



IBM Software Group

## Exceptions!

Users can't live with 'em

Programmers can't live without 'em

Barbara Morris  
IBM



# Agenda

- Why exceptions are good for you (yes, they are)
- Exception handling that is available for RPG programmers
- Getting information about exceptions
- Creating your own exceptions



Exceptions are good for you ... in their place

Exceptions are good for you!

Pardon?

But they need careful handling because ...

Exceptions are **not good** for your users!

Keep them to yourself!



# Exceptions are good for you ... in their place

Exceptions are like pain. Not nice in themselves, but the alternative is much worse.

- If you are running across hot coals, and there is no pain in your feet, you might continue to run until you burnt off your feet.

Owww.

- If your program runs into an error situation, and there is no exception, the program might continue to run until your database is completely corrupted.

Double Owww.



## Exceptions are good for you ... in their place

*"I'd rather just check in advance instead of having to deal with exceptions."*

That seems reasonable, but it's hard to check for everything that might go wrong.

- If you know you might run into hot coals, you can stop regularly and look at what you are running over. But what if you get hit by lightning?
- If you know that you might have an index too big for an array, you can check before you use the index. But what if you have a pointer not set?



# The Dreaded Inquiry Message

If you don't do something to handle exceptions, you risk that a user might see something that you don't want them to see.

**Procedure AR01WX2 in program UTIL/AR01WX2 attempted to divide by zero (C G D F)**

User: (*thinks*) "G probably means Go Ahead, so I'll try that"

*... time passes ...*

User: (*calls helpdesk*) "I keep getting the same message over and over about divide by zero"

**CEE9901 unmonitored by AR01WX2 at statement 539**

User: (*calls helpdesk*) "What does CEE9901 mean?"



## Avoid inquiry messages by using exception handling

An exception handler can **sometimes** correct the problem so the program can keep running.

If it can't do that, it can log information about the error to help with later problem determination, and inform the user that a problem occurred in such a way that the user knows exactly what to do next:

We're sorry. The program could not complete your request. We have forwarded information about your problem to the system administrator. You do not have to take any further action.



# RPG exception handling

- You can handle errors on one statement or many statements using one of RPG's exception handlers
  - ▶ (E) extender (*or error indicator*)
  - ▶ MONITOR
  - ▶ \*PSSR, INFSR subroutines
- You can handle errors using a procedure that you *register* and *unregister* with the system using an API
  - ▶ CEEHDLR
  - ▶ CEEHDLU
- For a cycle-main procedure, RPG also has a function-check handler
  - ▶ Default handler



## (E) extender

The (E) extender, like the error indicator, can handle exceptions for one statement.

```
chain(e) keys(i) record result;
```

Following a statement with (E), you code your error handling:

```
chain(e) keys(i) record result;  
if %error();  
    // handle the error  
endif;
```



## (E) Extender - notes

- The (e) extender says to
  - ▶ set %ERROR off
  - ▶ if an error occurs on that statement, set %ERROR on and continue with the next statement



# BUT, does (E) always do what we expect?

## What happens if

1. the file is not open?
2. there is bad decimal data in one of the input fields?
3. the index  $i$  is zero?

The (E) extender and the error indicator only handle "opcode-specific" errors.



## (E) notes

- The (E) extender and the error indicator only handle "opcode-specific" errors. For I/O opcodes, (E) handles file-related errors, but it doesn't handle errors such as array index errors on search arguments, or errors in input or output specs. For string opcodes, (E) handle length and start position errors, but not array index or decimal data errors.
- This is desirable behaviour. File errors are different in nature from program errors such as array index or decimal data. If you are coding (E) for a file I/O opcode, you probably don't want your file-error handling code to get mixed up trying to handle index-out-of-bounds error.



## (E) and %ERROR clutter

(E) is great for handling "expected" errors on one statement

But checking %ERROR after every (E) statement can clutter up your code.

```
open(e) myfile;
```

```
  if %error;
```

```
    --- handle error
```

```
  endif;
```

```
  i = 1;
```

```
  read(e) myfile  
res(i);
```

```
dow not %eof and not %error;
```

```
  if %error;
```

```
    --- handle error
```

```
  endif;
```

```
  --- process record
```

```
  i += 1;
```

```
  read(e) myfile res(i);
```

```
enddo;
```



# Use MONITOR to handle several statements

All the code between MONITOR and ON-ERROR is monitored.

```
monitor;  
    open myfile;  
    i = 1;  
    read myfile result(i);  
    dow not %eof;  
        --- process the record ---  
        i += 1;  
        read myfile result;  
    endif;  
  
on-error;  
    --- handle the error ---  
endmon;
```



# MONITOR handles all status codes

MONITOR can handle opcode-specific errors AND general program or file errors

```
monitor;
  --- code that might have an error
on-error stat_open_error; // constant = 01217
  --- handle file-open error
on-error *FILE;
  --- handle other file errors
on-error; // catch-all
  --- handle all other errors
endmon;
```



# MONITOR - notes

Mix (E) and MONITOR. (E) handles expected errors; MONITOR handles unexpected errors.

```
monitor;  
  open(e) myfile;  
  if %error;  
    myfileName = findMyfile();  
    open myfile;  
  endif;  
  --- read loop  
on-error stat_open_error;  
  --- handle file-open error  
on-error *FILE;  
  --- handle other file error  
on-error;  
  --- handle all other errors  
endmon;
```

The first open assumes the EXTFILE may not be a valid filename. (e) to handle that error, but not others like pointer-not-set.

The second open does not expect any errors, so (e) is not used.



# Error subroutines (deprecated)

## ■ \*PSSR

If there is a subroutine with the name \*PSSR, it will get called any time there is an unhandled **Program** exception (anything other than a File exception).

```
begsr *pssr;  
    --- handle program errors here ---  
endsr;
```

## ■ INFSR

```
Fmyfile if e disk INFSR(myfileHandler)  
  
begsr myfileHandler;  
    --- handle errors for MYFILE here ---  
endsr;
```



## \*PSSR to handle file errors too

If you want your \*PSSR to also act as a file error handler, code  
INFSR(\*PSSR)

```
Fmyfile  if  e      disk      INFSR(*PSSR)
Fqsysprt  o  f  132  printer  INFSR(*PSSR)

  begsr *pssr;
    --- handle program and file errors here ---
  endsr;
```



## \*PSSR - notes

A common type of error subroutine is one that just returns from the program or procedure, possibly logging some information about the error.

```
begsr *pssr;  
    cmd = 'DSPJOBLOG OUTPUT(*PRINT)';  
    callp(e) qcmdexc(cmd : %len(cmd));  
    return; // return from procedure  
endsr;
```

The RETURN operation caused the program or procedure to end normally. That may be what you want, if you were able to recover from all errors.

If you want to let your caller to know that all was not well, don't code the RETURN operation. Instead, let the \*PSSR get to its ENDSR opcode.



# ENDSR for an error subroutine

- For a main procedure, ENDSR can have a value in Factor 2 that indicates what should get control next:
  - ▶ \*CANCL sends an exception message to the caller of the main procedure
  - ▶ Blank or no Factor 2 invokes the default RPG handler (more about that later)
  - ▶ Any other value goes to the specified point in the RPG cycle. These values only make sense if there is a primary file, where you want to continue processing records even if there is an error with one record.
- For a subprocedure, reaching ENDSR of an error subroutine always causes the procedure to end abnormally.



# Exceptions within an error subroutine

**Never allow an unhandled exception  
in an error subroutine.**

**It can cause infinite looping.**

If an error occurs while an error subroutine is running, the error subroutine will get called again --- and if the error occurs again, it will get called again and again ...



# Avoid \*PSSR exceptions – notes

To ensure that the \*PSSR can't fail with an exception, use MONITOR to monitor all code. If you need to do something in the ON-ERROR, put the whole MONITOR inside another MONITOR.

```
begsr *pssr;  
  monitor;  
    monitor;  
      --- do stuff  
    on-error;  
      --- handle error  
    endmon;  
  on-error;  
    // Warning: Don't code anything in this outer  
    // ON-ERROR. This subroutine must handle all  
    // its exceptions.  
  endmon;  
endsr;
```



# Choosing between RPG's handlers

## Choose (E)

- ▶ If the errors for that particular statement have their own specific error handling. For example, an OPEN error where you will create or locate the file in the error handling.

## Choose \*PSSR

- ▶ If you don't expect any errors in your normal processing

**AND** you don't want to continue after handling an error

**AND** your error handling is so trivial that it is not possible for it to have any errors.

## Choose MONITOR

- ▶ If you have several statements that have the same exception handling
- ▶ If you have complex error handling
- ▶ If you want to continue after handling the error
- ▶ If there are some errors you want to handle and others you want to percolate



# CEEHDLR – notes 1

The CEEHDLR API lets you register a procedure that will get called if an exception occurs. The CEEHDLU API lets you unregister it.

CEEHDLR exception handlers are outside of RPG's control. The system calls your exception handler without RPG's exception handlers being aware of it.

RPG's (E) and MONITOR exception handlers have priority. If an exception would get handled by (E) or MONITOR, the CEEHDLR handler will not get control. If the exception would get handled by \*PSSR, the CEEHDLR will get control first.

For example, you have a MONITOR with an ON-ERROR for file errors, and you enable a handler with CEEHDLR. If an OPEN error occurs, the CEEHDLR handler will not get called, since the error would be handled by the MONITOR. If an index error occurs, the CEEHDLR handler would get called.



## CEEHDLR – notes 2

Your handler can set the “Action” parameter to indicate what the system should do with the exception.

Here are two things that are **safe** for your handler to do:

- Percolate the error
- Promote the error to a different error



## CEEHDLR – notes 3

Here is something that is **not safe** for your handler to do:  
Continue at the next instruction.

The reason this is not safe is that the next instruction is **not** the next RPG statement. It is the next low-level system instruction which might be somewhere in the middle of the RPG statement. Your handler cannot know exactly where the program would continue; you should assume the worst.

For example, if you allow the program to continue after a decimal data error, the next low-level instruction might be to assign the invalid data to a result field, eventually leading to corruption of your database.

**This is not a far-fetched example. It really happened to a real database.**



# How many handlers do you need?

*“Do I have to add an exception handler to every procedure?”*

No. You need at least one exception handler to be active at all times, if you want to avoid inquiry messages or meaningless messages from the system about your program crashing.

But you don't necessarily need an exception handler for every procedure.

**An exception handler can handle exceptions for itself, and for the procedures that it calls.**



# System exception handling

When something in the system detects a problem, an exception message is sent. The Message Handler component of the system starts looking for an exception handler.

It first looks for an exception handler enabled for the procedure that the exception was sent to.

If no handler is enabled for that procedure, it checks its caller, and works back through the call stack until either

- ▶ It finds a procedure that has an exception handler enabled
- ▶ It hits a *Control Boundary* (a procedure in a different activation group)



# Finding an exception handler

**PGM  
1**

```
a. CALLP (E)  PGM2
b. CALLP      PGM2
```

**PGM  
2**

```
c. CALLP (E)  PGM3
d. CALLP      PGM3
```

**PGM  
3**

```
        MONITOR
e:      x = y / z
        ON-ERROR
        ENDMON
f:      x = y / z
```

**Case 1: Statement e.**

The MONITOR handles it.

*Good!*

**Case 2: Statements c, f.**

The PGM2 (E) handles it.

*Good!*

**Case 3: Statements a, d, f.**

The PGM1 (E) handles it.

*Good!*

**Case 4: Statements b, d, f.**

No handlers! THREE  
inquiry messages!

*Oh no!*



# System exception handling - notes

- When the system finds an exception handler enabled, it calls the handler. When the handler eventually returns, the handler either *handled* or *percolated* the exception.
  - ▶ If it handled the exception, then the system immediately terminates all the other procedures that didn't have exception handlers enabled, and control branches to wherever the handler indicated.
  - ▶ If it didn't handle the exception, then the system continues searching for another exception handler.
- If the search reaches a control boundary, the system creates a *function check* (message CEE9901 or CPF9999), and starts searching for a function-check handler, starting with the To-procedure of the original message.
- If the procedure it checks has a function-check handler, that handler gets called. The handler can either handle or percolate the exception, as before. If it percolates the exception, the procedure gets removed from the program stack.
- If the procedure it checks does not have a function-check handler, the procedure gets removed from the program stack.
- If the system doesn't find a function-check handler by the time it reaches the control boundary, it sends the CEE9901 or CPF9999 to the caller of the procedure at the control boundary.



# Get information about exceptions

*“What went wrong?”*

- %STATUS has the RPG status code for the most recent error
- The RPG Program Status Data Structure (PSDS) and File Information Data Structures (INFDS) have some information about errors. The PSDS has the message ID and the first 80 bytes of the message itself.
- You can get a formatted dump of all the variables in the current module by using the DUMP opcode.
- The joblog and job information are useful to determine more about unexpected errors. You can capture this information in your exception handlers by calling a CL program or QCMDEXC to do these commands:
  - ▶ DSPJOBLOG OUTPUT(\*PRINT)
  - ▶ DSPJOB OUTPUT(\*PRINT)
- Your program can work with messages in the joblog by using CL programs or by calling system APIs.



# Create your own exceptions

If you have a procedure that returns customer information for a particular customer name, you have to consider the possibility that the customer name is not in the customer file.

You have a few options:

1. Define your procedure to have a return code that indicates “found” or “not found”. Callers will check the return code to see if they should use the customer information.
2. Send an exception (*use CL SNDPGMMSG or API QMHSNDPM*).
3. Allow the caller to choose whether to get a return code or an exception.



# Create your own exceptions – notes 1

- If you decide to use a return code, it is possible that some callers will forget to check the return code after they call your procedure.

```
rc = getBalance (cust_id : cust_info);  
if rc = rc_OK;
```

```
getBalance (cust_id : cust_info);
```

- By having your procedure send an exception, you can avoid the problem in the second call above, where the caller ignored the return code. Now, the problem will not go unnoticed in either case.

```
callp(e) getBalance (cust_id : cust_info);  
if not %error;
```

```
getBalance (cust_id : cust_info);
```



## Create your own exceptions – notes 2

- You could design your procedure to allow the caller to choose whether to receive error information as a return code or as an exception. One way to do this is to use an optional parameter.
- This is a useful technique if you expect the error situation to occur frequently.
  - ▶ Using return codes to indicate problems is usually faster than using exceptions
  - ▶ Using return codes helps keeps the joblog tidy
- But design it so that if the caller is careless, they get an exception. Make them code something extra to get the return code.

```
getBalance (cust_id : cust_info);  
rc = getBalance (cust_id : cust_info : OPT_USE_RC);
```



# Recommendations

## Top-level program of an application:

- ▶ Use CALLP(E) or MONMSG, and if an error is trapped by the handler, send a message, or put up a screen, or send an e-mail, or do something to indicate that an error occurred.

## Other programs and procedures in the application:

- ▶ Use exception handlers to trap local exceptions if either
  - It is possible to correct the problem and continue running
  - There is some specific error-logging that can't be done at the top-level of the application
- ▶ Percolate exceptions if there is no need to handle them locally
- ▶ Use your own exceptions to indicate that an error or problem was detected

## General:

- ▶ Standardize your exception handling by writing some routines that handle some of the drudgery such as sending exceptions and receiving messages from the joblog.
- ▶ Have your unexpected-exception handlers call the same procedure (in a service-program) rather than duplicating the same logic over and over.



# Bibliography

- ILE Concepts
  - ▶ General information on ILE exception handling
- ILE RPG Reference **and** ILE RPG Programmer's Guide
  - ▶ Information on RPG exception handling, status codes, PSDS, INFDS
- IBM Info Center
  - ▶ Information on APIs that can be used to send exception messages and to work with messages in the joblog



# Thank YOU

[→ Go to IBM](#)

© Copyright IBM Corporation 2009. All rights reserved.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

IBM, the IBM logo, the on-demand business logo, Rational, the Rational logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

