection block. The equivalent of a response block in this structure is a final block of code which will be executed no matter what else has occurred inside the **try**…**finally** block.

So, to fix up our example:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  MyStringlist: TStringlist;
  ii: integer;
begin
  MyStringlist := TStringList.Create;
  with MyStringlist do
    try // start the try..finally block
      try  // start the try..except block
        for ii := 1 to 20 do
          Add(EditText1.Text[ii]);
         with IB_Query1.FieldByName('BLOBCOL') as TIB_ColumnBlob do
           Assign(MyStringList);
       // Free;  this is moved out of the protected block
       except
         on E: Exception do
           E.Message := 'Your text must be at least 20 characters long.')
       end;
    finally
      Free;  // here it will be executed no matter what
    end;
end;
```

Here is the same thing in C++ Builder. First, the 'broken' piece of code:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TStringList *MyStringList;
    int ii;

    MyStringList = new TStringList;
    try {
       // BEWARE : AnsiStrings start at 1 in C++ as well as in Delphi !!!
       for (ii = 1; ii < 20; ++ii) {
          MyStringList->Add(Edit1->Text[ii]);
       }

       // In C++ We don't have to cast to a TIB_ColumnBlob :o)
       IB_Query1->FieldByName("BLOBCOL")->Assign(MyStringList);
       delete MyStringList;
    } catch(...) {
       throw Exception("Your text must be at least 20 characters long.");
    }
}
```

As with the Delphi code above, when the Index Out of Bounds error occurs, execution jumps out of the protected block into the **catch** block so the call to **delete** never gets called for the stringlist and a memory leak is the end result.

The solution is to nest the **try**…**catch**… block inside a **try**…**__finally** (note the double underscore)resource-protection block. The equivalent of a response block in this structure is a final block of code which will be executed no matter what else has occurred inside the

try...**__finally** block.

__finally is a Borland extension to the C++ language. It has been added for compatibility with Delphi's **finally** keyword, and works in a similar manner. Code inside a **__finally** block is always executed regardless of whether an exception happened, or not, within the **try** block.

So, to fix up our example:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TStringList *MyStringList;
    int ii;

    MyStringList = new TStringList;
    try {
       try {
           // BEWARE : AnsiStrings start at 1 in C++ as well as in Delphi !!!
           for (ii = 1; ii < 20; ++ii) {
              MyStringList->Add(Edit1->Text[ii]);
           }

           IB_Query1->FieldByName("BLOBCOL")->Assign(MyStringList);
       } catch(...) {
         throw Exception("Your text must be at least 20 characters long.");
       }
    } __finally {
       delete MyStringList;
    }
}
```

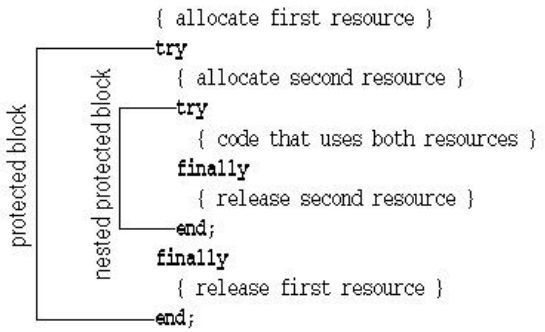
NOTE :: Because Delphi and C++ Builder always free Exception objects internally, this is one resource you never have to worry about.

Nesting exception responses

(digested from Delphi help)

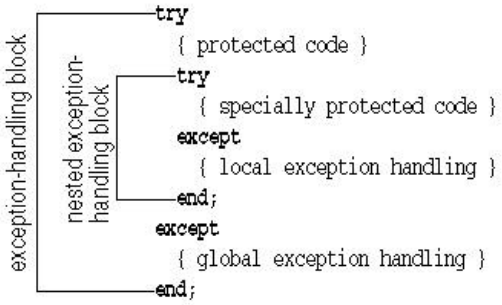
Because Pascal allows you to nest blocks inside other blocks, you can customize responses even within blocks that already customize responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:



[Picture: Delphi help]

You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:



[Picture: Delphi help]

You can also mix different kinds of exception-response blocks, nesting resource protections inside exception handling blocks and vice versa.

The same processes can be used in C++ Builder to nest protection blocks, simply replace Delphi's **finally** with **__finally** in the above skeleton layouts.

Flow of Control

As we have already seen, you create an exception object by calling the constructor of the exception class inside a **raise** statement in Delphi or a **throw** statement in C++.
For example,

```
raise EMathError.Create;
```

In C++:

```
throw EMathError("A Maths error occurred");
```

In C++ the class EMathError's constructor does not allow an empty parameter list, so there must be a message supplied. If not, all you get is a blank message box on the screen – which isn't very helpful.

When an exception is referenced in Delphi in a **raise** statement, or in C++ in a **throw** statement, it is governed by special exception-handling logic. A **raise** or **throw** statement never returns control in the normal way. Instead, it transfers control to the *innermost* exception handler that can handle exceptions of the given class. (The innermost handler is the one whose **try**…**except** block (C++ **try**…**catch** block) was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an ERangeError exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);  // StrToInt is declared in SysUtils
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

Notice the CreateFmt method called in the raise statement. Exception and its descendants have a choice of constructors that provide alternative ways to create exception messages. (See *The Constructors of the Exception Class*.)

Converting the above example to C++:

```
int StrToIntRange(const AnsiString S, int Min, int Max)
{
   int Result = StrToInt(S);
   if ((Result < Min) || (Result > Max)) {
      throw ERangeError("%d is not within the valid range of %d to %d",
                        ARRAYOFCONST((Result, Min, Max))
                       );
   }
   return Result;
}
```

Notice the ERangeError exception class has a built in constructor which accepts a format string and a list of values to insert into it. The usage of this constructor is similar to the way in which the Format() function is called. In C++ constructors need not have different names as the compiler knows which one to use by checking the list of passed parameters.

Both Delphi and C++ destroy a raised (thrown) exception automatically after it is handled. Never attempt to destroy a raised exception manually. Even if a call to Exit, Break, or Continue causes control to leave the exception handler, the exception object is still automatically destroyed.

Re-raising exceptions in Delphi

Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

The raise statement has a general form:

```
raise object at address
```

where *object* and ***at*** *address* are both optional.

So far, we have looked at examples using the syntax

```
raise object
```

e.g.

```
raise Exception.Create('I am an Exception');
```

If object is omitted, and the reserved word **raise** occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way (e.g. by sending a simple message) and then *re-raise* the exception.

In this example from the Delphi help, the `GetFileList` function allocates a TStringList object and fills it with file names matching a specified search path:

```
function GetFileList(const Path: string): TStringList;
```

```
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
    begin
      Result.Add(SearchRec.Name);
      I := FindNext(SearchRec);
    end;
  except
    Result.Free;
    raise;
  end;
end;
```

`GetFileList` creates a TStringList object, then uses the FindFirst and FindNext functions (defined in SysUtils) to initialize it. If the initialization fails—for example because the search path is invalid, or because there is not enough memory to fill in the string list—GetFileList needs to dispose of the new string list, since the caller does not yet know of its existence.

For this reason, initialization of the string list is performed in a **try**...**except** statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

NOTES

1. Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the SysUtils unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units—including SysUtils—are finalized and the exception is re-raised. Then the System unit catches the exception and handles it, usually by interrupting the program.

2. When an address is specified a **raise** statement, it is usually a pointer to a procedure or function; you can use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

Re-throwing exceptions in C++

Re-throwing is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

So far, we have looked at examples using the syntax

```
throw exception_object
```

e.g.

```
throw Exception("I am an Exception");
```

If object is omitted, and the reserved word **throw** occurs in an exception block without an object reference following it, it re-throws whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way (e.g. by sending a simple message) and then *re-throw* the exception.

In the following example, the StrToIntRange example above, has been used again. This time there are two **try**...**catch**... blocks, an inner and an outer. The ErangeError exception thrown by the StrToIntRange function is first trapped in the inner handler, processed and re-thrown whereupon, it is trapped in the outer exception handler.

The code for StrToIntRange has not changed and is not shown in the example below.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    try {
       try {
          Edit1->Text = IntToStr(StrToIntRange("7", 0, 5));
       } catch (ERangeError &E) {
          ShowMessage("Inner : " + E.Message);
          throw;
       }
    } catch (Exception &E) {
       ShowMessage("Outer : " + E.Message);
    }
}
```

Note that we need to catch (Exception &E) in the outer handler, rather than just catch(...) to get at the Message property of whatever exception we are catching.

Exception types

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from an exception class, ultimately from the Exception class defined in SysUtils.

The SysUtils unit declares several standard routines for handling exceptions, including ExceptObject, ExceptAddr, and ShowException. SysUtils and other VCL units also include dozens of exception classes, all of which (except OutlineError) derive from Exception.

The Exception class has a choice of constructors providing properties like Message and HelpContext that can be used to pass an error description and a context ID for context-sensitive help.

The Constructors of the Exception Class

In Delphi, the constructor is, by convention, usually called **Create.** However, a class can have more than one constructor, each with an arbitrary identifier.

In C++ this is not the case. A constructor's name is always the same as the class it is constructing. This means that the Exception class's constructor is also called Exception.

The Delphi Exception class has eight constructors, described below. In C++, to choose the one you want and, regardless of which one you choose, simply **throw Exception(parameters)** and the correct one will be used. The compiler will sort out which constructor is required.

Do not attempt to use the Delphi names, **CreateFmt**, for example – it will not work.

In Delphi, the eight possible constructors that you can use with **raise**, according to how you want to present the exception to the user, are as follows:

- The most commonly used is **Create** which has just one argument, Msg, a string consisting of the message which will be displayed to the user when an Exception is **raise**d.

```
raise TMyException.Create('You have just done something illegal');
```

- **CreateFmt** is similar to Create but it provides for special formatting of the error message. It has two arguments: a string containing the template for the message and an array of constant values containing the values to be inserted into the message template.

```
raise TMyException.CreateFmt('Exception %s:  supplied crop length %d is
longer than the supplied string, s%.', [EStringTooShort, Len, Edit1.text]);
```

  The format specifiers for CreateFmt's template are the same as those available to Delphi's Format function.

- **CreateRes** enables you to load a string resource for your error message. For Delphi 3 and above, it is an overloaded function, allowing you to reference either a string table resource or the address of a resource string.

  The string table version has one argument, an unsigned integer which is the identifier of the string resource in the table:

```
raise TMyException.CreateRes(144);
```

  The **resourcestring** version has one argument, a pointer to the resourcestring:

```
...
resourcestring
  rsBadDogMsg = 'Oh, Fido, you are such a bad dog!';
...

raise TMyException.CreateRes(@rsBadDogMsg);
```

- **CreateResFmt** is like CreateRes and CreateFmt combined. Your STRINGTABLE or resourcestring entry would be templates containing format identifiers.

  The STRINGTABLE version has two arguments, the string resource number and an array of fill-in values for the format identifiers in the string. For example, for a string stored as

```
...
145, "s%, you are such a bad s%"
...

raise TMyException.CreateResFmt(145, [sPetName, sPetType]);
```

  The resourcestring version similarly has two arguments, the first pointing to the address of the resource string, the second an array of fill-in values:

```
resourcestring
...
rsBadPetMsg = 'Oh, s%, you are such a bad s%!';
...

raise TMyException.CreateResFmt(@rsBadPetMsg, ['Wally', 'wombat']);
```

- **CreateHelp** is like Create but provides you with a way to use context-sensitive help to handle exceptions. It accepts a second argument, a HelpContext ID named AHelpContext. Your handler can refer this argument to a MessageDlg() procedure's HelpCtx argument, e.g.

```
...
on E:MyException.CreateHelp('Something is wrong here.  Press Help for more
info.', 12345)
do begin
...
MessageDlg(E:Msg, mtError, [mbOK, mbHelp], E:AHelpContext);
...
  end;
```

- **CreateFmtHelp** is similar but allows formatting like CreateFmt. See the Delphi Help.

- **Create ResHelp** is similar to CreateHelp but takes its message string from a resource, as CreateRes does. See the Delphi Help.

- **Create ResFmtHelp** is like CreateReshelp but handles formatted resource strings as CreateResFmt does. See the Delphi Help.

Logic Flow in Delphi Exception Handling

An exception handler has the form

```
on identifier: type do statement
```

where *identifier:* is the optional identifier of the Exception object (e.g. E:). If included, it can be used subsequently in the handling code to refer to properties or methods of the Exception object.

type is a declared Exception type.

statement is any statement or block of statements, including conditional constructs such as **if**..**then**..**else** or **case** blocks.

You can have as many handlers as you need in the exception handling block, e.g.

```
except
  on E: EMyIB_ISCError do
  begin
    ...
    ...
  end;
  on E: EJoesIB_ISCError do
  begin
    ...
    ...
  end;
  ....
end;
```

If an exception is raised during execution of the block of statements between **try** and **except**, either in one of the statements list or by a procedure or function called one of the statements, an attempt is made to handle the exception using this precedence:

1. If any of the handlers in the exception block matches the exception, control passes to the

first such handler. A match is made if the exception type in the handler matches the class of the exception or an ancestor of that class. For this reason, place the descendants before the ancestors in the "pecking order".

2. If no such match is found, control passes to the statement in the **else** clause, if there is one.

3. If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

4. If none of the preceding conditions above is satisfied, the search continues in the exception block of the next-most-recently entered try...except statement that has not yet exited.

5. If no appropriate handler, **else** clause, or statement list is found there, the search propagates to the next-most-recently entered **try**...**except** statement, and so forth.

6. If the outermost **try**...**except** statement is reached and the exception is still not handled, the program terminates. (The IB Objects TIB_Session class has a catch-all handler named HandleException, which guarantees that every exception will be handled without terminating the program).

When an exception is handled, the stack is traced back to the procedure or function containing the **try**...**except** statement where the handling occurs, and control is transferred to the executed exception handler, **else** clause, or statement list.

This process discards all procedure and function calls that occur following the statement where the exception is actually handled. The exception object is then automatically destroyed through a call to its Destroy destructor and control is passed to the statement following the **try**...**except** statement.

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. EMathError appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try

  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can declare an identifier before the name of the exception class, to represent the exception object during execution of the statement that follows **on**...**do**. The scope of the identifier is limited to that statement. For example,

```
try

  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an **else** clause, the **else** clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
 ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** clause handles any exception that is not an EMathError.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try

 ...
except
  HandleException;
end;
```

Here, the HandleException routine handles any exception that occurs as a result of executing the statements between try and except.

Logic Flow in C++ Exception Handling

An exception handler has the form

```
catch (type &identifier) { statement }
```

where *identifier:* is the optional identifier of the Exception object (e.g. E:). If included, it can be used subsequently in the handling code to refer to properties or methods of the Exception object.

type is a declared Exception type.

statement is any statement or block of statements. The braces - {} - are mandatory.

You can have as many handlers as you need in the exception handling block, e.g.

```
catch(EmyIB_ISCError &E) {
    ...
    ...
}

catch(EJoesIB_ISCError &E) {    ...
    ...
    ...
}
```

If an exception is thrown during execution of the block of statements between **try** and the first **catch**, either in one of the statements list or by a procedure or function called one of the statements, an attempt is made to handle the exception using this precedence:

1. If any of the handlers in the catch blocks match the exception, control passes to the first

such handler. A match is made if the exception type in the handler matches the class of the exception or an ancestor of that class. For this reason, place the descendants before the ancestors in the "pecking order" – place **catch(Exception &E)** or **catch(...)** as the final entry in the list of handlers.

2. If no such explicit match is found, control passes to the statement in the **catch(Exception &E)** or **catch(...)** clause, if there is one.

3. If none of the preceding conditions above is satisfied, the search continues in the exception block of the next-most-recently entered try...except statement that has not yet exited.

4. If no explicit handler, **catch(Exception &E)**, or **catch(...)** is found there, the search propagates to the next-most-recently entered **try**...**catch** statement, and so forth.

5. If the outermost **try**...**catch** statement is reached and the exception is still not handled, the program terminates. (The IB Objects TIB_Session class has a catch-all handler named HandleException, which guarantees that every exception will be handled without terminating the program).

When an exception is handled, the stack is traced back to the procedure or function containing the **try**...**catch** statement where the handling occurs, and control is transferred to the executed exception handler.

This process discards all procedure and function calls that occur following the statement where the exception is actually handled. The exception object is then automatically destroyed through a call to its destructor and control is passed to the statement following the **try**...**catch** statement as in this example :

```
try {

 ...
} catch(Exception &E) {
       // Process exceptions here ...
       ...
}

ShowMessage("I have control again !");
```

Once the exception handler has finished processing, control returns to the **ShowMessage** statement and the program will continue from that point.

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. EMathError appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try {

 ...
} catch(EZeroDivide &) { HandleZeroDivide();
} catch(EOverflow &)   { HandleOverflow();
} catch(EMathError &)  { HandleMathError();
}
```

You will note that we are not actually interested in the Exception object itself in the above handlers, merely catching each type and handling it accordingly. You always need the ampersand (&) because VCL exceptions – and this is what we are discussing here – must always be caught by reference.

If an exception handler declares an identifier after the name of the exception class, to represent the exception object during execution of the exception handler, the scope of the identifier is limited to that statement. For example,

```
catch(Exception &E) {
       // E has scope only between the '{' above and the '}' below.
       ErrorDialog(E.Message, E.HelpContext);
}
// E no longer exists at this point
```

If the exception block specifies a **catch (Exception &E)** or **catch(...)** clause, this handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try {

 ...
} catch(EZeroDivide &E) { HandleZeroDivide(E);
} catch(EOverflow &E)   { HandleOverflow(E);
} catch(EMathError &E)  { HandleMathError(E);
} catch (Exception &E) { HandleALLOthers(E));
}
```

Here, the **catch (Exception &E)** clause handles any exception that is not an EMathError.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try {

 ...
} catch (...)
  HandleException;
}
```

Here, the HandleException routine handles any exception that occurs as a result of executing the statements in the protected try block.

Exceptions raised in a ..finally.. clause

If an exception is raised but not handled in the finally clause, that exception is propagated out of the **try**...**finally** statement, and any exception already raised in the **try** clause is lost. The **finally** clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

Responses to exceptions

Once an exception is raised, your application can execute cleanup code, handle the exception, or both. We've already discussed how the **try**...**finally**...**end** construct is applied to your exception code to ensure that resources get freed, even if the process encounters an exception that abends the intended sequence of events.

By handling an exception, you make a a specific response to a specific kind of exception. Once the exception is handled, the error condition is cleared and the program can continue executing.

Typically, you will have your exception handler allow the application to recover from errors and continue running in a way that makes sense to the user and ensures that steps in a task are not duplicated or omitted because the exception occurred.

What kinds of exceptions can occur?

Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as “File not found,” are easy to correct and retry, while others, such as running out of memory, might be more difficult for the application or the user to correct.

Apart from these, In a database application you have a whole range of things that can cause exceptions.

- A lock conflict when the user goes to commit an update or acquire a pessimistic lock is an situation that needs to be handled. What will you tell the user in the exception message? What will you do with the transaction? Roll it back? Place it into a retry loop? These are some of the options available to you. You might consider defining a custom class or even a hierarchy of EIB_ISCErrors which you can raise to meet each condition. You could provide a generic handler procedure to which you pass an error object of the parent class and then handle each individual class conditionally in your procedure.

- An attempt to post an update to a record that doesn't exist will return a baffling message to the user and leave her not knowing what to do next, if you don't catch the exception, tell her what is happening and handle it so that she can fix the problem and try again.

- Key violations of various kinds. You can handle these explicitly. For example, in the rare event that a user has edit access to a unique column, you can design a more user-friendly message to replace "ISC ERROR: VIOLATION OF UNIQUE CONSTRAINT". Something along the lines of "Customer code XY12345 is already in use for another customer" would be more informative for the user. If you need multi-lingual error messages, you can make good use of resource strings (Delphi3 and later) or string tables.

- Numeric and string overflows - for these you can test the columns that caused these errors and provide useful re-interpretations to your users.

- and many more..

iberror.h and IB_Header.pas

The error codes applicable to all the Firebird and InterBase engines, along with definitions of all the corresponding API constants, can be found in the .\include directory of your server installation, in a source file named **iberror.h**. New error codes get added from version to version, so you should look at one whose version matches that of your server program.

IBO translates these definitions into Pascal in the **IB_Header** unit. I mention it because some of these constant names are quite meaningful. It is one more piece of information to have to hand when you are planning your custom error handlers and it makes your code so much more readable if you detect the error codes using constants instead of those confusing numbers.

The IB_Constants unit

As its name indicates, this unit defines most of the constants used throughout IBO. Included there are resourcestring message definitions for scores of error conditions. (Community members maintain international versions of this unit as well - they can be found in the main source tree).

You will learn a lot about the information and error conditions for which IBO provides standard message responses because the identifiers of these strings have meaningful names, e.g.

```
E_NO_TRANS_STARTED  = 'Transaction has not been started';
```

When you need a message string for an error handler, look in IB_Constants first and see whether there is one already defined that fits what you want to do - especially if you are developing for multiple languages. If someone has already translated IB_Constants.pas into your language, it is very likely that you will find exactly the message you want in this substitute unit.

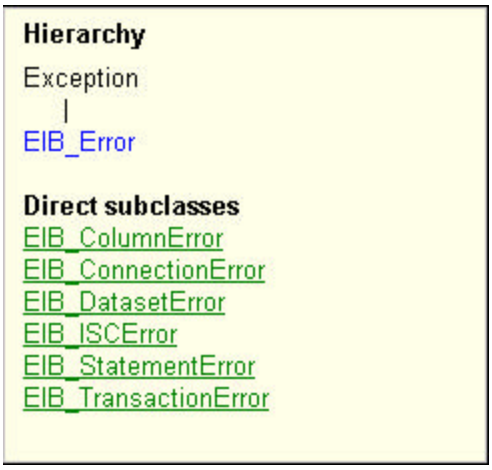
Although you can add your own special message resources to IB_Constants, you will have to update this unit each time you take a new IBO release or sub-release. (This is usually necessary, anyway, since only the English-language version of IB_Constants.pas is guaranteed to be fully up-to-date).

You might prefer to make your own ErrorHandler unit, defining message strings for the specific error conditions in your application, as well as generic error handlers which can be called from **on** SomeException **do**.. handler blocks, with the specific exception class passed in their argument lists. Some developers maintain a very broad-ranging unit for this purpose and re-use it in multiple projects.

Custom Error Classes

You can add any properties you need when you inherit your own custom error classes from Exception. You can group exceptions into families using inheritance.

IB Objects has its Exception descendant class, EIB_Error. As you can see from this graphic, it provides the base class for the family of exception classes that present information from errors that can occur in the use of the IBO components and, in the case of EIB_ISCError, in structures returned to IBO by the InterBase/Firebird API:



With such a hierarchy, a single EIB_Error exception handler will also handle the descendants and any special properties they have.

EIB_ISCError

The special descendant, type EIB_ISCError, declared in IB_Session.pas, introduces public properties that are buffers to provide access to all the items that are returned to the API in the error status vector. Its TDataset-compatible cousin is **TIBO_ISCError**:

- ERRCODE - the ISC error code of the last error, e.g. 335544580, a long integer
- ErrorCodes - stringlist delivering error codes of all errors that occurred in the sequence
- ErrorMessage - stringlist delivering the error messages for all corresponding items in ErrorCodes
- Sender - the component in which the exception was raised
- SQL - the SQL statement which caused the exception, if applicable
- SQLCODE - the SQL error code for the exception, if applicable
- SQLMessage - the text interpretation of the SQLCODE, if applicable. Messages from programmed EXCEPTION events thrown from stored procedures can be read from this field.

It introduces these public methods:

- CreateISC - constructor, sets up the data structures for the above properties
- Destroy - destructor, frees those data structures
- ShowDialog - displays a modal dialog box containing the details contained in the data structures

Handling all exceptions in one place

All components descended from TIB_Component have an OnError event. In addition, the dataset components have special error events for the Edit, Insert and Delete methods, which you will probably want to utilize at component level. TIBOStoredProc has its own OnError event.

However, the TIB_SessionProps component, which can be dropped into any IBO application, provides an ideal place to centralize all of your handlers for EIB_ISCErrors or EIBO_ISCErrors. As arguments, it supplies the Sender object, along with all of the items returned in the EIB_ISCError properties, making it easy to write generic handlers where otherwise you would need to repeat code in many individual handlers.

Silent Exceptions in Delphi code

Also included in the arguments of OnError is a Boolean **var,** RaiseException, which allows control over the raising of a Delphi exception. Its starting value is True, which normally you require in order to handle the exception.

However, it is possible to "silence" an exception immediately by calling SysUtils.Abort. You should only set this var False if Abort can be called on the operation that triggered the error without need of further handling to restore conditions to normal. DO NOT use this strategy to hide exceptions that should be raised.

You can, of course, call Abort as a step in an exception-handling routine. In this case, do not change the RaiseException var to False, since you are handling the exception! Abort could, for example, be used to cancel (back out of) a Post or Commit call after raising an exception that informs the user that some invalid data item needs to be fixed up.

The HandleException method

IB Objects has its own session-level HandleException method which is a catch-all for checking, interpreting and handling the most recent errcode returned from an API call. Any positive exception code will be interpreted and the error processing events will be triggered.

If a IB_SQLMonitor active, HandleException will also pass all of the details of the exception to this component as well.

HandleException always gets called by OnError if processing reaches the end of the OnError handler code without encountering an exception type that traps the exception.

A negative errcode will usually result in a Delphi exception being raised.

FIREBIRD/INTERBASE EXCEPTIONS

PSQL (the procedure language) itself provides exception-handling capability, allowing you to branch the normal flow of control of your stored procedure or trigger when Firebird/InterBase throws an error during execution.

Client-side handling
If you choose not to intercept and handle the exceptions inside your procedure or trigger, the procedure will fail and the exception code(s) are passed across to the client via the status_vector array. Choose to manage the exceptions on the server side only if you are certain that the exception can be handled completely and safely without further participation by your client application. If client program handling and/or user input might be needed to fix the exception condition, then you should not attempt to handle it at the server.

Server-side handling
The engine makes exception identification available to PSQL by passing an exception code back to your procedure when an exception occurs. Exceptions which you manage on the server side do not return any parameters to the status_vector array.

Server-side exception handling, by branching to alternative block of code, can be done at several levels using the WHEN symbol, which is somewhat equivalent to **OnError** in Delphi.

Exceptions that the engine can detect

1. When you want a certain block of code to be executed whenever an exception occurs, regardless of the exception, use WHEN ANY, e.g.

```
WHEN ANY DO
BEGIN
   /* Handle any errors generically */
END
```

2. If you need to detect what kind of error occurred and handle it specifically, you can trap the error by polling for its SQLCODE:

```
WHEN SQLCODE -nnn DO
BEGIN
   /* Handle the error according to its kind */
END
```

An SQLCODE is almost always a negative integer, with a correspondence to one or more of the *errcodes* that the engine could pass to the API. In PSQL, an *errcode* is referred to as a GDSCODE. There is not always an exact correspondence between each SQLCODE and a single GDSCODE. One SQLCODE often encompasses several GDSCODEs (*errcodes*).

You can find these SQLCODEs, along with the GDSCODEs, in Chapter 6 of the Language Reference manual. (Perversely, the manual refers to the GDSCODEs as "InterBase Number"!) As you may have already discovered, many of the anticipated exceptions that you are likely to want to handle have unique SQLCODES.

3. If a non-unique SQLCODE is too general for you to detect a particular exception you are interested in, you can poll for the GDSCODE in your WHEN statement:

```
WHEN GDSCODE nnnnnnnnn DO
BEGIN
   /* Handle the exact error */
END
```

User-defined Exceptions

You can go a step further and define your own exceptions. Any exceptions you use must be declared to the database first, in DDL statements. Here is an example:

```
CREATE EXCEPTION INVALID_NAME 'Some name part is invalid' ;
```

Here is a trivial example of how it could be used in a stored procedure:

```
CREATE PROCEDURE ADD_MEMBER (LNAME, M_INITIAL, FNAME, ...) AS
BEGIN
..
  IF (LNAME = '' OR LNAME IS NULL) THEN
    EXCEPTION INVALID_NAME;
  INSERT INTO MEMBERS(LNAME, M_INITIAL, LNAME, ...)
  VALUES (:LNAME, :M_INITIAL, FNAME, ...);
  ...
END
```

If the exception is raised, this procedure will terminate, reversing any operations that have already occurred in the procedure. We can test for the exception on the client side by looking ing at the Message property of the EIB_ISCError that will be raised by a database exception, e.g.

```
...
const

E_INVALID_NAME = 'Some name part is invalid' ;
...

...
on E:EIB_ISCError do
begin
  if E.Message = E_INVALID_NAME then ...
  ...
end;
```

This alone might not always be very useful - in some situations, we might want to catch the exception inside the stored procedure and handle it:

```
CREATE PROCEDURE ADD_MEMBER (LNAME, M_INITIAL, FNAME, ...) AS
BEGIN
..
  IF (LNAME = '' OR LNAME IS NULL) THEN
    EXCEPTION INVALID_NAME;
  INSERT INTO MEMBERS(LNAME, M_INITIAL, LNAME, ...)
  VALUES (:LNAME, :M_INITIAL, FNAME, ...);
  ...
  WHEN EXCEPTION INVALID_NAME DO
  BEGIN
    /* Handle the exception */
  END
END
```

Notice that the actual handling of exceptions (WHEN...DO...) is placed *after* the completion of the statement where you anticipate the error will occur. Don't try to make your PSQL code do any branching out of the main block to pre-empt the completion of the main statements: the procedure "knows" to jump to the appropriate WHEN block.

In your handler, you can do whatever you need to do to handle the exception: insert a row

into an error log, raise an event for which your client is listening, modify bad data, etc. It can even do nothing at all, i.e. it can "silence" the exception.

It is important to remember that, by handling the exception inside your procedure, you prevent the server from returning it to the client via the status_vector array.

Silent Exceptions in Stored Procedures

You can make any exception "silent" by including nothing in the WHEN block. The code for a silent exception would look something like this:

```
...
  WHEN EXCEPTION IGNORE_ME DO
  BEGIN
    /* Ignore the exception */
  END
```

Only silence exceptions if you are sure that so doing will not cause logically corrupt data to be stored, will not interfere with completion of the stored procedure and will not cause further exceptions.

Silent exceptions can be useful in FOR...SELECT loops, to bypass (and possibly log) rows that the loop cannot process. For example, if your stored procedure processes an external file, it can simply reject and log bad data.

When handling exceptions that may occur in loops, the WHEN..DO block must be after the SUSPEND statement.

WHERE TO FIND EXCEPTION-RELATED MATERIAL IN THE IBO SOURCE CODE

IB_Constants.pas - contains message strings associated with all errors which IBO handles
IB_Header.pas - identifies all the ISC errors with their error codes
IB_Session.pas, IBA_Session.IMP, IBA_Session.ING and IB_SessionProps.pas - contain the declarations and implementations of the IB_Error classes. Subclasses typically appear in the IBA_xxxx.IMP units for the components in which the exceptions are raised. These sources can be examined as examples of how to handle specific exceptions.

Also: You can find SQLCODEs, along with the GDSCODEs ("InterBase Error" codes) in Chapter 6 of the InterBase Language Reference manual (LangRef.pdf).

Helen Borrie, with C++ code interpretations by Norman Dunbar
February 2002

As always, if you have any comments or questions feel free to contact the public list server dedicated to IBO.

Jason Wharton <mailto:jwharton@ibobjects.com>
http://www.ibobjects.com
Copyright 2002 Computer Programming Solutions - Mesa AZ
