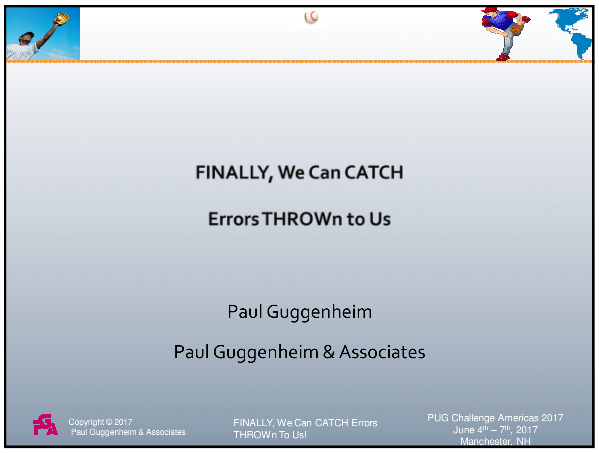
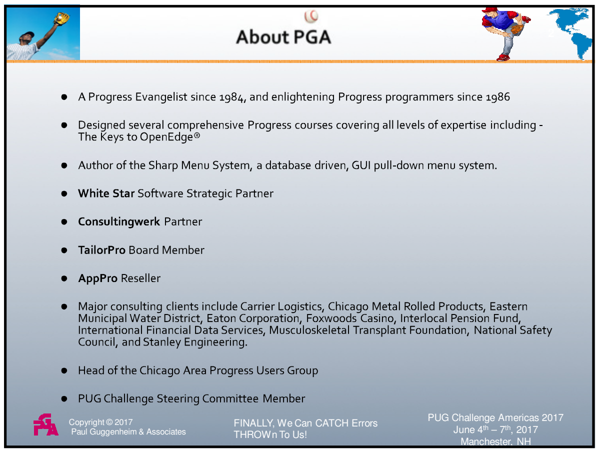


FINALLY, We Can CATCH Errors THROWn to Us

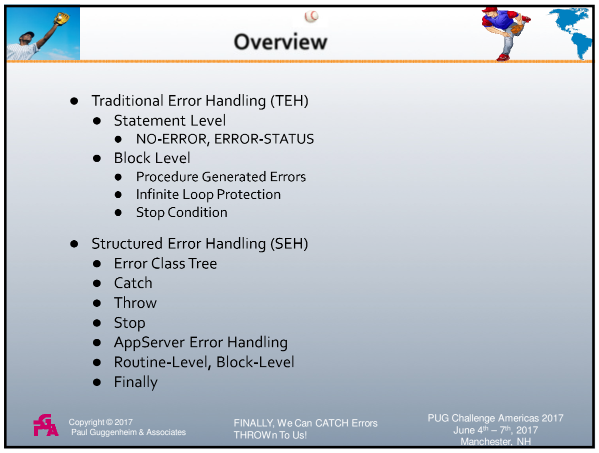
Paul Guggenheim

Paul Guggenheim & Associates



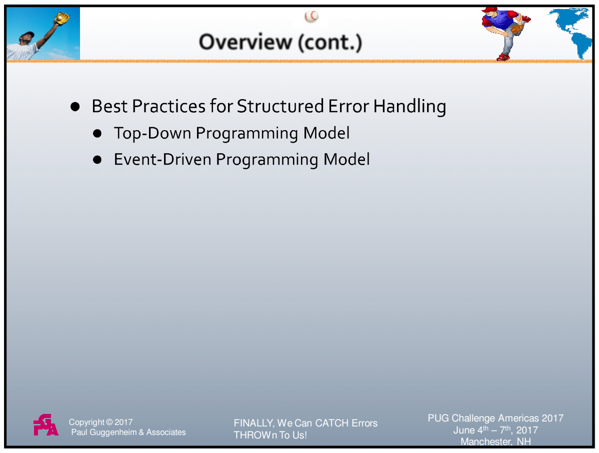
About PGA

- A Progress Evangelist since 1984, and enlightening Progress programmers since 1986
- 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
- **Consultingwerk** Partner
- **TailorPro** Board Member
- **AppPro** Reseller
- Major consulting clients include Carrier Logistics, Chicago Metal Rolled Products, Eastern Municipal Water District, Eaton Corporation, Foxwoods Casino, Interlocal Pension Fund, International Financial Data Services, Musculoskeletal Transplant Foundation, National Safety Council, and Stanley Engineering.
- Head of the Chicago Area Progress Users Group
- PUG Challenge Steering Committee Member



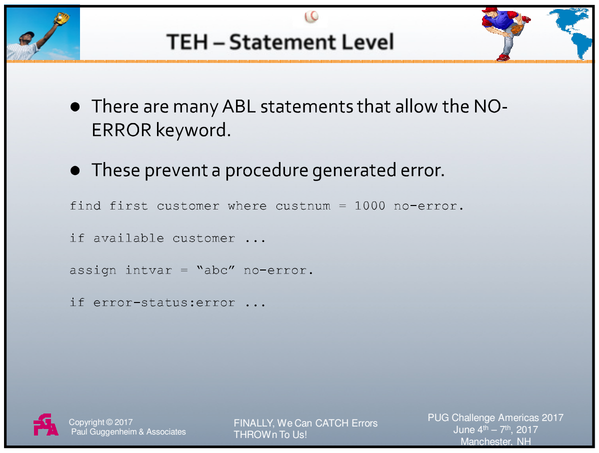
Overview

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



Overview (cont.)

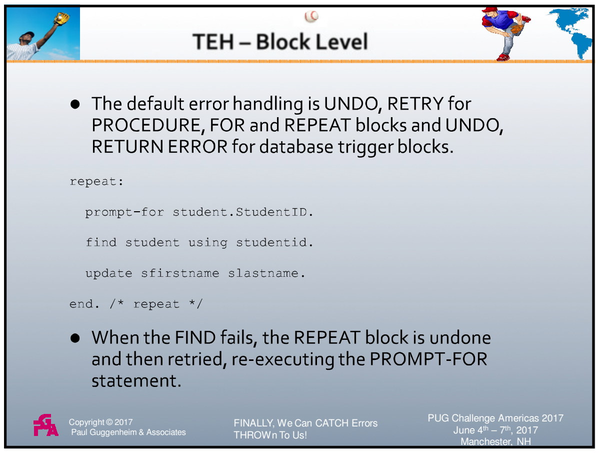
- Best Practices for Structured Error Handling
  - Top-Down Programming Model
  - Event-Driven Programming Model



TEH – Statement Level

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

```
find first customer where custnum = 1000 no-error.

if available customer ...

assign intvar = "abc" no-error.

if error-status:error ...
```

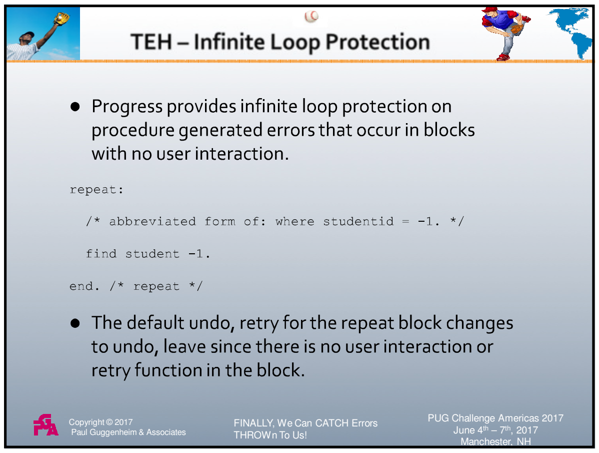


TEH – Block Level

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

```
repeat:
  prompt-for student.StudentID.
  find student using studentid.
  update sfirstname slastname.
end. /* repeat */
```

- When the FIND fails, the REPEAT block is undone and then retried, re-executing the PROMPT-FOR statement.

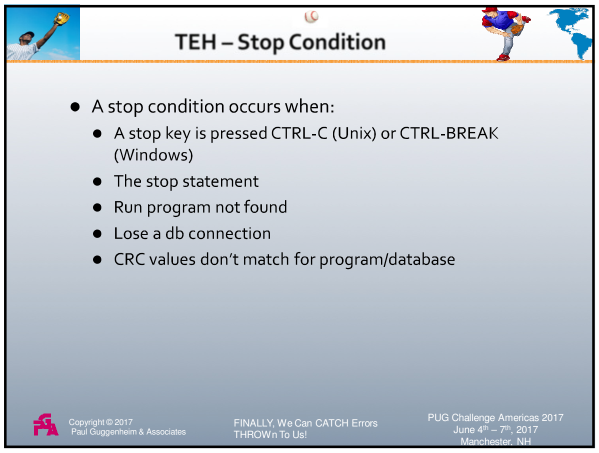


TEH – Infinite Loop Protection

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

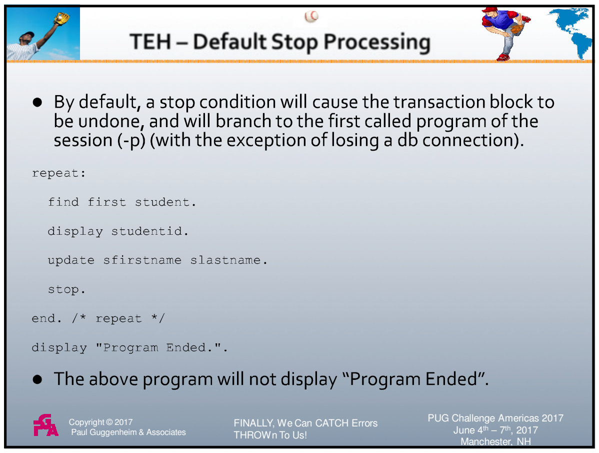
```
repeat:
  /* abbreviated form of: where studentid = -1. */
  find student -1.
end. /* repeat */
```

- 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

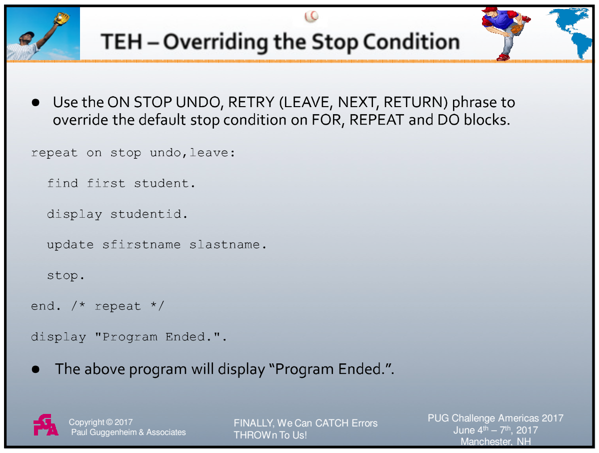


TEH – Default Stop Processing

- 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).

```
repeat:
  find first student.
  display studentid.
  update sfirstname slastname.
  stop.
end. /* repeat */
display "Program Ended.".
```

- The above program will not display "Program Ended".

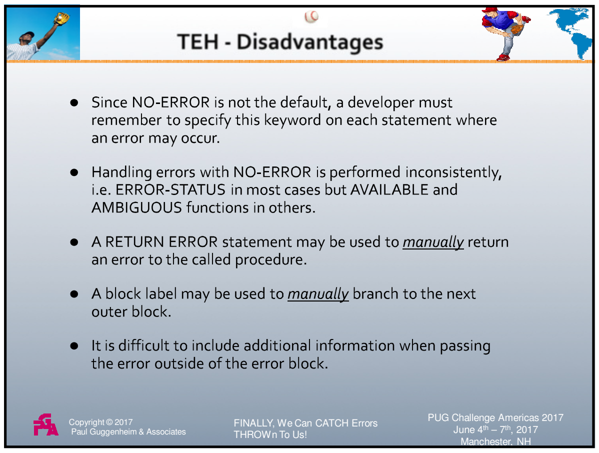


TEH – Overriding the Stop Condition

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

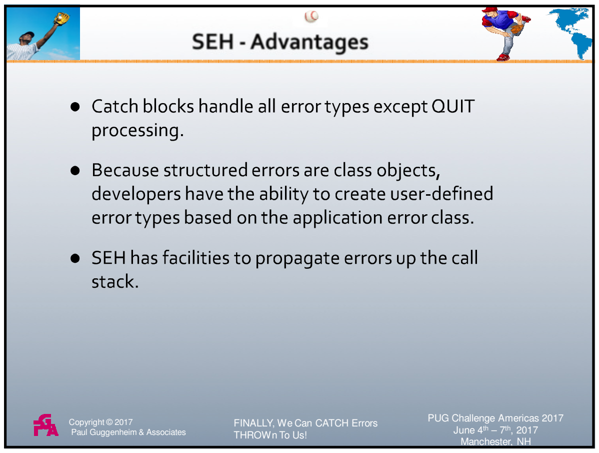
```
repeat on stop undo,leave:
  find first student.
  display studentid.
  update sfirstname slastname.
  stop.
end. /* repeat */
display "Program Ended.".
```

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



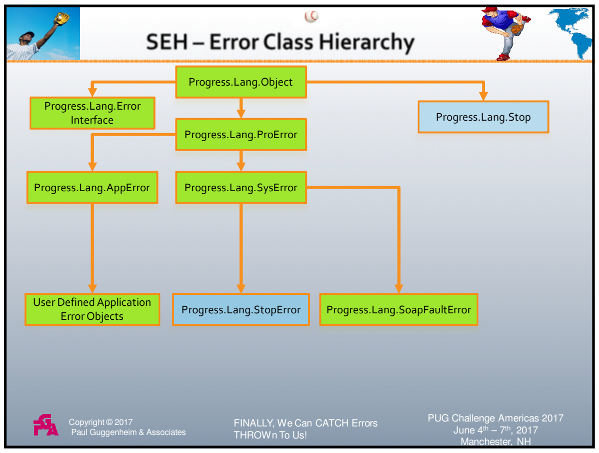
TEH - Disadvantages

- 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.

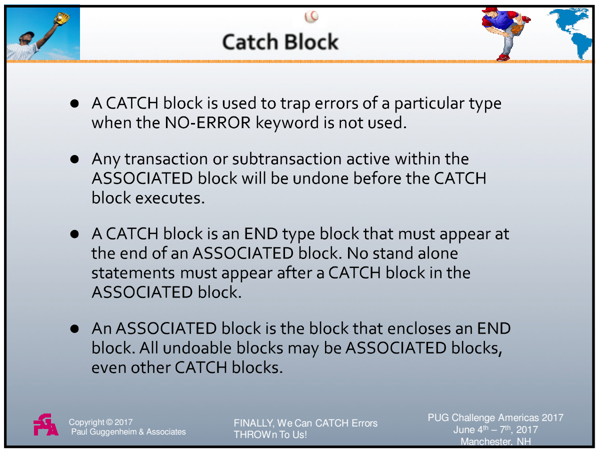


SEH - Advantages

- 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.

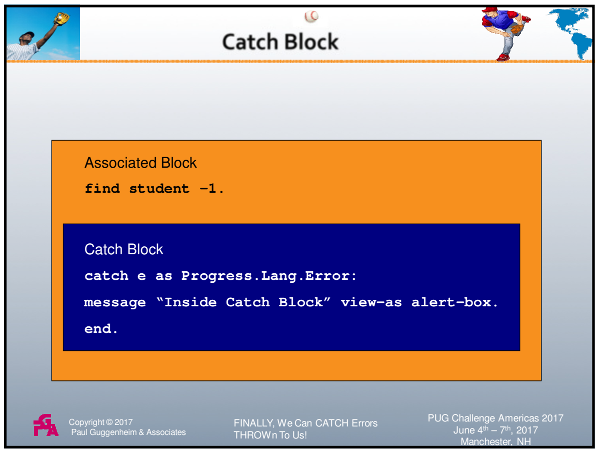


SEH – Error Class Hierarchy



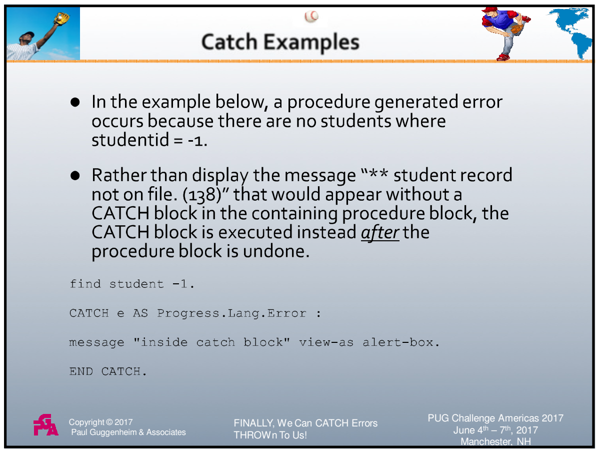
Catch Block

- A CATCH block is used to trap errors of a particular type when the NO-ERROR keyword is not used.
- Any transaction or subtransaction active within the ASSOCIATED block will be undone before the CATCH block executes.
- A CATCH block is an END type block that must appear at the end of an ASSOCIATED block. No stand alone statements must appear after a CATCH block in the ASSOCIATED block.
- An ASSOCIATED block is the block that encloses an END block. All undoable blocks may be ASSOCIATED blocks, even other CATCH blocks.



Catch Block

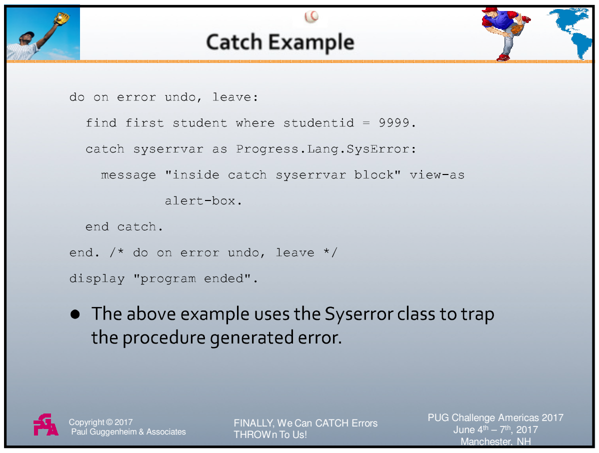
Associated Block
```
find student -1.
```
Catch Block
```
catch e as Progress.Lang.Error:
message "Inside Catch Block" view-as alert-box.
end.
```



Catch Examples

- In the example below, a procedure generated error occurs because there are no students where studentid = -1.
- Rather than display the message "** student record not on file. (138)" that would appear without a CATCH block in the containing procedure block, the CATCH block is executed instead *after* the procedure block is undone.

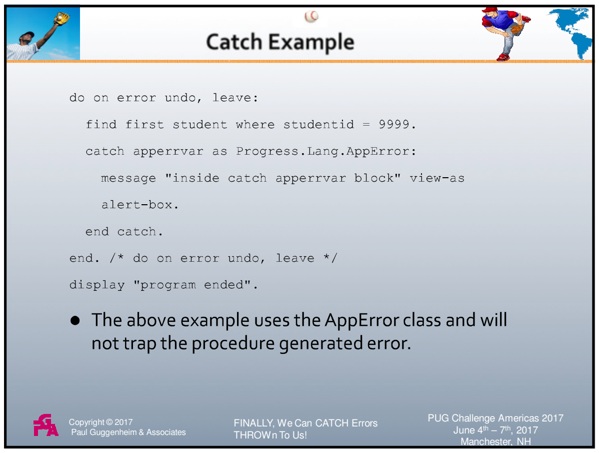
```
find student -1.

CATCH e AS Progress.Lang.Error :

message "inside catch block" view-as alert-box.

END CATCH.
```



Catch Example

```
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".
```

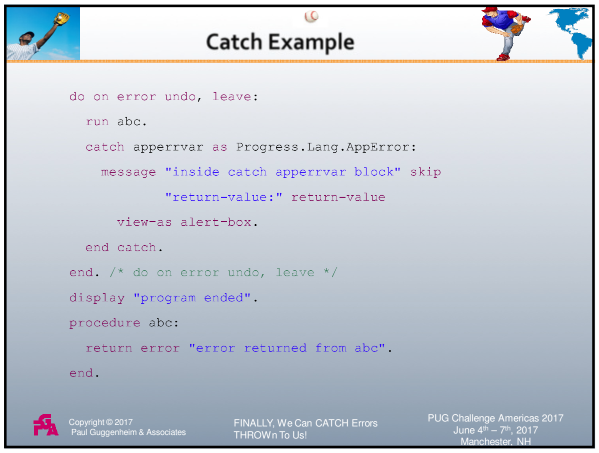
- The above example uses the Syserror class to trap the procedure generated error.



Catch Example

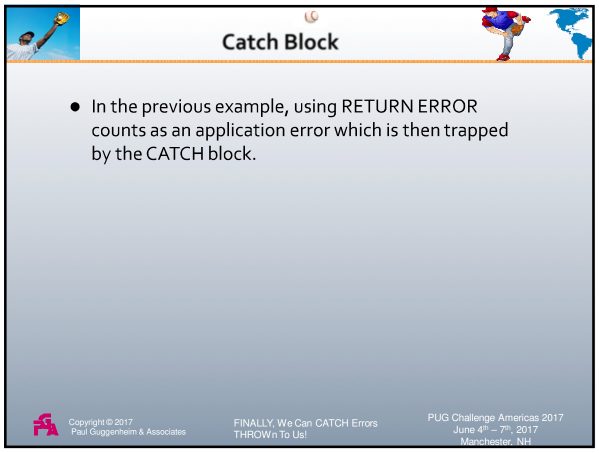
```
do on error undo, leave:
  find first student where studentid = 9999.
  catch apperrvar as Progress.Lang.AppError:
    message "inside catch apperrvar block" view-as
    alert-box.
  end catch.
end. /* do on error undo, leave */
display "program ended".
```

- The above example uses the AppError class and will not trap the procedure generated error.



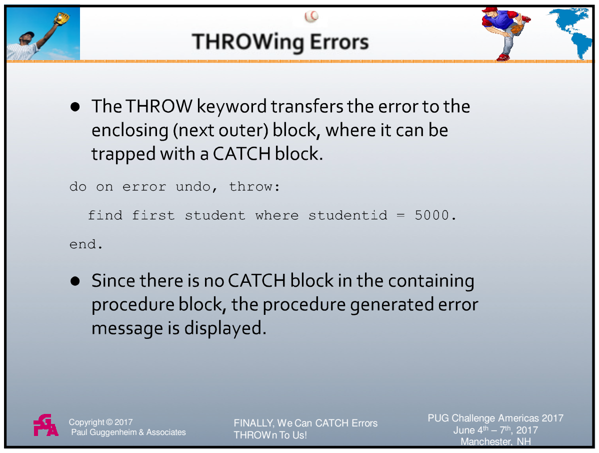
Catch Example

```
do on error undo, leave:
  run abc.
  catch apperrvar as Progress.Lang.AppError:
    message "inside catch apperrvar block" skip
            "return-value:" return-value
      view-as alert-box.
  end catch.
end. /* do on error undo, leave */
display "program ended".
procedure abc:
  return error "error returned from abc".
end.
```



Catch Block

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

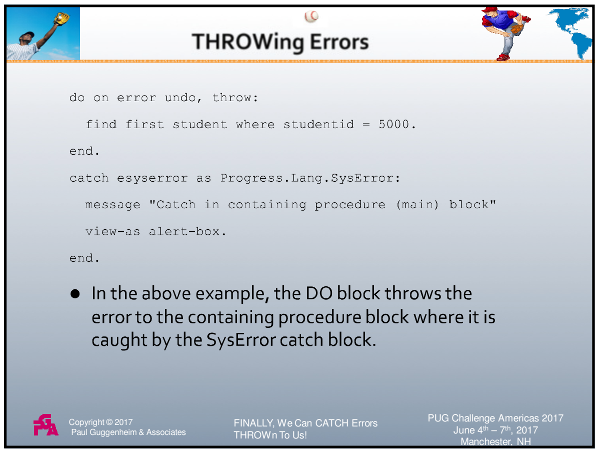


THROWing Errors

- The THROW keyword transfers the error to the enclosing (next outer) block, where it can be trapped with a CATCH block.

```
do on error undo, throw:
  find first student where studentid = 5000.
end.
```

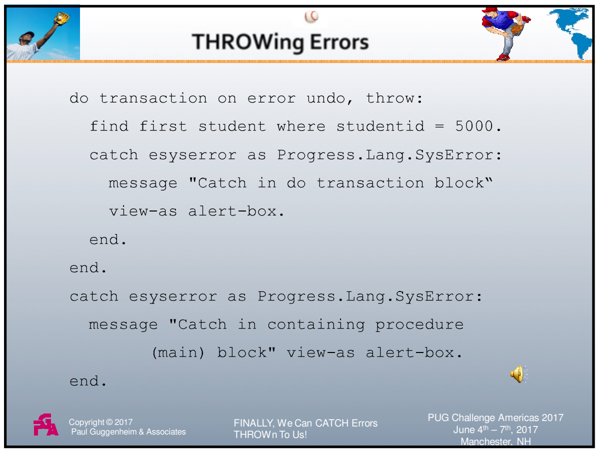
- Since there is no CATCH block in the containing procedure block, the procedure generated error message is displayed.



THROWing Errors

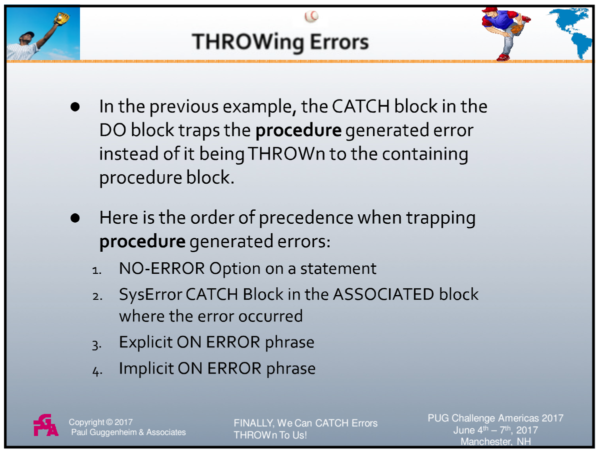
```
do on error undo, throw:
  find first student where studentid = 5000.
end.
catch esyserror as Progress.Lang.SysError:
  message "Catch in containing procedure (main) block"
  view-as alert-box.
end.
```

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



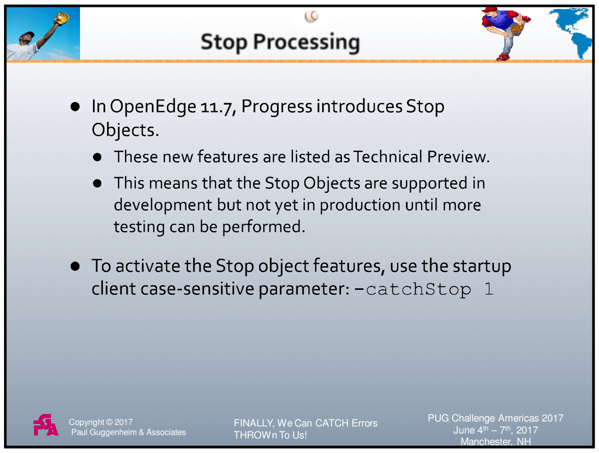
THROWing Errors

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



THROWing Errors

- 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

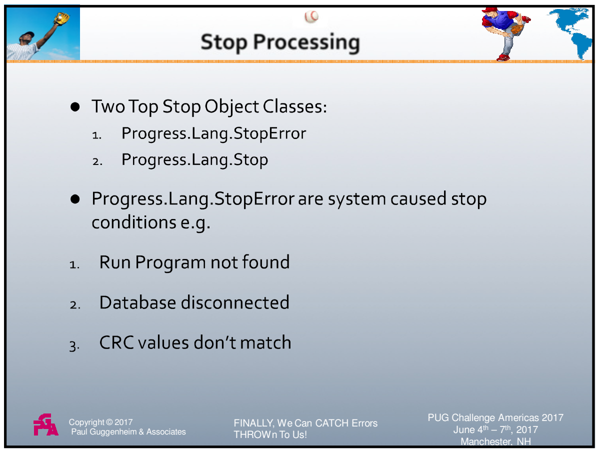


Stop Processing

- 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`

FINALLY, We Can CATCH Errors THROWn To Us!

PUG Challenge Americas 2017
June 4th – 7th, 2017
Manchester, NH



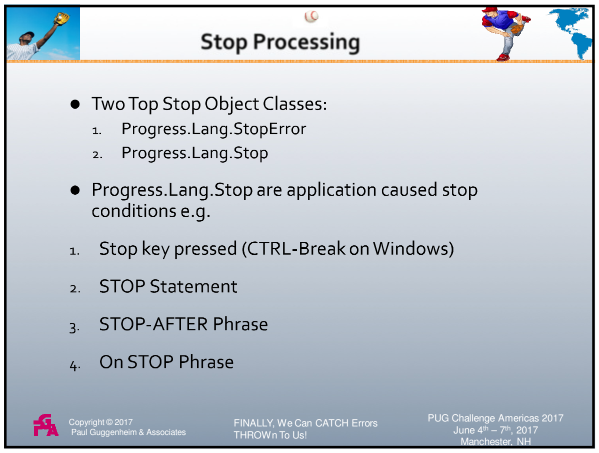
Stop Processing

- 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

FINALLY, We Can CATCH Errors THROWn To Us!

PUG Challenge Americas 2017
June 4th – 7th, 2017
Manchester, NH



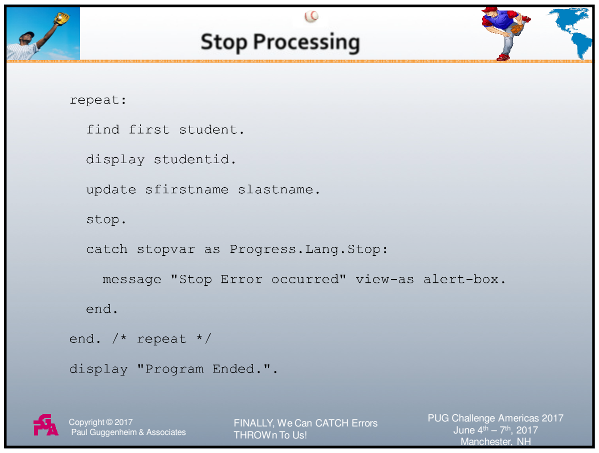
Stop Processing

- 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

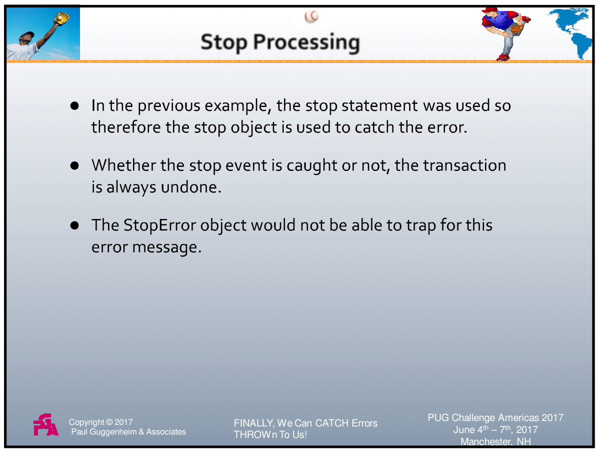
FINALLY, We Can CATCH Errors THROWn To Us!

PUG Challenge Americas 2017
June 4th – 7th, 2017
Manchester, NH



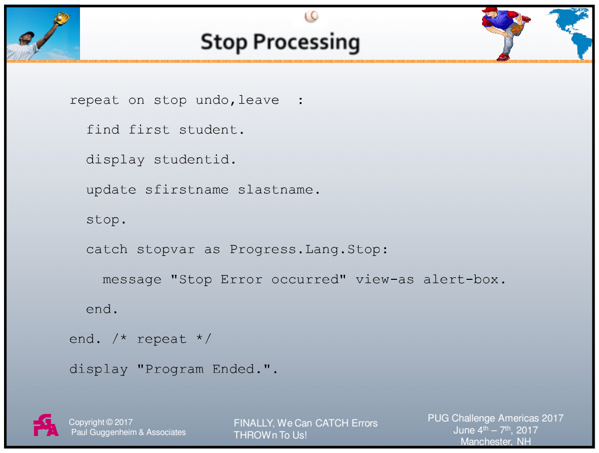
Stop Processing

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



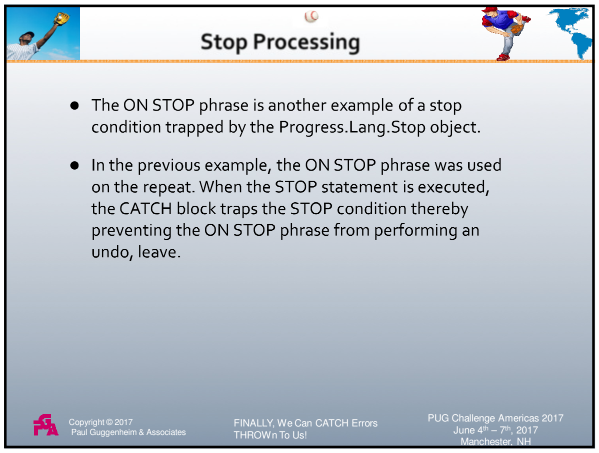
Stop Processing

- 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.



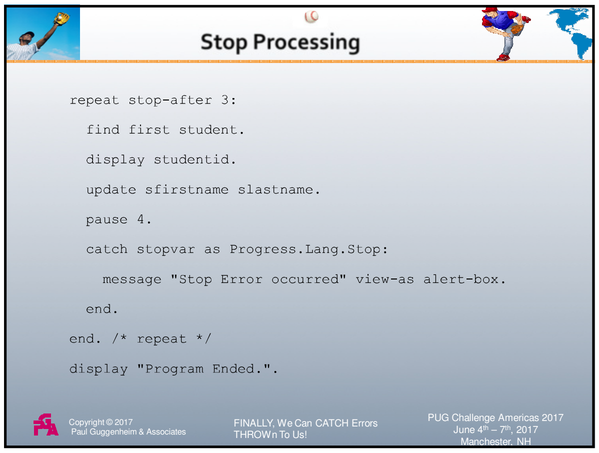
Stop Processing

```
repeat on stop undo,leave  :
  find first student.
  display studentid.
  update sfirstname slastname.
  stop.
  catch stopvar as Progress.Lang.Stop:
    message "Stop Error occurred" view-as alert-box.
  end.
end. /* repeat */
display "Program Ended.".
```



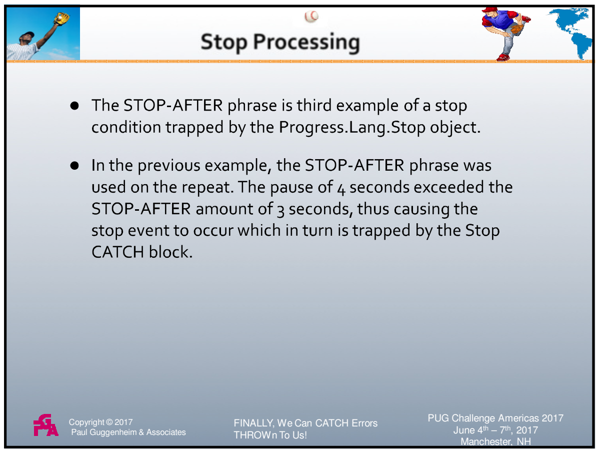
Stop Processing

- 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.



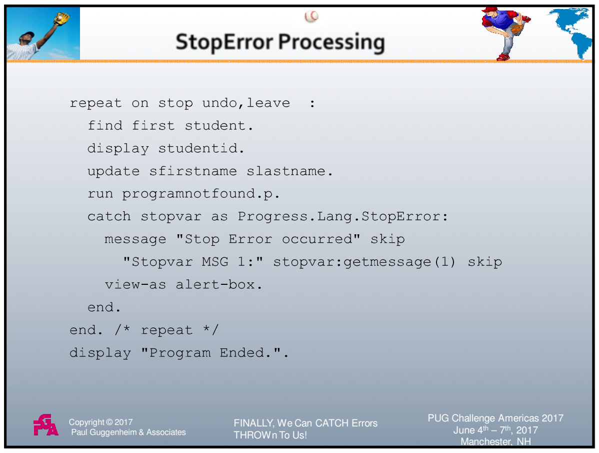
Stop Processing

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



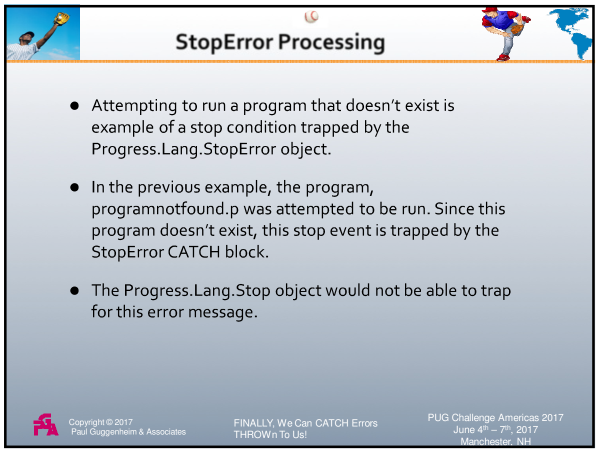
Stop Processing

- 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.



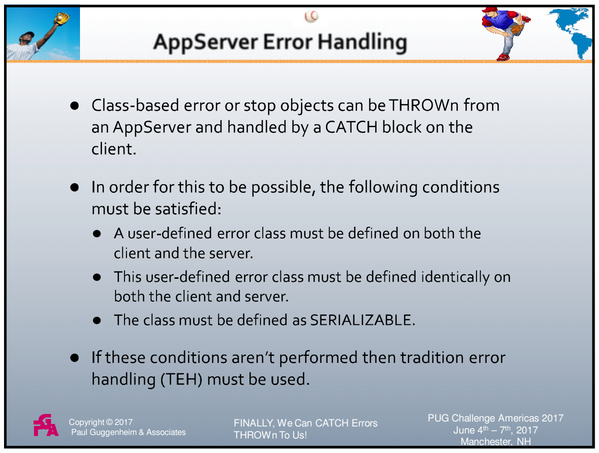
StopError Processing

```
repeat on stop undo,leave  :
  find first student.
  display studentid.
  update sfirstname slastname.
  run programnotfound.p.
  catch stopvar as Progress.Lang.StopError:
    message "Stop Error occurred" skip
      "Stopvar MSG 1:" stopvar:getmessage(1) skip
    view-as alert-box.
  end.
end. /* repeat */
display "Program Ended.".
```



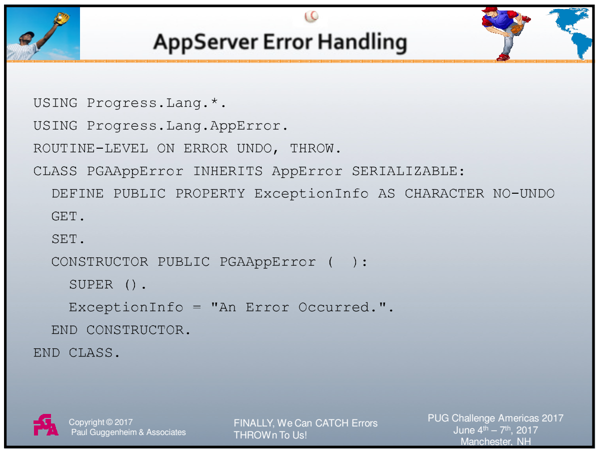
StopError Processing

- Attempting to run a program that doesn't exist is 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.



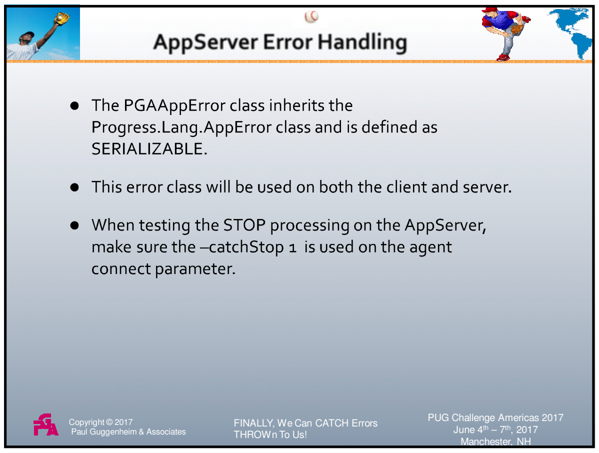
AppServer Error Handling

- 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 (TEH) must be used.



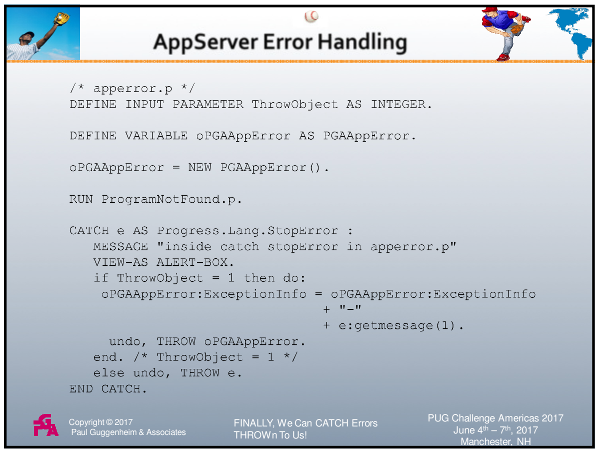
AppServer Error Handling

```
USING Progress.Lang.*.
USING Progress.Lang.AppError.
ROUTINE-LEVEL ON ERROR UNDO, THROW.
CLASS PGAAppError INHERITS AppError SERIALIZABLE:
  DEFINE PUBLIC PROPERTY ExceptionInfo AS CHARACTER NO-UNDO
  GET.
  SET.
  CONSTRUCTOR PUBLIC PGAAppError (  ):
    SUPER ().
    ExceptionInfo = "An Error Occurred.".
  END CONSTRUCTOR.
END CLASS.
```



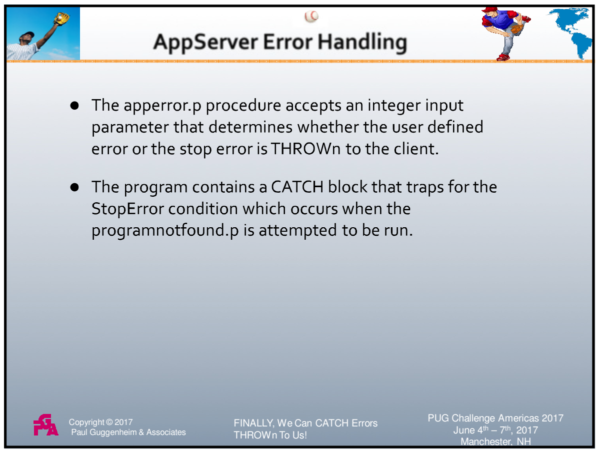
AppServer Error Handling

- 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.



AppServer Error Handling

```
/* apperror.p */
DEFINE INPUT PARAMETER ThrowObject AS INTEGER.

DEFINE VARIABLE oPGAAppError AS PGAAppError.

oPGAAppError = NEW PGAAppError().

RUN ProgramNotFound.p.

CATCH e AS Progress.Lang.StopError :
   MESSAGE "inside catch stopError in apperror.p"
   VIEW-AS ALERT-BOX.
   if ThrowObject = 1 then do:
    oPGAAppError:ExceptionInfo = oPGAAppError:ExceptionInfo
                                + "-"
                                + e:getmessage(1).
     undo, THROW oPGAAppError.
   end. /* ThrowObject = 1 */
   else undo, THROW e.
END CATCH.
```



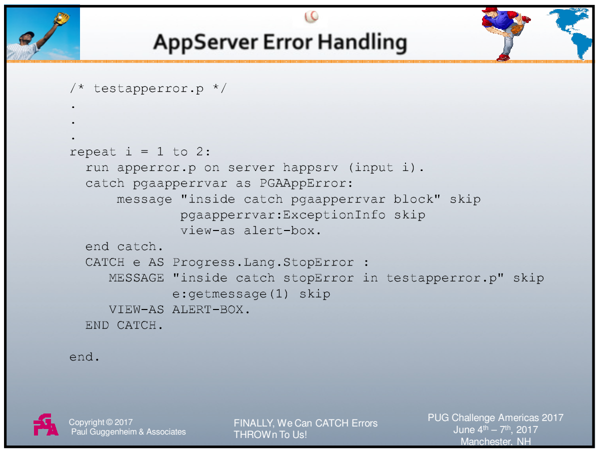
AppServer Error Handling

- The apperror.p procedure accepts an integer input parameter that determines whether the user defined error or the stop error is THROWn to the client.
- The program contains a CATCH block that traps for the StopError condition which occurs when the programnotfound.p is attempted to be run.

Copyright © 2017
Paul Guggenheim & Associates

FINALLY, We Can CATCH Errors THROWn To Us!

PUG Challenge Americas 2017
June 4th – 7th, 2017
Manchester, NH



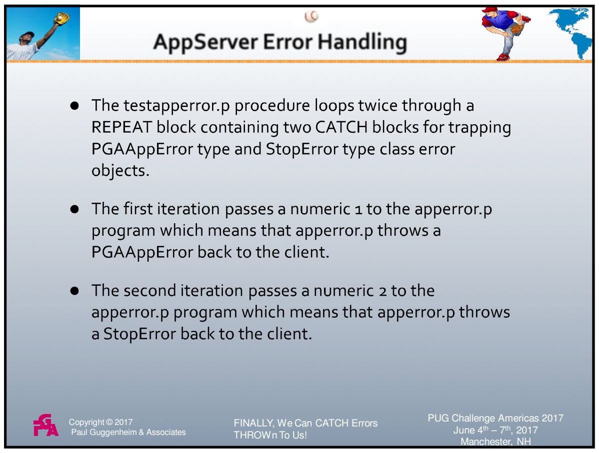
AppServer Error Handling

```
/* testapperror.p */
.
.
.
repeat i = 1 to 2:
  run apperror.p on server happsrv (input i).
  catch pgaapperrvar as PGAAppError:
      message "inside catch pgaapperrvar block" skip
              pgaapperrvar:ExceptionInfo skip
              view-as alert-box.
  end catch.
  CATCH e AS Progress.Lang.StopError :
     MESSAGE "inside catch stopError in testapperror.p" skip
             e:getmessage(1) skip
     VIEW-AS ALERT-BOX.
  END CATCH.

end.
```

Copyright © 2017
Paul Guggenheim & Associates

FINALLY, We Can CATCH Errors THROWn To Us!

PUG Challenge Americas 2017
June 4th – 7th, 2017
Manchester, NH



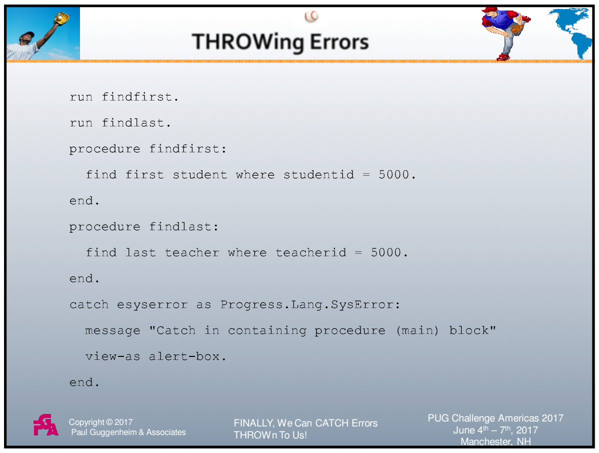
AppServer Error Handling

- The testapperror.p procedure loops twice through a REPEAT block containing two CATCH blocks for trapping PGAAppError 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 PGAAppError 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.

Copyright © 2017
Paul Guggenheim & Associates

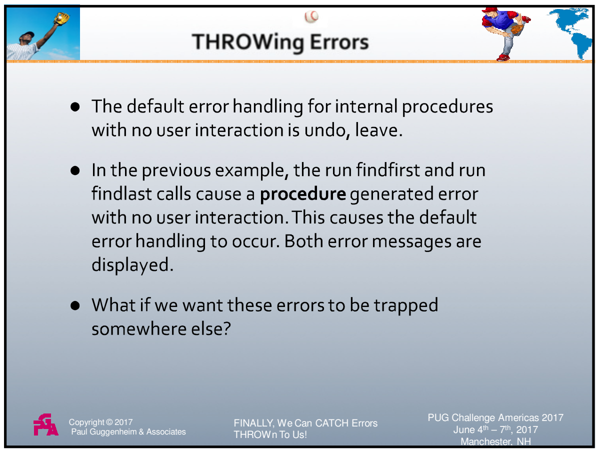
FINALLY, We Can CATCH Errors THROWn To Us!

PUG Challenge Americas 2017
June 4th – 7th, 2017
Manchester, NH



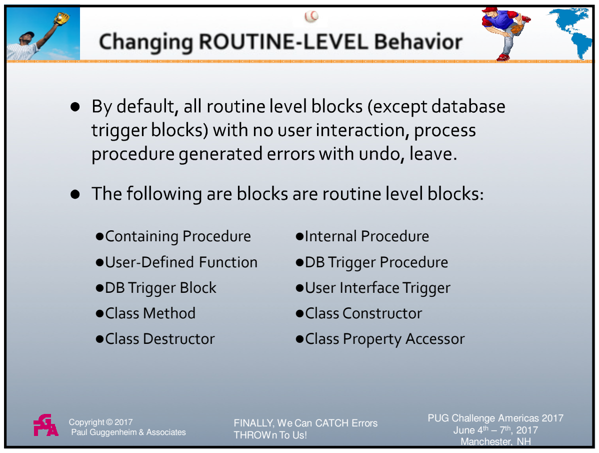
THROWing Errors

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



THROWing Errors

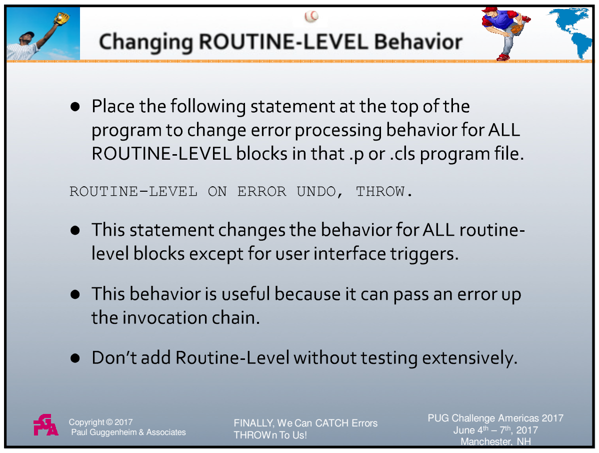
- The default error handling for internal procedures with no user interaction is undo, leave.
- In the previous example, the run findfirst and run findlast calls cause a **procedure** generated error with no user interaction. This causes the default error handling to occur. Both error messages are displayed.
- 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	●Internal Procedure
●User-Defined Function	●DB Trigger Procedure
●DB Trigger Block	●User Interface Trigger
●Class Method	●Class Constructor
●Class Destructor	●Class Property Accessor

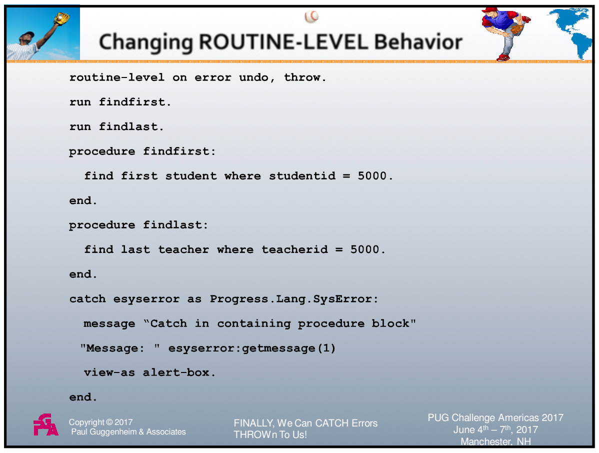


Changing ROUTINE-LEVEL Behavior

- Place the following statement at the top of the program to change error processing behavior for ALL ROUTINE-LEVEL blocks in that .p or .cls program file.

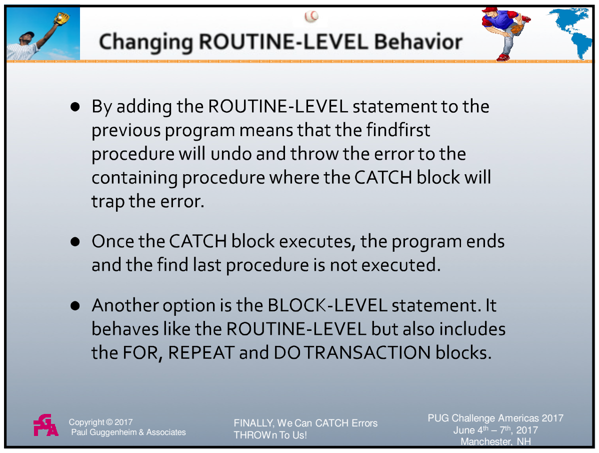
```
ROUTINE-LEVEL ON ERROR UNDO, THROW.
```

- This statement changes the behavior for ALL routine-level blocks except for user interface triggers.
- This behavior is useful because it can pass an error up the invocation chain.
- Don't add Routine-Level without testing extensively.



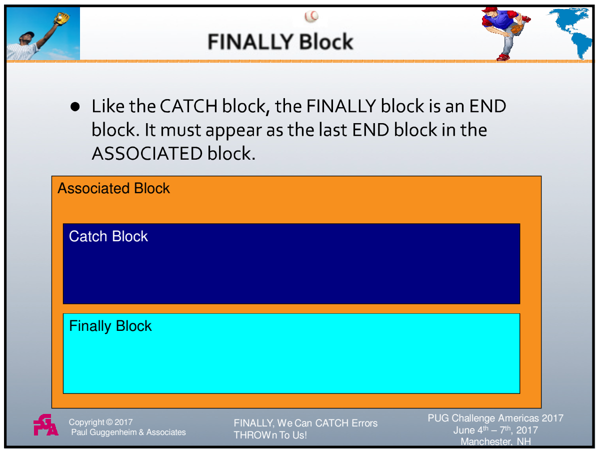
Changing ROUTINE-LEVEL Behavior

```
routine-level on error undo, throw.
run findfirst.
run findlast.
procedure findfirst:
  find first student where studentid = 5000.
end.
procedure findlast:
  find last teacher where teacherid = 5000.
end.
catch esyserror as Progress.Lang.SysError:
  message “Catch in containing procedure block"
 "Message: " esyserror:getmessage(1)
  view-as alert-box.
end.
```



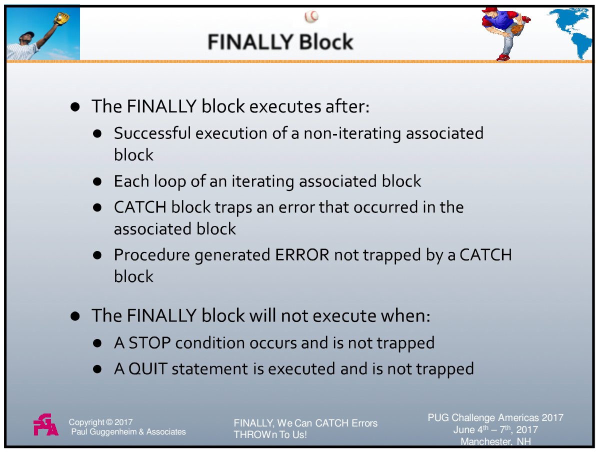
Changing ROUTINE-LEVEL Behavior

- By adding the ROUTINE-LEVEL statement to the previous program means that the findfirst procedure will undo and throw the error to the containing procedure where the CATCH block will trap the error.
- Once the CATCH block executes, the program ends and the find last procedure is not executed.
- Another option is the BLOCK-LEVEL statement. It behaves like the ROUTINE-LEVEL but also includes the FOR, REPEAT and DO TRANSACTION blocks.



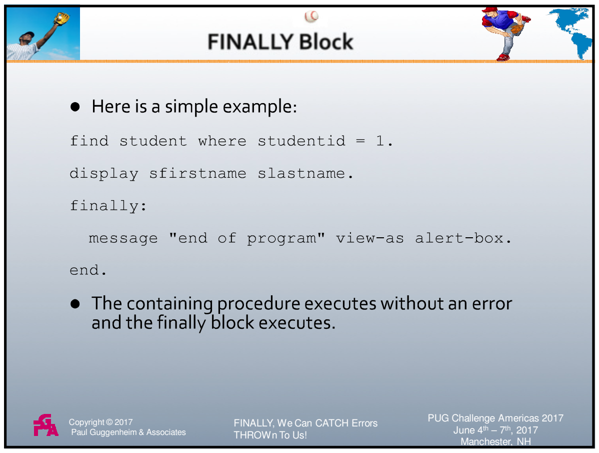
FINALLY Block

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



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:

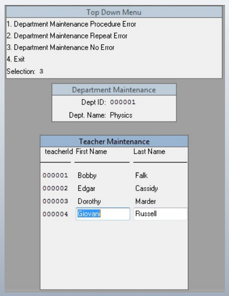
```
find student where studentid = 1.
display sfirstname slastname.
finally:
  message "end of program" view-as alert-box.
end.
```

- 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 maintainablity.
- Two environments will be demonstrated:
  - Top Down Programming
  - Event Driven Programming

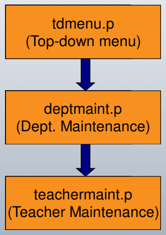
Top-Down Menu

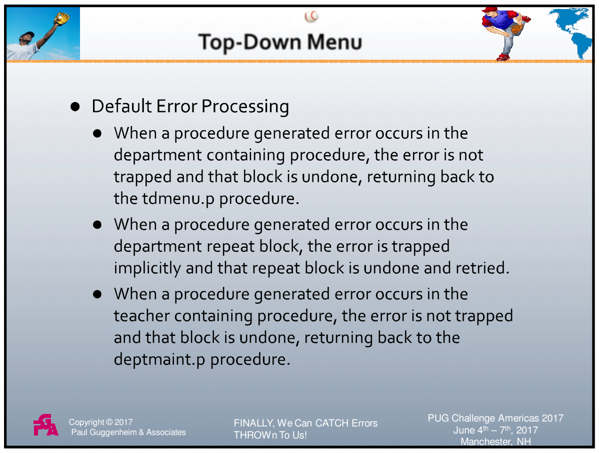


```
Top Down Menu
1. Department Maintenance Procedure Error
2. Department Maintenance Repeat Error
3. Department Maintenance No Error
4. Exit
Selection:  3

Department Maintenance
   Dept ID:  000001
Dept. Name:  Physics

Teacher Maintenance
teacherId  First Name  Last Name

000001     Bobby       Falk
000002     Edgar       Cassidy
000003     Dorothy     Marder
000004     Giovani     Russell
```

SEH Best Practices

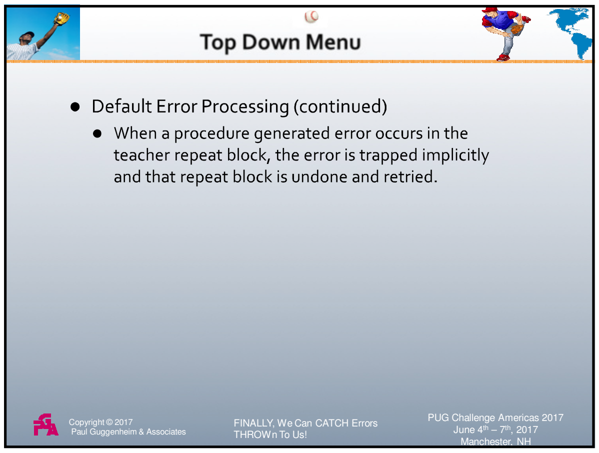
- The following are the programs in a top down example:





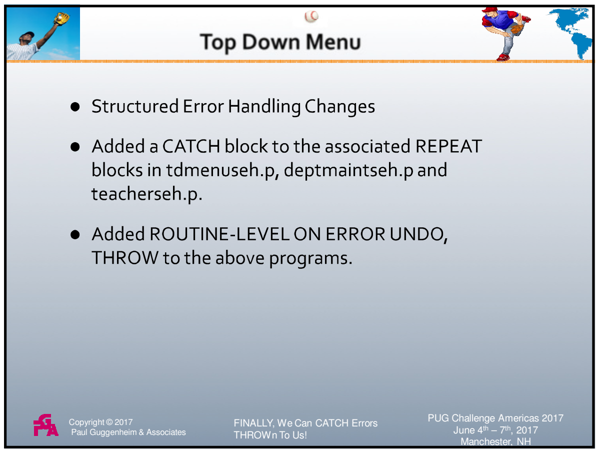
Top-Down Menu

- 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 deptmaint.p procedure.



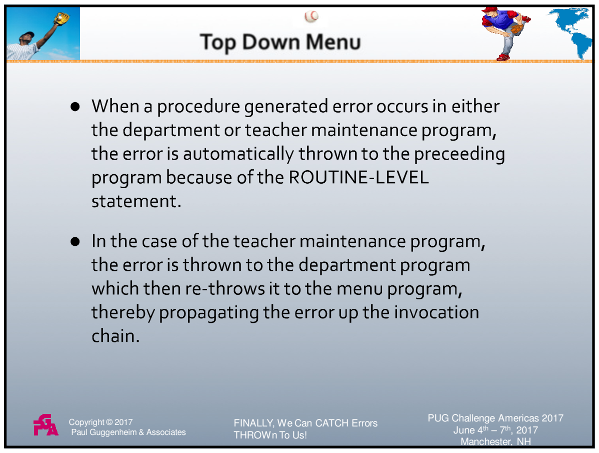
Top Down Menu

- 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.



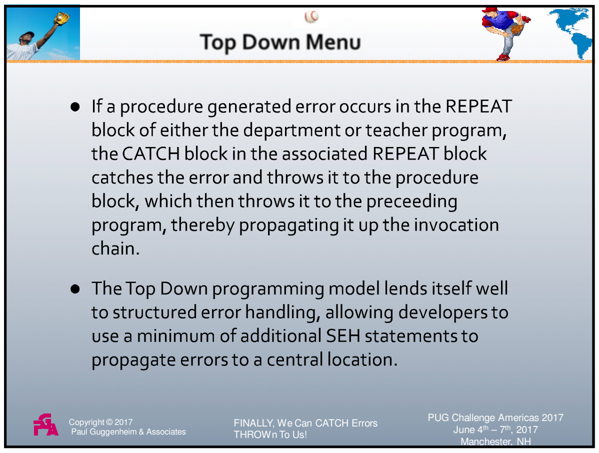
Top Down Menu

- Structured Error Handling Changes
- Added a CATCH block to the associated REPEAT blocks in tdmenuseh.p, deptmaintseh.p and teacherseh.p.
- Added ROUTINE-LEVEL ON ERROR UNDO, THROW to the above programs.



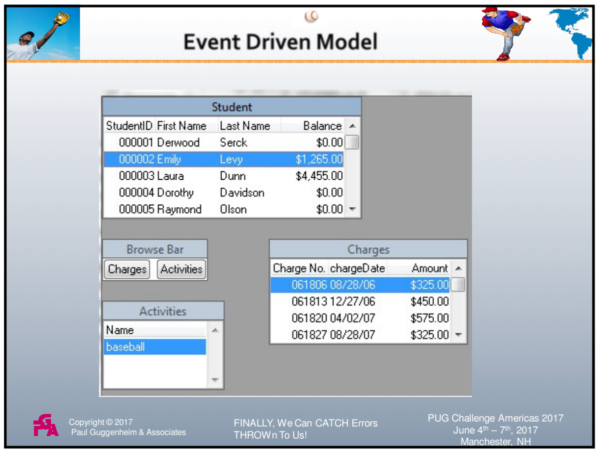
Top Down Menu

- When a procedure generated error occurs in either the department or teacher maintenance program, the error is automatically thrown to the preceeding program because of the ROUTINE-LEVEL statement.
- In the case of the teacher maintenance program, the error is thrown to the department program which then re-throws it to the menu program, thereby propagating the error up the invocation chain.



Top Down Menu

- If a procedure generated error occurs in the REPEAT block of either the department or teacher program, the CATCH block in the associated REPEAT block catches the error and throws it to the procedure block, which then throws it to the preceeding program, thereby propagating it up the invocation chain.
- The Top Down programming model lends itself well to structured error handling, allowing developers to use a minimum of additional SEH statements to propagate errors to a central location.



Event Driven Model

Student

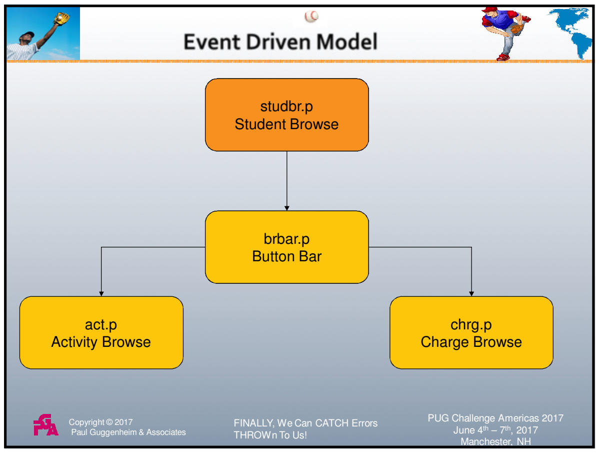
StudentID	First Name	Last Name	Balance
000001	Derwood	Serck	$0.00
000002	Emily	Levy	$1,265.00
000003	Laura	Dunn	$4,455.00
000004	Dorothy	Davidson	$0.00
000005	Raymond	Olson	$0.00

Charges

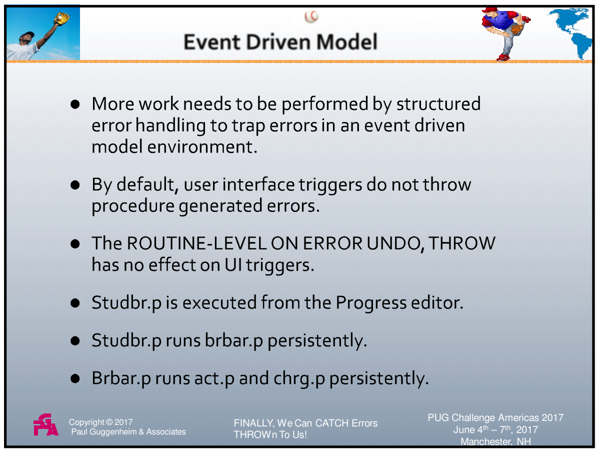
Charge No.	chargeDate	Amount
061806	08/28/06	$325.00
061813	12/27/06	$450.00
061820	04/02/07	$575.00
061827	08/28/07	$325.00

Activities

Name
baseball

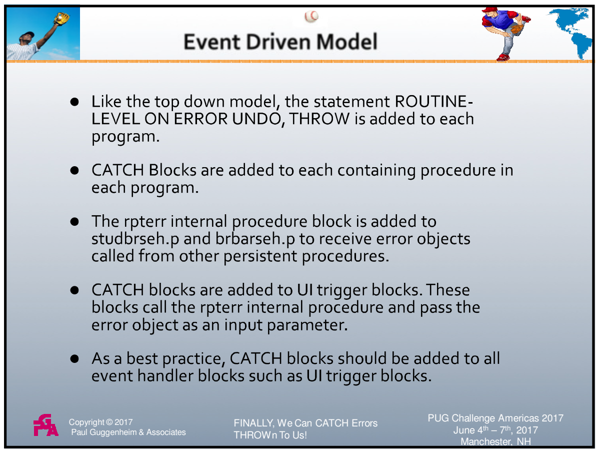


Event Driven Model



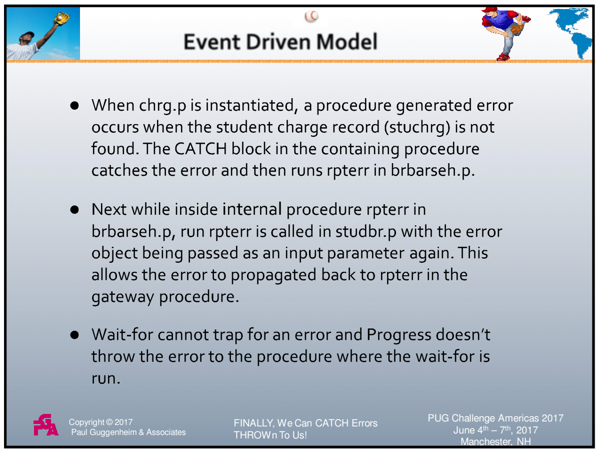
Event Driven Model

- 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 chrg.p persistently.



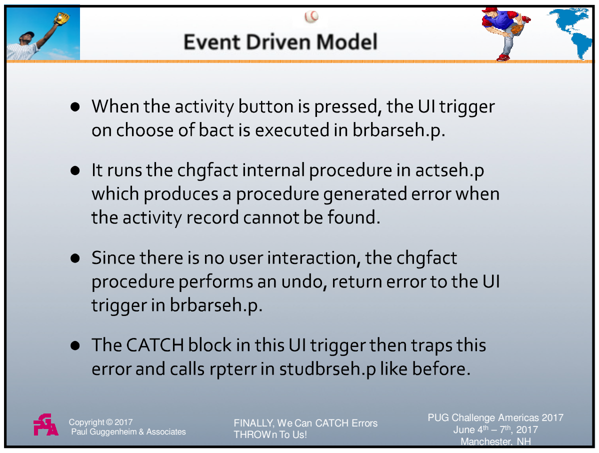
Event Driven Model

- 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 studbrseh.p and brbarseh.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.



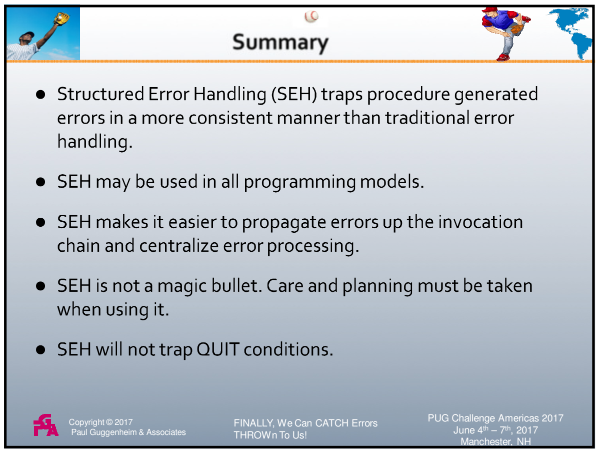
Event Driven Model

- 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 brbarseh.p.
- Next while inside internal procedure rpterr in brbarseh.p, run rpterr is called in studbr.p with the error object being passed as an input parameter again. This allows the error to 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.



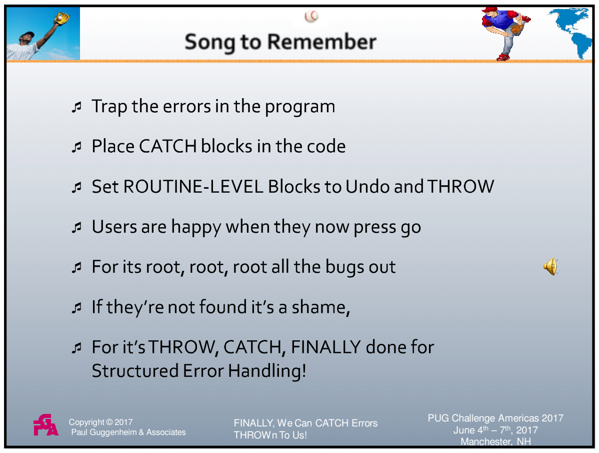
Event Driven Model

- When the activity button is pressed, the UI trigger on choose of bact is executed in brbarseh.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 brbarseh.p.
- The CATCH block in this UI trigger then traps this error and calls rpterr in studbrseh.p like before.



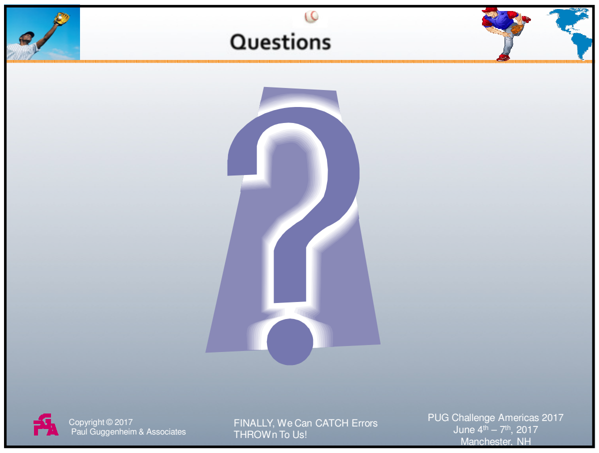
Summary

- 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.



Song to Remember

- Trap the errors in the program
- Place CATCH blocks in the code
- Set ROUTINE-LEVEL Blocks to Undo and THROW
- Users are happy when they now press go
- For its root, root, root all the bugs out
- If they're not found it's a shame,
- For it's THROW, CATCH, FINALLY done for Structured Error Handling!



Questions