FINALLY, We Can CATCH Errors THROWn to Us

Paul Guggenheim
Paul Guggenheim & Associates

About PGA

- A Progress Evangelist since 1994, and enlightening Progress programmers since 1996
- Designed several comprehensive Progress courses covering all levels of expertise including The Keys to OpenEdge®
- Author of the Sharp Menu System, a database driven, GUI pull-down menu system.
- White Star Software Strategic Partner
- Consultant/Partner
- TalkPro Board Member
- AppPro Leader
- Head of the Chicago Area Progress User Group
- PUG Challenge Steering Committee Member

Overview

- Traditional Error Handling (TEH)
  - Statement Level
  - NO-ERROR, ERROR-STATUS
  - Block Level
  - Procedure Generated Errors
  - Infinite Loop Protection
  - Exit
- Structured Error Handling (SEH)
  - Error Class Tree
  - Catch
  - Throw
  - Stop
  - AppServer Error Handling
  - Routine-Level, Block-Level
  - Finally
Overview (cont.)

- Best Practices for Structured Error Handling
  - Top-Down Programming Model
  - Event-Driven Programming Model

FINALLY, We Can CATCH Errors

TEH – Statement Level

- There are many ABL statements that allow the NO-ERROR keyword.
  - These prevent a procedure generated error.

```plaintext
find first customer where customer = 1000 no-error.
if available customer ...
  assign intvar = "abc" no-error.
  if error=statu=error ...
```

TEH – Block Level

- The default error handling is UNDO, RETRY for PROCEDURE, FOR and REPEAT blocks and UNDO, RETURN ERROR for database trigger blocks.

```plaintext
repeat:
  prompt-for student, studentID.
  find student using studentid.
  update ssnname snlname.
end. */ repeat */
```

- When the FIND fails, the REPEAT block is undone and then retried, re-executing the PROMPT-FOR statement.
**TEH – Infinite Loop Protection**

- Progress provides infinite loop protection on procedure generated errors that occur in blocks with no user interaction.

```plaintext
repeat:
    /* abbreviated form of: where studentid = -2. */
    find student -1.
end. /* repeat */
```
- The default undo, retry for the repeat block changes to undo, leave since there is no user interaction or retry function in the block.

---

**TEH – Stop Condition**

- A stop condition occurs when:
  - A stop key is pressed CTRL-C (Unix) or CTRL-BREAK (Windows)
  - The stop statement
  - Run program not found
  - Lose a db connection
  - CRC values don't match for program/database

---

**TEH – Default Stop Processing**

- By default, a stop condition will cause the transaction block to be undone, and will branch to the first called program of the session (-p) (with the exception of losing a db connection).

```plaintext
repeat:
    find first student.
    display student.
    update firstname lastname.
    stop.
end. /* repeat */
```
- The above program will not display “Program Ended.”
**TEH – Overriding the Stop Condition**

- Use the ON STOP UNDO, RETRY, (LEAVE, NEXT, RETURN) phrase to override the default stop condition on FOR, REPEAT and DO blocks.

  ```plaintext
  repeat on stop undo, leave:
  find first student.
  display student.d.
  update student.fname, student.lname.
  stop.
  end. /* repeat */
  display "Program Ended.".
  ```

- The above program will display "Program Ended.".

---

**TEH - Disadvantages**

- Since NO-ERROR is not the default, a developer must remember to specify this keyword on each statement where an error may occur.

- Handling errors with NO-ERROR is performed inconsistently, i.e. ERROR-STATUS in most cases but AVAILABLE and AMBIGUOUS functions in others.

- A RETURN ERROR statement may be used to **manually** return an error to the called procedure.

- A block label may be used to **manually** branch to the next outer block.

- It is difficult to include additional information when passing the error outside of the error block.

---

**SEH - Advantages**

- Catch blocks handle all error types except QUIT processing.

- Because structured errors are class objects, developers have the ability to create user-defined error types based on the application error class.

- SEH has facilities to propagate errors up the call stack.
FINALLY, We Can CATCH Errors

PUG Challenge Americas 2017

Manchester, NH

THrown To Us!

– 7

Copyright © 2017

Paul Guggenheim & Associates

Progress.Lang.ProError

end.

Copyright © 2017

Paul Guggenheim & Associates

Progress.Lang.ProError

end.

Copyright © 2017

Paul Guggenheim & Associates

Progress.Lang.ProError

end.
FINALLY, We Can CATCH Errors

PUG Challenge Americas 2017
Manchester, NH

do on error undo, leave:
find first student where studentid = 9999.
catch syserrvar as Progress.Lang.Syserror:
  message "inside catch syserrvar block" view-as alert-box.
end catch.
end. /* do on error undo, leave */
display "program ended".

The above example uses the Syserror class to trap the procedure generated error.

Copyright © 2017
PUG Challenge Americas
Manchester, NH

do on error undo, leave:
find first student where studentid = 9999.
catch apperrvar as Progress.Lang.AppError:
  message "inside catch apperrvar block" view-as alert-box.
end catch.
end. /* do on error undo, leave */
display "program ended".

The above example uses the AppError class and will not trap the procedure generated error.
**Catch Example**

```pascal
do an error undo, leave:
  run abc.
  catch apperror as progress:apperror
    message "inside catch apperror block" skip
    return-value
    view-as alert-box.
  end catch.
end. /* do an error undo, leave */
```

**Catch Block**

- In the previous example, using `RETURN ERROR` counts as an application error which is then trapped by the `CATCH` block.

**THROWing Errors**

- The `THROW` keyword transfers the error to the enclosing (next outer) block, where it can be trapped with a `CATCH` block.
  ```pascal
do an error undo, throw:
  find first student where studentid = 5000.
end.
```
- Since there is no `CATCH` block in the containing procedure block, the procedure generated error message is displayed.
do on error undo, throw:
  find first student where studentid = 5000.
end.
catch syserror as Progress.Lang.SysError:
  message "Catch in containing procedure (main block"
  view-as alert-box.
end.

• In the above example, the DO block throws the error to the containing procedure block where it is caught by the SysError catch block.

do transaction on error undo, throw:
  find first student where studentid = 5000.
  catch syserror as Progress.Lang.SysError:
    message "Catch in do transaction block"
    view-as alert-box.
  end.
end.
catch syserror as Progress.Lang.SysError:
  message "Catch in containing procedure (main) block" view-as alert-box.
end.

• In the previous example, the CATCH block in the DO block traps the procedure generated error instead of it being THROWn to the containing procedure block.

• Here is the order of precedence when trapping procedure generated errors:
  1. NO-ERROR Option on a statement
  2. SysError CATCH Block in the ASSOCIATED block where the error occurred
  3. Explicit ON ERROR phrase
  4. Implicit ON ERROR phrase
Stop Processing

In OpenEdge 11.7, Progress introduces Stop Objects.
- These new features are listed as Technical Preview.
- This means that the Stop Objects are supported in development but not yet in production until more testing can be performed.
- To activate the Stop object features, use the startup client case-sensitive parameter -catchStop 1

Stop Processing

- Two Top Stop Object Classes:
  1. Progress.Lang.StopError
  2. Progress.Lang.Stop

- Progress.Lang.StopError are system caused stop conditions e.g.
  1. Run Program not found
  2. Database disconnected
  3. CRC values don't match

Stop Processing

- Two Top Stop Object Classes:
  1. Progress.Lang.StopError
  2. Progress.Lang.Stop

- Progress.Lang.Stop are application caused stop conditions e.g.
  1. Stop key pressed (CTRL-Break on Windows)
  2. STOP Statement
  3. STOP-AFTER Phrase
  4. On STOP Phrase
FINALLY, We Can CATCH Errors

stop:
find first student.
display studentid.
update sfname sname.
stop.
catch stopvar as Progress,Lang,Stop:
message "Stop Error occurred" view-as alert-box.
end.
end. /* repeat */
display "Program Ended.".

Stop Processing

- In the previous example, the stop statement was used so therefore the stop object is used to catch the error.
- Whether the stop event is caught or not, the transaction is always undone.
- The StopError object would not be able to trap for this error message.

stop: on stop undo,leave :
find first student.
display studentid.
update sfname sname.
stop.
catch stopvar as Progress,Lang,Stop:
message "Stop Error occurred" view-as alert-box.
end.
end. /* repeat */
display "Program Ended.".
FINALLY, We Can CATCH Errors

STOP PROCESSING

- The ON STOP phrase is another example of a stop condition trapped by the Progress.Lang.Stop object.
- In the previous example, the ON STOP phrase was used on the repeat. When the STOP statement is executed, the CATCH block traps the STOP condition thereby preventing the ON STOP phrase from performing an undo, leave.

```
repeat stop-after 3:
  find first student.
  display studentid.
  update firstname lastname.
  pause 4.
  catch stopvar as Progress.Lang.Stop:
    message "Stop Error occurred" view-as alert-box.
  end.
end; /* repeat */
display "Program End.".
```

STOP PROCESSING

- The STOP-AFTER phrase is third example of a stop condition trapped by the Progress.Lang.Stop object.
- In the previous example, the STOP-AFTER phrase was used on the repeat. The pause of 4 seconds exceeded the STOP-AFTER amount of 3 seconds, thus causing the stop event to occur which in turn is trapped by the Stop CATCH block.
StopError Processing

- Attempting to run a program that doesn’t exist is an example of a stop condition trapped by the Progress.Lang.StopError object.
- In the previous example, the program programnotfound.p was attempted to be run. Since this program doesn’t exist, this stop event is trapped by the StopError CATCH block.
- The Progress.Lang.Stop object would not be able to trap for this error message.

AppServer Error Handling

- Class-based error or stop objects can be THROWn from an AppServer and handled by a CATCH block on the client.
- In order for this to be possible, the following conditions must be satisfied:
  - A user-defined error class must be defined on both the client and the server.
  - This user-defined error class must be defined identically on both the client and server.
  - The class must be defined as Serializable.
- If these conditions aren’t performed then tradition error handling (TIEH) must be used.
FINALLY, We Can CATCH Errors

- The PGAAppError class inherits the Progress.Lang.AppError class and is defined as Serializable.
- This error class will be used on both the client and server.
- When testing the STOP processing on the AppServer, make sure the ‒catchStop 1 is used on the agent connect parameter.
FINALLY, we can catch errors.

```java
/* testapperror.p */

repeat i = 1 to 3:
    run apperror.p as NOAppError.
    catch pgapperror as NOAppError:
        message "inside catch pgapperror block": skip
        pgapperror = exceptionInfo, skip
        view-as alert-box.
    end catch.
CATCH err AS Progress.Lang.StopError:
    message "inside catch stoperror in testapperror.p": skip
    View-as alert-box.
    and catch.
end.
```

The testapperror.p procedure loops twice through a REPEAT block containing two CATCH blocks for trapping PGAppError type and StopError type class error objects.

The first iteration passes a numeric 1 to the apperror.p program which means that apperror.p throws a PGAppError back to the client.

The second iteration passes a numeric 2 to the apperror.p program which means that apperror.p throws a StopError back to the client.
THROWing Errors

run findfirst;
run findlast;
procedure findfirst:
    find first student where studentid = 5000.
end.
procedure findlast:
    find last teacher where teacherid = 5000.
end.
catch e as error as Progress.lang.SysError;
message "Catch in containing procedure (main) block"
view as alert-box.
end.

THROWing Errors

- The default error handling for internal procedures with no user interaction is undo, leave.
- In the previous example, the run findfirst and run findlast calls cause a procedure generated error with no user interaction. This causes the default error handling to occur. Both error messages are displayed.
- What if we want these errors to be trapped somewhere else?

Changing ROUTINE-LEVEL Behavior

- By default, all routine level blocks (except database trigger blocks) with no user interaction, process procedure generated errors with undo, leave.
- The following are blocks are routine level blocks:
  - Containing Procedure
  - User-Defined Function
  - DB Trigger Block
  - Class Method
  - Class Destructor
  - Internal Procedure
  - DB Trigger Procedure
  - User Interface Trigger
  - Class Constructor
  - Class Property Accessor
Finally, we can catch errors.

PUG Challenge Americas 2017

Manchester, NH

June 4 – 7, 2017

Copyright © 2017
FINALLY Block

- Like the CATCH block, the FINALLY block is an END block. It must appear as the last END block in the ASSOCIATED block.

Associated Block

- Catch Block
- Finally Block

FINALLY Block

- The FINALLY block executes after:
  - Successful execution of a non-iterating associated block
  - Each loop of an iterating associated block
  - CATCH block traps an error that occurred in the associated block
  - Procedure generated ERROR not trapped by a CATCH block

- The FINALLY block will not execute when:
  - A STOP condition occurs and is not trapped
  - A QUIT statement is executed and is not trapped

FINALLY Block

- Here is a simple example:
  find student where studentid = 1.
  display sfirname s1astname.
  finally:
    message "end of program" view-as alert-box.
  end.
  - The containing procedure executes without an error and the finally block executes.
SEH Best Practices

- This section will attempt to illustrate the best practices for structured error handling.
- The goal is to use the new features as efficiently as possible and to centralize error processing for optimal maintainability.
- Two environments will be demonstrated:
  - Top Down Programming
  - Event Driven Programming

Top-Down Menu

SEH Best Practices

- The following are the programs in a top down example:
  
  - tmenu.p (Top-down menu)
  - deptmaint.p (Dpt: Maintenance)
  - teachmaint.p (Teacher Maintenance)
Top-Down Menu

- Default Error Processing
  - When a procedure generated error occurs in the department containing procedure, the error is not trapped and that block is undone, returning back to the tdmENU.p procedure.
  - When a procedure generated error occurs in the department repeat block, the error is trapped implicitly and that repeat block is undone and retried.
  - When a procedure generated error occurs in the teacher containing procedure, the error is not trapped and that block is undone, returning back to the deptmain.p procedure.

Top Down Menu

- Default Error Processing (continued)
  - When a procedure generated error occurs in the teacher repeat block, the error is trapped implicitly and that repeat block is undone and retried.

Top Down Menu

- Structured Error Handling Changes
  - Added a CATCH block to the associated REPEAT blocks in tdmenus.h, deptmainseh.p and teachereh.p.
  - Added ROUTINE-LEVEL ON ERROR UNDO, THROW to the above programs.
FINALLY, We Can CATCH Errors

PUG Challenge Americas 2017

THROWN TO US!

Manchester, NH

Copyright © 2017
Finally, we can catch errors.

The PUG Challenge Americas 2017

Manchester, NH

Copyright © 2017

Paul Guggenheim & Associates

21

Event Driven Model

- More work needs to be performed by structured error handling to trap errors in an event driven model environment.
- By default, user interface triggers do not throw procedure generated errors.
- The ROUTINE-LEVEL ON ERROR UNDO, THROW has no effect on UI triggers.
- Studbr.p is executed from the Progress editor.
- Studbr.p runs brbar.p persistently.
- Brbar.p runs act.p and chgp.p persistently.

Event Driven Model

- Like the top down model, the statement ROUTINE-LEVEL ON ERROR UNDO, THROW is added to each program.
- CATCH Blocks are added to each containing procedure in each program.
- The rpterr internal procedure block is added to studarsh.p and brbansh.p to receive error objects called from other persistent procedures.
- CATCH blocks are added to UI trigger blocks. These blocks call the rpterr internal procedure and pass the error object as an input parameter.
- As a best practice, CATCH blocks should be added to all event handler blocks such as UI trigger blocks.
Event Driven Model

- When CHRG.P is instantiated, a procedure generated error occurs when the student charge record (stuchrg) is not found. The CATCH block in the containing procedure catches the error and then runs RPTERR in BRBRASEH.P.
- Next while inside internal procedure RPTERR in BRBRASEH.P, run RPTERR is called in STUDBR.P with the error object being passed as an input parameter again. This allows the error to be propagated back to RPTERR in the gateway procedure.
- Wait-for cannot trap for an error and Progress doesn’t throw the error to the procedure where the wait-for is run.

Event Driven Model

- When the activity button is pressed, the Ui trigger on choose of BACT is executed in BRBRASEH.P.
- It runs the CHGFACT internal procedure in ACTSEH.P which produces a procedure generated error when the activity record cannot be found.
- Since there is no user interaction, the CHGFACT procedure performs an undo, return error to the Ui trigger in BRBRASEH.P.
- The CATCH block in this Ui trigger then traps this error and calls RPTERR in STUDBRASEH.P like before.

Summary

- Structured Error Handling (SEH) traps procedure generated errors in a more consistent manner than traditional error handling.
- SEH may be used in all programming models.
- SEH makes it easier to propagate errors up the invocation chain and centralize error processing.
- SEH is not a magic bullet. Care and planning must be taken when using it.
- SEH will not trap QUIT conditions.
FINALLY, We Can CATCH Errors

PUG Challenge Americas 2017

Questions