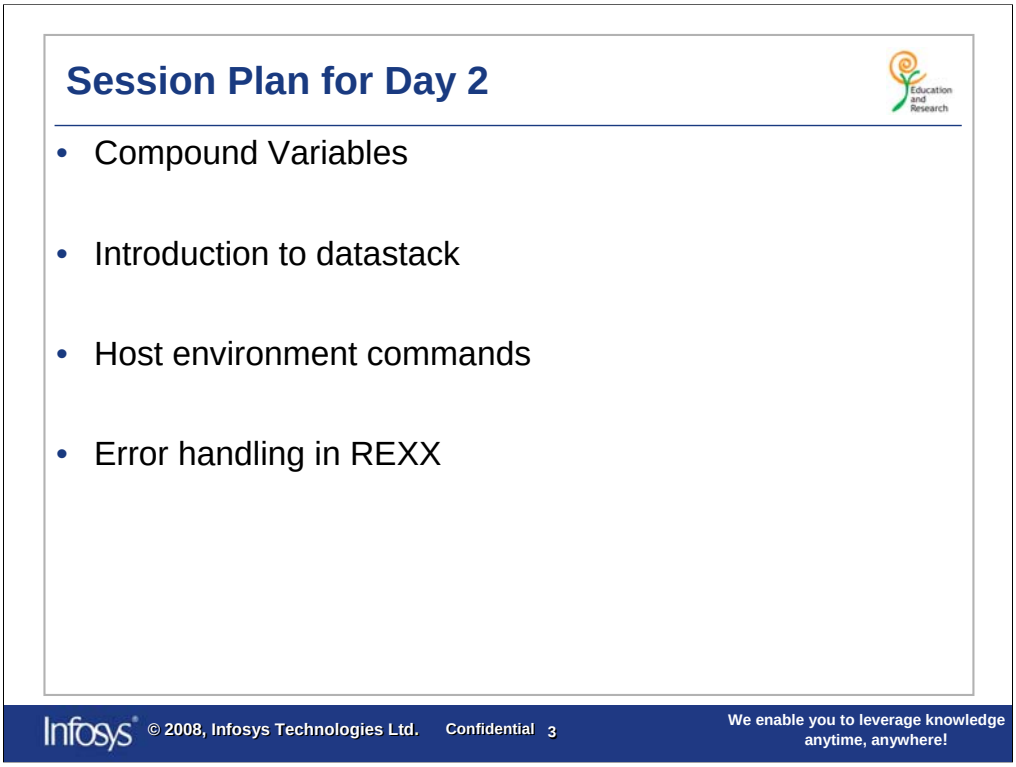
Hello and welcome to a 2-day course on REXX-TSO Programming!!!

Confidential Information

- *This Document is confidential to Infosys Technologies Limited. This document contains information and data that Infosys considers confidential and proprietary ("Confidential Information").*
- *Confidential Information includes, but is not limited to, the following:*
  - *Corporate and Infrastructure information about Infosys;*
  - *Infosys' project management and quality processes;*
  - *Project experiences provided included as illustrative case studies.*
- *Any disclosure of Confidential Information to, or use of it by a third party, will be damaging to Infosys.*
- *Ownership of all Infosys Confidential Information, no matter in what media it resides, remains with Infosys.*
- *Confidential information in this document shall not be disclosed, duplicated or used – in whole or in part – for any purpose without specific written permission of an authorized representative of Infosys.*



Session Plan for Day 2 - Notes

Upon completion of Day 2 of this course, you will gain an understanding of the topics that are listed here.

Compound Variables (1 of 12)

- Many a times, it is useful to store all related data in a way, such that it can be easily accessed.
- For example: Names of all the students in a school can be stored in an array & retrieved by number

STUDENT No	NAME
1	John
2	Bill
3	Lucy

- In REXX, compound variables are used to create a one-dimensional array or list of variables

Compound Variables (1 of 12) - Notes

We may often require to process lists or collections of related data. It would be inconvenient if we had to store each data item to be processed in a separate variable. Instead it would be better if we could refer to the entire set of data items using one common name . In REXX , this is possible through the use of compound variables.

A compound variable is a variable the consists of at least one period(.) with at least one character on either side of it. E.g. WEEK.DAY, ARRAY.J.K etc.

Compound Variables (2 of 12)

- A compound variable is a variable that has at least one period with characters on both sides of it
- Ex: Week.Temperature, Student.Name, Row.1
- **Rules for forming compound variables**:
  - Compound variable names cannot start with a digit or with a period
  - If there is only one period , it may not be the last character
- Invalid names: .NAME.I , 9.DATE, MAT.

Compound Variables (2 of 12) - Notes

We now look at some rules for creating valid compound variable names. Compound variable names cannot start with a digit or a period (.) E.g. 7.DAY , .NAME etc are invalid compound variable names. Also if there is only one period, it may not be the last character . E.g. NAME. is also an invalid compound variable name.

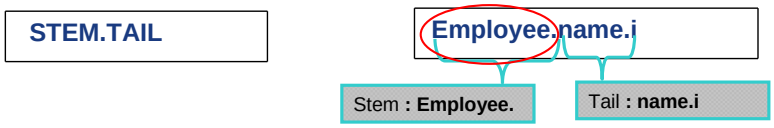
Compound Variables (3 of 12)

- A compound variable has two parts:
  - **Stem**:

    A stem is that part of a compound variable name, up to & including the first period
  - **Tail:**

    The part of the compound variable name excluding the stem is called the tail

- Format:



Compound Variables (3 of 12) - Notes

The format of a compound variable is STEM.TAIL .

That part of a compound variable name upto and including the first period (.) is known as the stem followed by the tail part of the variable name delimited by periods.

E.g. AGE.NAME.J : Here AGE. is the stem and NAME.J is the tail.

ARRAY.J.K : Here ARRAY. is the stem and J.K is the tail

Compound Variables (4 of 12)

- Unlike in arrays, in compound variables, subscripts need not be numeric
  - Ex: day = Tuesday; Week.day = 'Week 52'
- A compound variable should contain at least one period with at least one character on both sides of it
  - Ex: **name.** (invalid) ; **name.i** (valid)
- In a compound variable, all variables except the (STEM) will take on previously assigned values.
- The STEM will be the upper case equivalent of itself

Compound Variables (4 of 12) – Notes

Some facts:

Unlike in arrays, in compound variables, subscripts need not be numeric

Ex: day = Tuesday;

Week.day = 'Week 52'

A compound variable should contain at least one period with at least one character on both sides of it

Ex: name. (invalid). It should be noted that this is an invalid compound variable name but it is a valid stem name.

Ex: name.i (valid)

In a compound variable, all variables except the first variable (STEM) will take on previously assigned values.

The STEM will be the upper case equivalent of itself

Compound Variables (5 of 12)

- During substitution, lower-case values will not be converted to uppercase
- The values of the simple variables substituted in the tail may contain any characters (including periods and blanks)

```
000200  day = 'Thursday'
000300  week = week52
000400  Say 'before: ' week.day
000400  week.day = 'NIL'
000500  Say 'after: ' week.day
```

```
before: WEEK.thursday
after:  NIL
```

```
000600  day = '#&*.'
000700  Say week.day
```

```
WEEK.#&*.
```


Compound Variables (5 of 12) – Notes

Some facts:

During substitution, lower-case values will not be converted to uppercase

The values of the simple variables substituted in the tail may contain any characters (including periods and blanks) .

Compound Variables (6 of 12)

- If the simple variables are not given any value previously, the variables will take on the upper case equivalent of the name.

```
000300  week = week52
000400  Say 'before: ' week.day
000400  week.day = 'NIL'
000500  Say 'after: ' week.day
```

```
before: WEEK.DAY
after:  NIL
```


Compound Variables (6 of 12) – Notes

Some facts

If the simple variables are not given any value previously, the variables will take on the upper case equivalent of the name.

```
000300  week = week52
000400  Say 'before: ' week.day
000400  week.day = 'NIL'
000500  Say 'after: ' week.day
```

In the above example, day is a simple variable in the compound variable week.day. But 'day' has not been initialized. Hence when we display week.day, the output will be WEEK.DAY

Compound Variables (7 of 12)

- A DO loop can be used to setup & initialize an array
  - In this example, an array of 5 names is created using the DO loop

```
000300  DO I = 1 to 5
000400     say 'Enter a name'
000400     pull name.i
000500  END
000600  SAY name.3
```

NIKI

  - Lets assume that the user enters the following names:

    Jack, James, Niki, Kate & Kyle

Compound Variables (7 of 12) – Notes

A DO loop can be used to setup & initialize an array

In this example, an array of 5 names is created using the DO loop

```
000300  DO I = 1 to 5
000400     say 'Enter a name'
000400     pull name.i
000500  END
000600  SAY name.3
```

Lets assume that the user enters the following names:

Jack, James, Niki, Kate & Kyle

The SAY name.3 statement will give 'NIKI' as output.

Compound Variables (8 of 12)

- Usage of STEM:
  - A STEM can be used as a target of an assignment statement.
  - A STEM can be used in an EXECIO statement to read from or write into a dataset
  - A STEM can also be used in OUTTRAP external function to trap the command output
  - More about EXECIO will be covered later

Compound Variables (8 of 12) - Notes

The stem of a compound variable can be the target of an assignment .

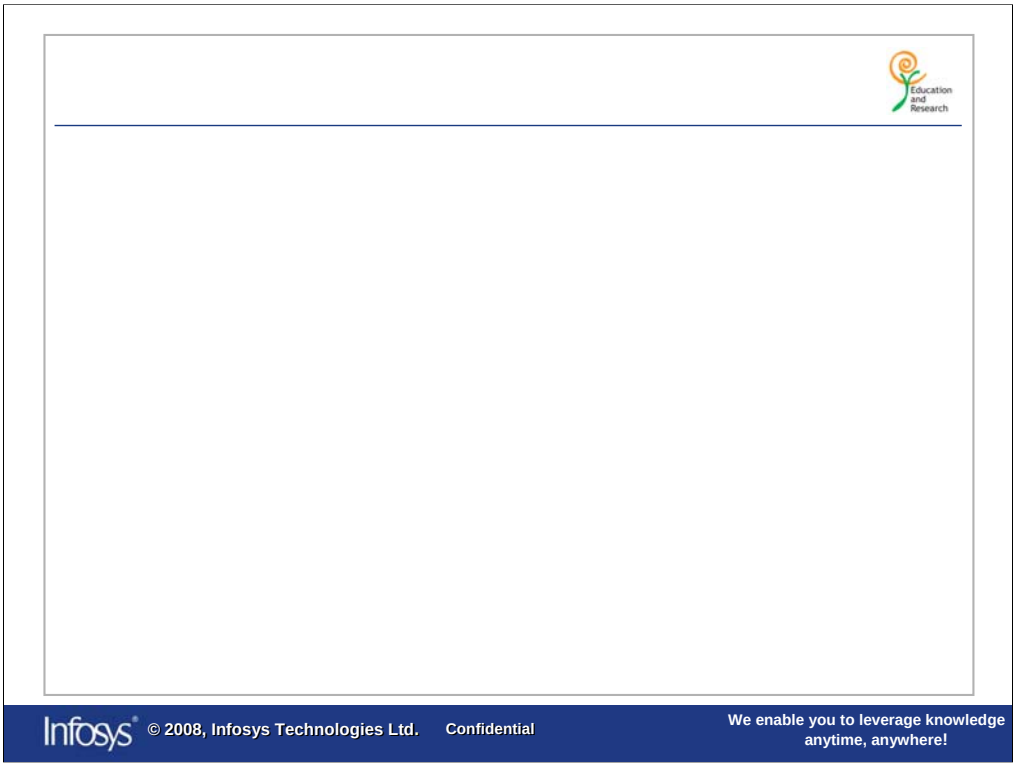
E.g. Total. = 0

After the execution of the above statement, all compound variables whose name starts with the stem 'TOTAL.' will get initialized to 0. If some of these variables had some previous value, then that value will be overwritten.

E.g. Total.BARNEY = 50

```
Total. = 0
say Total.BARNEY
```

The output would be 0



After the assignment (to the stem), whenever a compound variable whose name starts with that stem is referenced, it will return the new value which was assigned to the STEM in the initialization statement. However, this new value will be in effect only until the same stem is initialized to some other value or the individual compound variable is initialized to a new value.

E.g. Total. = 0

Total.BARNEY = 80

say Total.BARNEY

Here the variable Total.BARNEY is assigned a new value 80 . So any reference to this variable will return 80 and not 0 as assigned to the stem in the statement

Total. = 0

Compound Variables (9 of 12)

- An entire array can be initialized to the same value by making the STEM as the target of an assignment statement.
- Any compound variable whose name starts with STEM student. will have the value ‘none’ until another statement overrides it.

```
000300   SUM.J = 80
000400   SUM. = 0
000500   SAY SUM.K  SUM.J  SUM.9
000600   SUM. = 100
000700   SUM.J = 90
000800   SAY SUM.K  SUM.J  SUM.9
```

```
0      0      0
100    90     100
```


Compound Variables (9 of 12) - Notes

An entire array can be initialized to the same value by making the STEM as the target of an assignment statement.

Ex: student. = ‘none’ ; This statement initializes all compound variables starting with STEM student. to ‘none’.

Here initially SUM.j is assigned value 80. Then the assignment SUM. = 0 is done . So the next statement

Say SUM.k SUM.j SUM.9 will print out the values 0 0 0 . This is because when an assignment statement initializes the stem to some value all the compound variables whose names start with that stem will get the new value. This is true even if some of these compound variables contained data previously (the previous content will be over written).

Next we assign a new value 100 to the stem sum. After that the variable SUM.J is assigned 90 .

Now the statement

Say SUM.K SUM.J SUM.9

will print out values 100 90 100

After the assignment (to the stem), whenever a compound variable whose name starts with that stem is referenced, it will return the new value which was assigned to the STEM in the initialization statement. However, this new value will be in effect only until the same stem is initialized to some other value or the individual compound variable is initialized to a new value.

Compound Variables (10 of 12)

•Example 1:

```
COUNTRY = 'INDIA'
STATE = 'Karnataka'
HOME = 'Indiranagar'
SAY HOME.COUNTRY.STATE.city
```

HOME.INDIA.Karnataka.CITY

•Example 2:

```
000300  rate. = 0
000400  null = ''
000500  rate.null = rate.null + 10
000600  say rate. rate.null rate.G
```

0 10 0

Compound Variables (10 of 12) - Notes

In the code snippet shown above we can see that, at the time of reference, the stem is the upper case equivalent of itself. The remaining variables in the tail are substituted with their values .Also please note that since the simple variable city in the tail has not been initialized , at the time of reference , it is replaced by it's upper case equivalent.

Example 2:As seen in this example, the derived name of a compound variable maybe same as the stem . However assigning a value to such a compound variable is not the same as assigning a value to an entire collection of variables using the stem.

In the code snippet shown here , we see that the value of the stem rate. and the variable rate.G remains 0 as initialized in the beginning . It is not affected by the statement

rate.null = rate.null + 10

Compound Variables (11 of 12)

- Some more examples

```
a = 9              /*assigns 9 to variable A */
p = 6              /*assigns 6 to variable P */
q = 8              /*assigns 8 to variable Q */
r = 'Jack'         /*assigns Jack to variable R */
a.p = 20           /*assigns 20 to variable A.6 */
c.r = 'John'       /*assigns John to variable C.Jack */
a.john = 3         /* assigns 3 to variable A.JOHN */
```


Compound Variables (11 of 12) - Notes

Here are some examples of assignment of values to different variables (both simple and compound). Note that in the case of compound variables , as discussed earlier , at the time of reference, the stem is the upper case equivalent of itself and any simple variables in the tail are substituted by their values . These values can be any character(s) including blanks and periods(.) It is also to be noted that during substitution , lower case will not be converted to upper case.

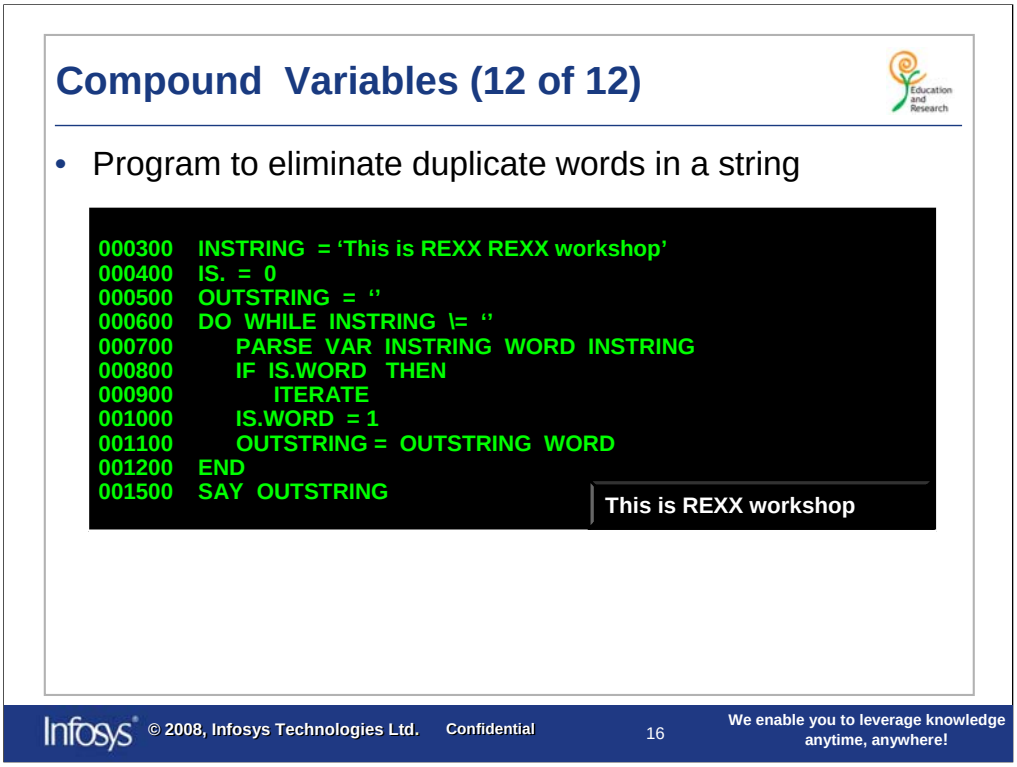
E.g.

c = 'Mark'

r = 'Jack'

c.r = 'John'

will assign John (NOT JOHN) to C.Jack (not to C.JACK nor to Mark.Jack)



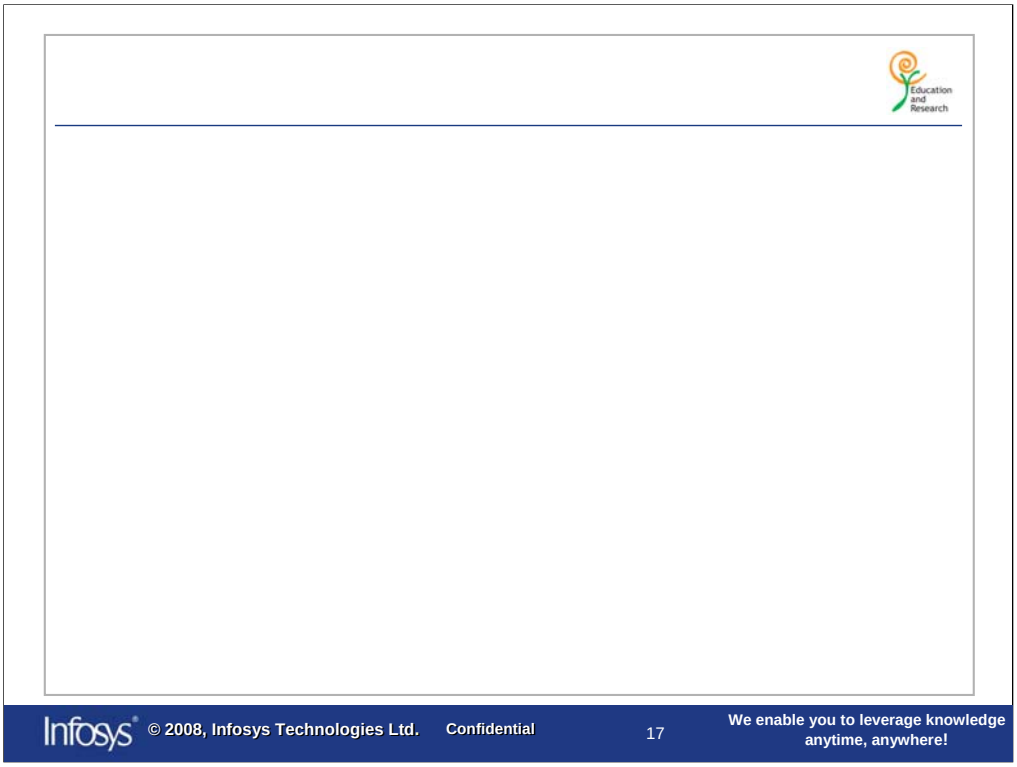
Compound Variables (12 of 12) - Notes

The code snippet given above takes as input a string of word(s) . It removes any duplication and prints the string without duplication.

We first take as input the string of words from where duplication has to be removed. We then initialize the stem of a compound variable IS. to 0 . We will use this later to keep track of which words in the input string are duplicated and which are not.

We then initialize the output string (which we will be printing) to null at first.

We then set up a loop which will break the input string into the first word and the remainder till there is no more input (i.e. no more word left in the input string) . During the loop body , we first check if the first word (just separated from the input string) has been encountered before . This is where the compound variable (and the fact that the subscripts need not be numeric) comes into play . If IS.WORD = 1 (here the value of the simple variable WORD in the tail is the value of the first word separated from the input string . The rest of the string continues to be stored in the variable INSTRING .) , it means that word (separated from the input string and stored in the variable WORD) has been encountered before and so the ITERATE instruction starts the next iteration of the loop . Otherwise if the word (separated from the input string) has been encountered for the first time, the compound variable IS.WORD is set to 1 . This will help us remember that this particular word has been encountered before .



Next the word is added to the output string OUTSTRING by virtue of the concatenation operator blank (as discussed earlier) .

Finally after the loop is exited , the final output list (without duplication) is printed.

E.g. Suppose our input string is HELLO HELLO HI

In the first iteration of the loop, HELLO will be stored in the variable WORD after the instruction

PARSE VAR INSTRING WORD INSTRING

Now IS.HELLO will be 0 (since we have initialized IS. as 0) .

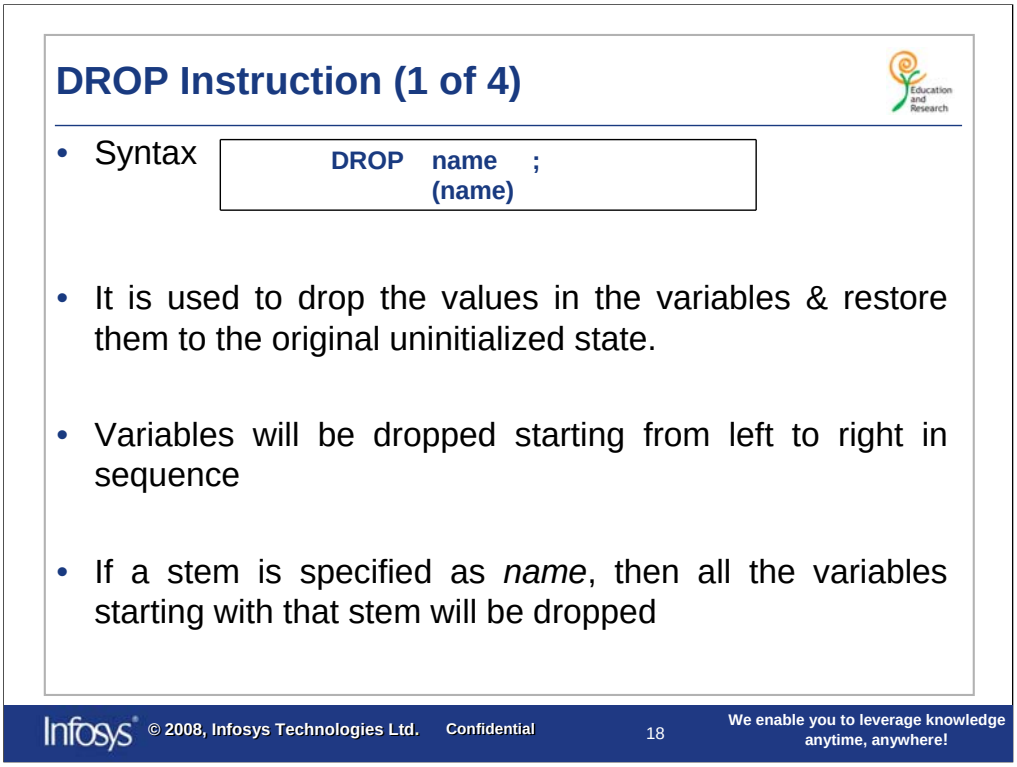
So now IS.HELLO will be set to 1 . Then HELLO will be added to OUTSTRING and the next iteration of the loop starts.

This time INSTRING contains the value HELLO HI

Now after the PARSE instruction , the variable WORD will contain the value HELLO once again.

This time however IS.HELLO is 1 , so the rest of the statements in the loop body are skipped (i.e. HELLO is not added to the output list OUTSTRING) and the next iteration starts.

Now, let us see how to uninitialize simple & compound variables using DROP instruction.



DROP Instruction(1 of 4) – Notes

Syntax: **DROP** ***name***;

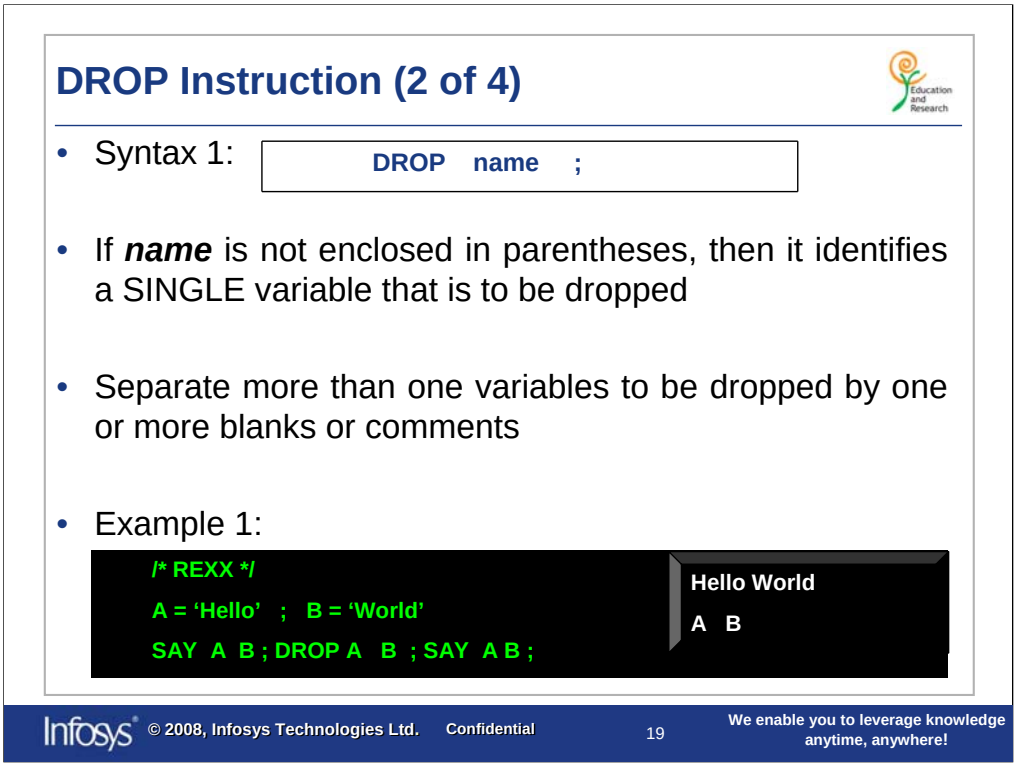
(name)

It is used to drop the values in the variables & restore them to the original uninitialized state.

Variables will be dropped starting from left to right in sequence.

If the same variable is specified more than once in a DROP instruction, it will not cause an error. No error will occur if an unknown variable is specified in the DROP instruction.

If a stem is specified as *name*, then all the variables starting with that stem will be dropped.



DROP Instruction(2 of 4) – Notes

Syntax 1: **DROP** ***name*** ***;***

If ***name*** is not enclosed in parentheses, then it identifies a variable that is to be dropped. The ***name*** specified must be a valid variable name.

If more than one variable is to be dropped in the same instruction, then names should be separated by one or more blanks or comments.

Example 1 :

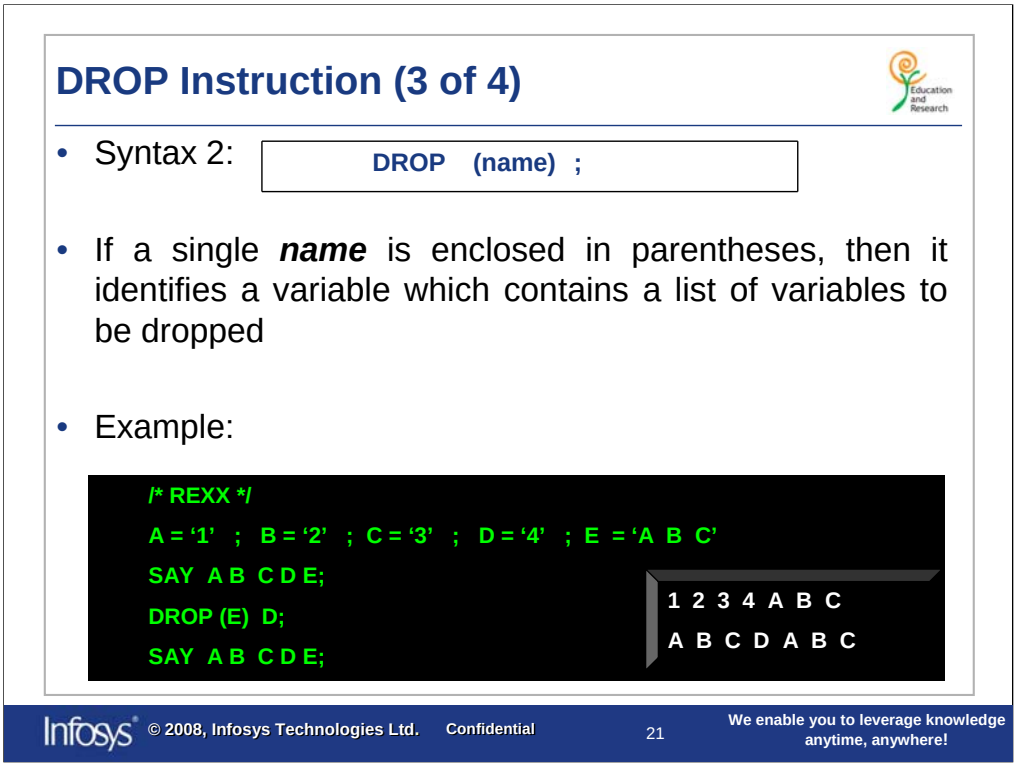
```
A = 'Hello'  ;  B = 'World'
SAY  A B ;
DROP A   B  ;
SAY  A B ;
EXIT
```

In the above example, the DROP instruction will drop the contents of variables A & B. Note that A & B must be separated by at least one blank or a comment. If there is no space & the instruction is written as

DROP AB, then the instruction will try to drop a variable by name **AB.**

Below shown is a deviation of example 1 given in previous slide, where A & B are separated by a comment instead of being separated by spaces & it is valid to do so. Hence, the output will remain same as example 1.

```
A = 'Hello'  ;  B = 'World'
SAY  A B ;
DROP A/*drops B*/B  ;
SAY  A B ;
EXIT
```



DROP Instruction(3 of 4) – Notes

Syntax 2: **DROP** ***(name)***;

If a single ***name*** is enclosed in parentheses, then it identifies a variable which contains a list of variables to be dropped

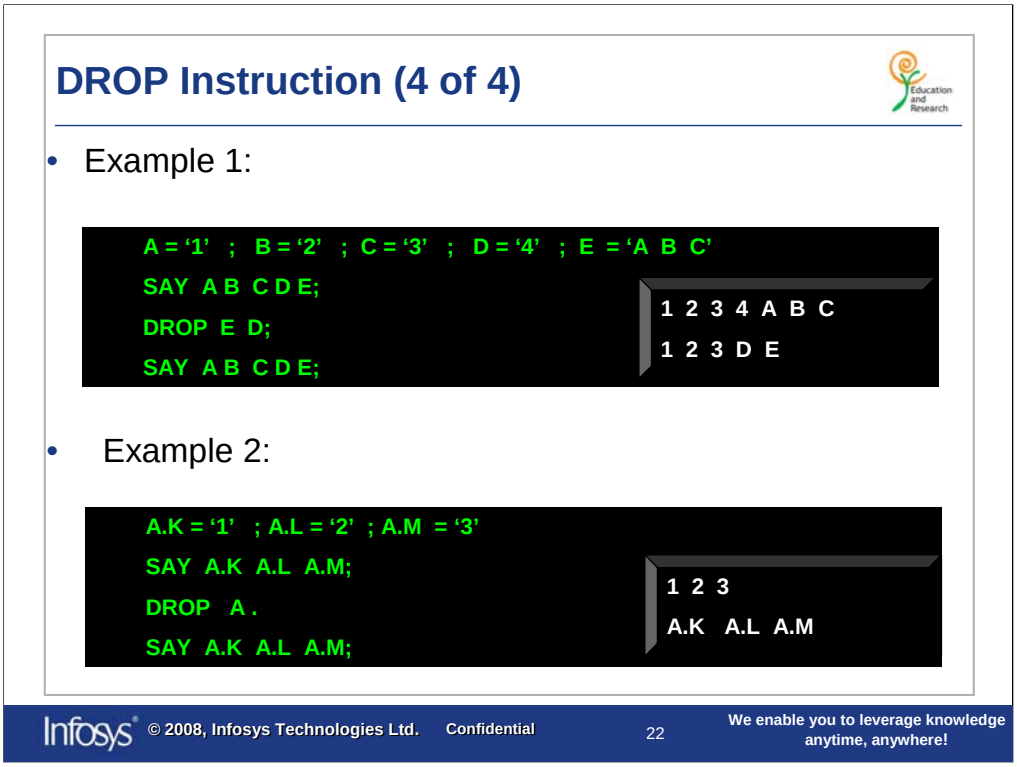
```
/* REXX */
A = '1' ; B = '2' ; C = '3' ; D = '4' ; E = 'A B C'
SAY A B C D E;
DROP (E) D;
SAY A B C D E;
EXIT
```

In the above example, E is a variable which contains a list of variables A, B & C.

In the DROP instruction, E is specified in parentheses. Thus, the contents of A B & C will be dropped.

Since D is also specified (outside parentheses), it's value will also be dropped.

NOTE: **E**'s value will not be dropped. This is because it is specified within parentheses & thus, it will be assumed to contain a list of variables that are to be dropped.



DROP Instruction(4 of 4) – Notes

Example 1:

```
A = '1' ; B = '2' ; C = '3' ; D = '4' ; E = 'A B C'
SAY A B C D E;
DROP E D;
SAY A B C D E;
```

Output:

```
1 2 3 4 A B C
1 2 3 D E
```

In the above example, E is specified without the parentheses. Thus contents of E will be dropped.

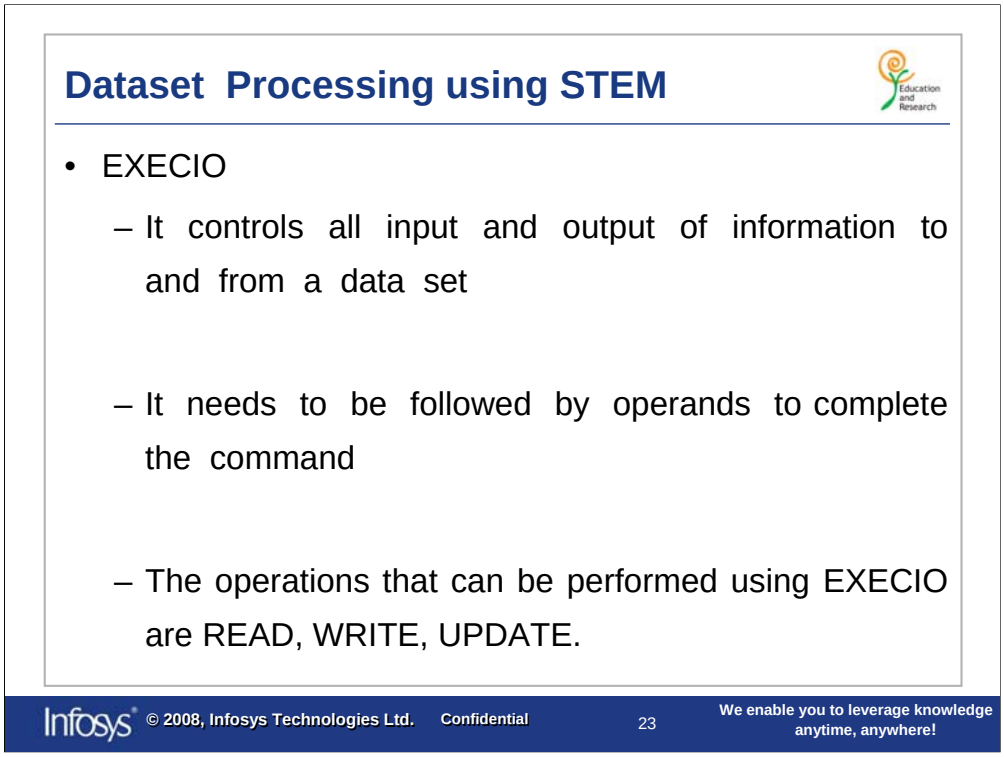
Example 2:

```
A.K = '1' ; A.L = '2' ; A.M = '3'
SAY A.K A.L A.M;
DROP A .
SAY A.K A.L A.M;
```

Output:

```
1 2 3
A.K A.L A.M
```

In the above example, stem A. is specified in the DROP instruction. Thus all the compound variables starting with A. will be dropped.



Dataset Processing using STEM - Notes

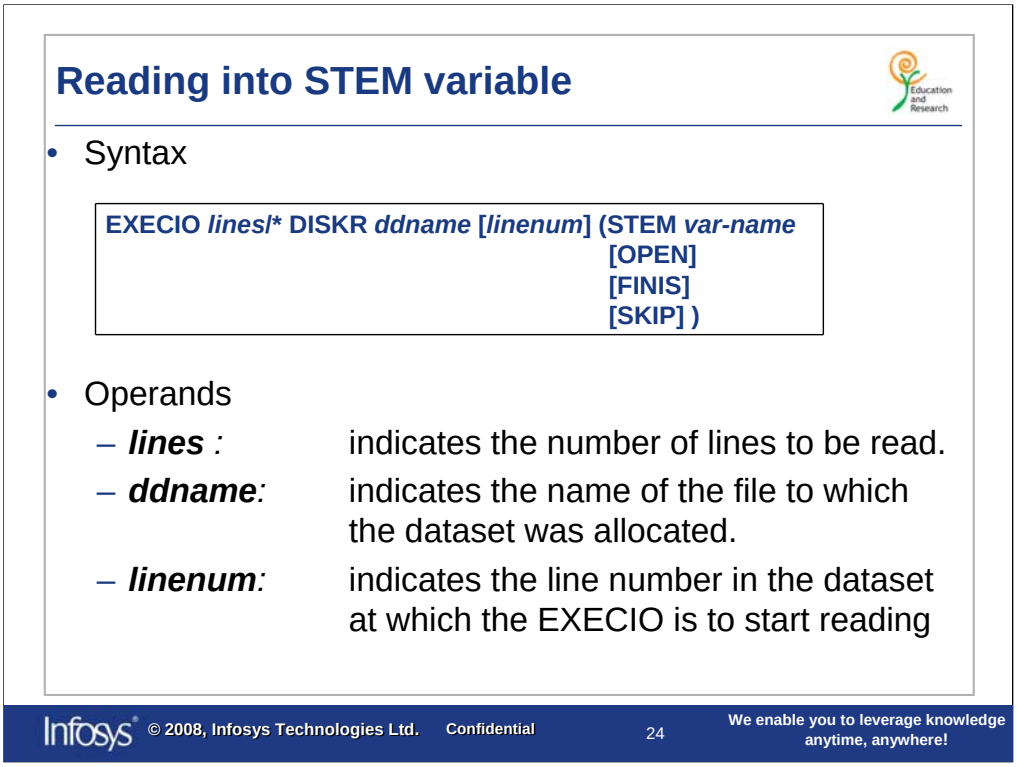
The main instruction for all data set processing is EXECIO . It controls all input and output of information to and from a data set . To complete the command , EXECIO needs to be followed by operands . As we go ahead , we discuss different EXECIO input and output operands that need to be used in conjunction while reading or writing to a data set .

The operations that can be performed are READ, WRITE, UPDATE.

Using EXECIO, it is possible to read from a dataset to a datastack or to a STEM variable. It is also possible to write into a dataset from a datastack or from a stem variable.

In this course, all operations on a dataset like READ, WRITE, UPDATE using stem variables will only be covered.

Dataset operations using datastack is out of scope for this course.



Reading into STEM variable – Notes

EXECIO *lines*/* DISKR *ddname* [*linenum*] (STEM var-name [OPEN][FINIS][SKIP])

Following are the read operands:

lines :

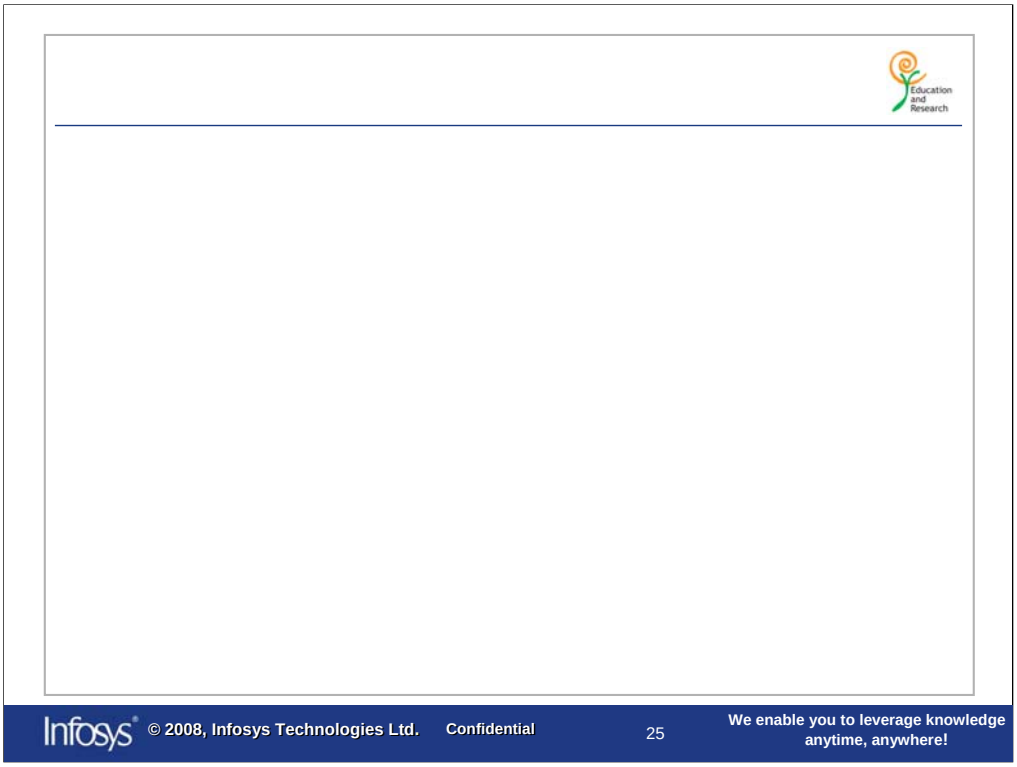
It indicates the number of lines to be read. To read all the lines, specify ***** instead of a number. If the value specified for *lines* is 0, no I/O operations are performed unless either OPEN or FINIS or both OPEN & FINIS are specified. In

such cases:

1. If OPEN is specified and the dataset is closed, EXECIO opens the dataset but will not read any lines. If OPEN is specified and the dataset is already open, then also EXECIO will not read any lines. In both the cases, if a non-zero value is specified for the *linenum* operand, EXECIO will set the current record number to the record number specified by the *linenum* operand.

NOTE: By default, the current record number will be set to the first record. Current record can be defined as the number of the next record EXECIO will read. However, if a non-zero value is specified for *linenum* , then EXECIO sets the current record number to the record number indicated by the *linenum* operand.

2. If FINIS is specified & the dataset is open, EXECIO will not read any lines, but it will close the dataset. If FINIS is specified & the dataset is closed already, then EXECIO will not do anything.
3. If both OPEN & FINIS are specified, then EXECIO will process OPEN first & then FINIS.



DISKR: This operand will open a dataset for input (if it is not already open) & will read the specified number of lines from the dataset.

ddname: This indicates the name of the file to which the dataset was allocated. The dataset can either be a sequential file or a member of a PDS. The file has to be allocated before issuing EXECIO. ALLOCATE command can be used for this.

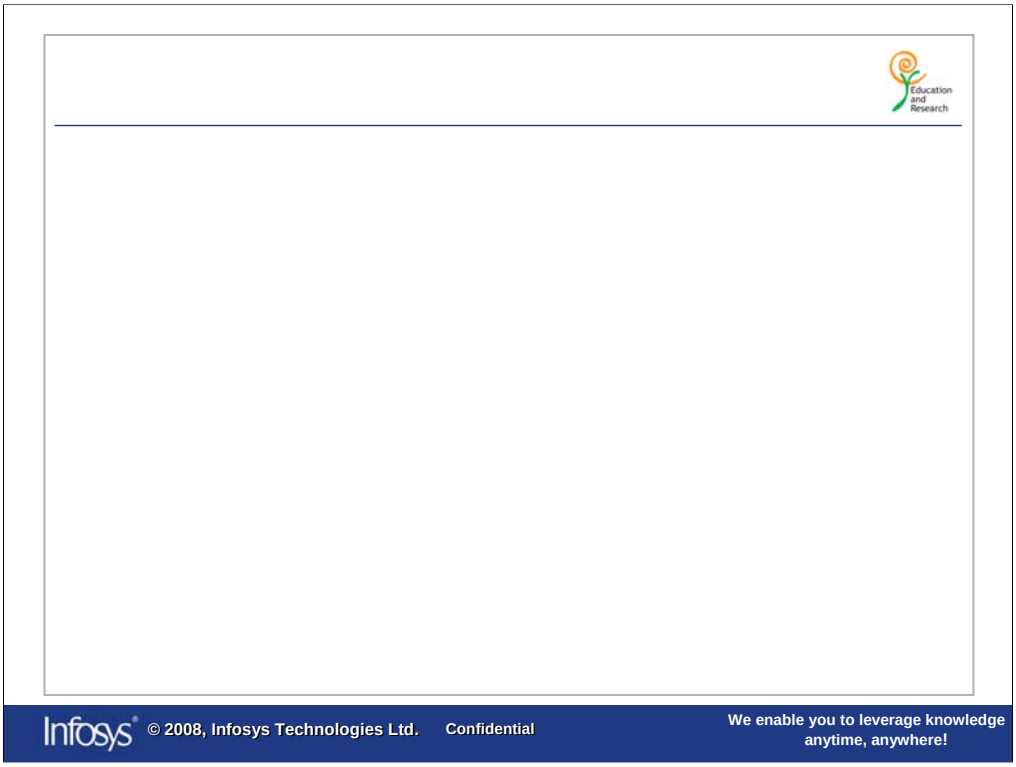
linenum: This indicates the line number in the dataset at which the EXECIO has to start reading. When a dataset is opened for input or update, the current record number is the number of the next record which will be read by EXECIO. If a zero value is specified for *linenum,* then it is as though *linenum* has not been specified at all.

FINIS: This is used to close the dataset after the EXECIO command completes. It is a good programming practice to explicitly close the datasets using FINIS. If FINIS is not specified, dataset is closed when one of the following occurs:

a. The task under which the dataset was opened, is terminated

b. The language processor environment associated with the task, under which the dataset was opened, is terminated

OPEN: This is used to open the specified dataset if it is not open already.

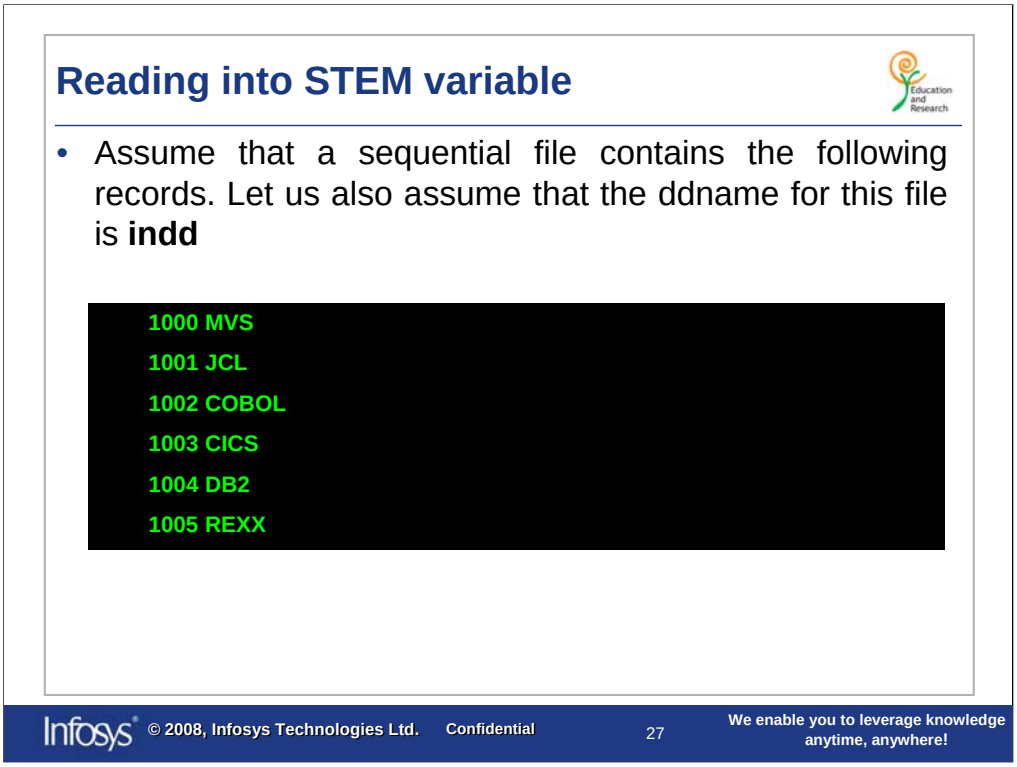


STEM *var-name*: It specifies the stem of the variables into which information is to be placed. To place information in compound variables, the *var-name* should end with a period. For example: record.

SKIP: If SKIP is specified, then the specified number of variables will be read but will not be placed in the variables

ddname: It indicates the name of the file to which the dataset was allocated.

*linenum:*It indicates the line number in the dataset at which the EXECIO is to start reading



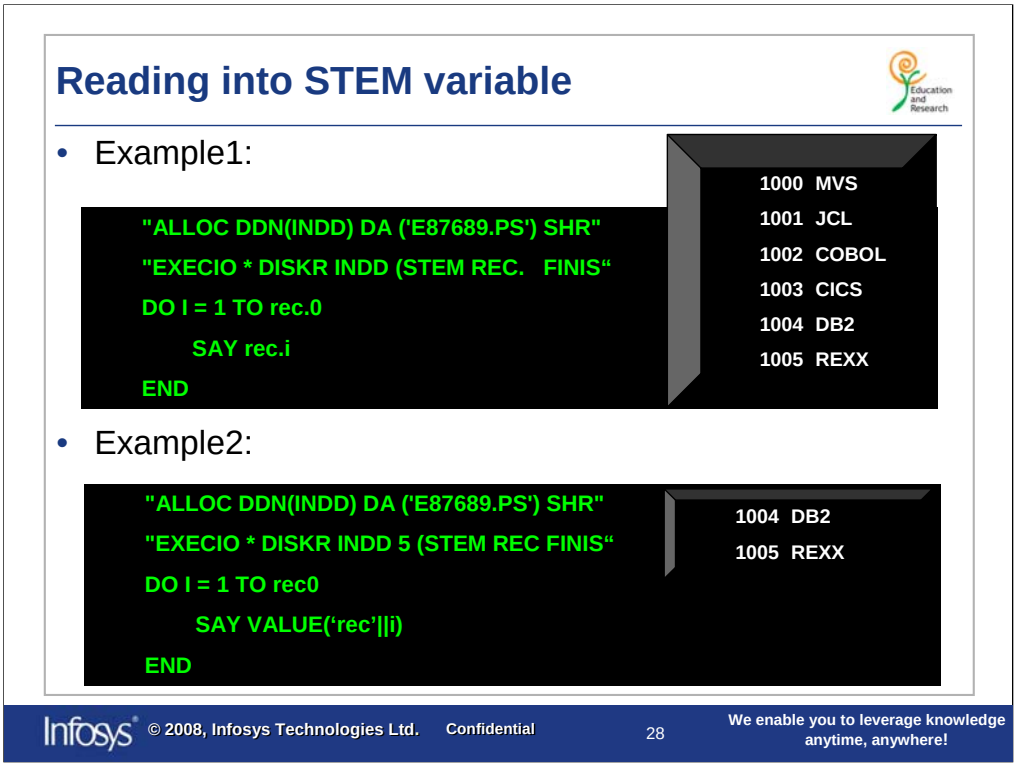
Reading into STEM variable – Notes

Assume that a sequential file contains the following records.

Let us also assume that the ddname for this file is **indd.** This ddname should be the same as specified in the ALLOCATE command.

Example for allocate: assuming that the sequential file name is userid.ps, below shown is the usage of allocate command.

“ALLOC DDN(indd) DA (‘userid.ps’) shr”



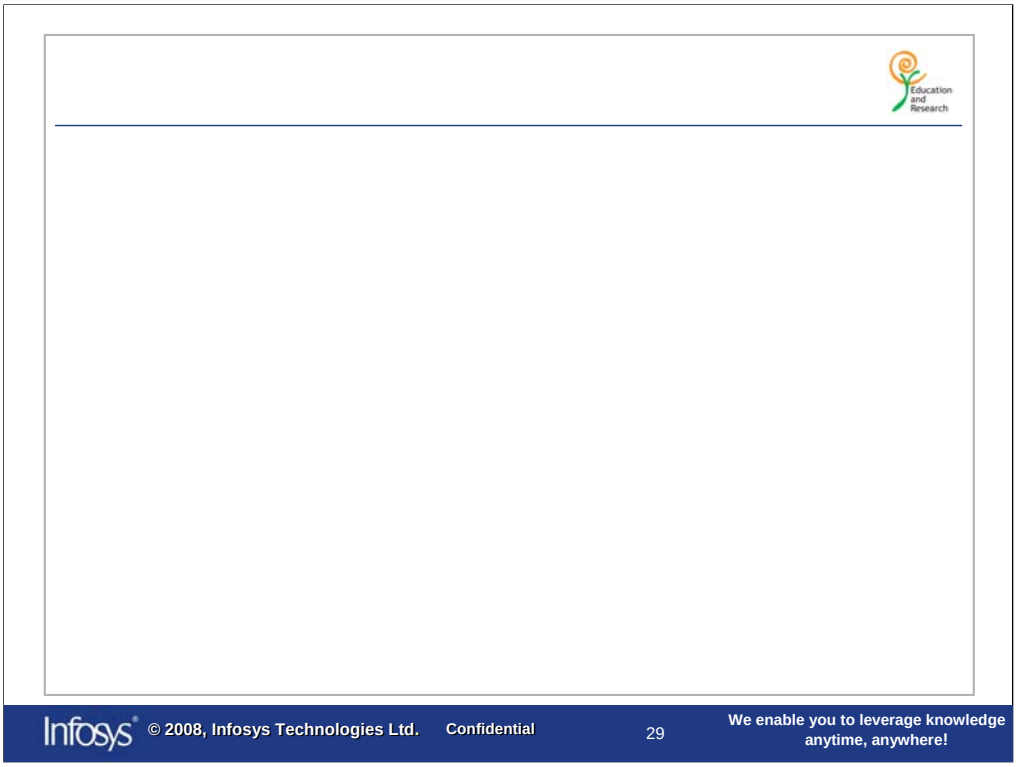
Reading into STEM variable – Notes

Example 1:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO * DISKR INDD (STEM REC. FINIS“
DO I = 1 TO rec.0
        SAY rec.i
END
```

In the above example, the EXECIO statement specifies * for *lines.* Thus, all the records will be read from the dataset. Note that *linenum* has not been specified. Hence, all records starting from the first record will be read into compound variables rec.1, rec.2 and so on up to rec.6 (since there are 6 records in the dataset).

Important thing to note is, when records are read into stem variable, stemname.0 will contain the number of records read. In the above example, rec.0 will contain 6 (total number of records read).



Example 2:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO * DISKR INDD 5 (STEM REC FINIS“
DO I = 1 TO rec0
        SAY VALUE(‘rec’||i)
END
```

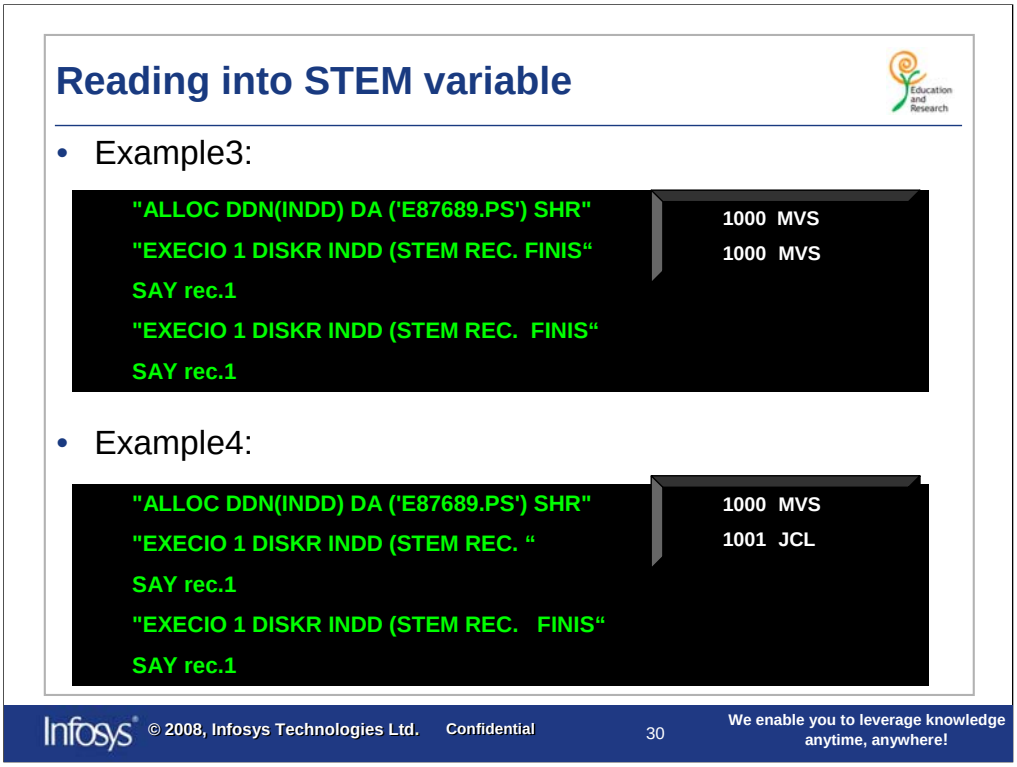
In this example, *lines* = * & *linenum* = 5. Hence, all the records starting from 5th will be read into the stem rec

Unlike in the first example, here, the stem specified does not contain a period. Thus records will be read into variables rec1, rec2, rec3 and so on…In this case, total number of records read will be present in stemname0 (rec0, in this example).

In order to display the records, the loop should be as shown below:

```
DO I = 1 TO rec0
        SAY VALUE(‘rec’||i)
END
```

Here, VALUE is a built-in function which returns the value of a variable. In this case, for every iteration the value of ‘i’ will be concatenated with ‘rec’ & the value of the resulting variable will be displayed.



Reading into STEM variable – Notes

Example 3:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO 1 DISKR INDD (STEM REC. FINIS“
SAY rec.1
"EXECIO 1 DISKR INDD (STEM REC. FINIS“
SAY rec.1
```

In the above example, there are two EXECIO statements. Both of them read a single record. rec.1 contains the record. The output will be:

```
1000  MVS
1000    MVS
```

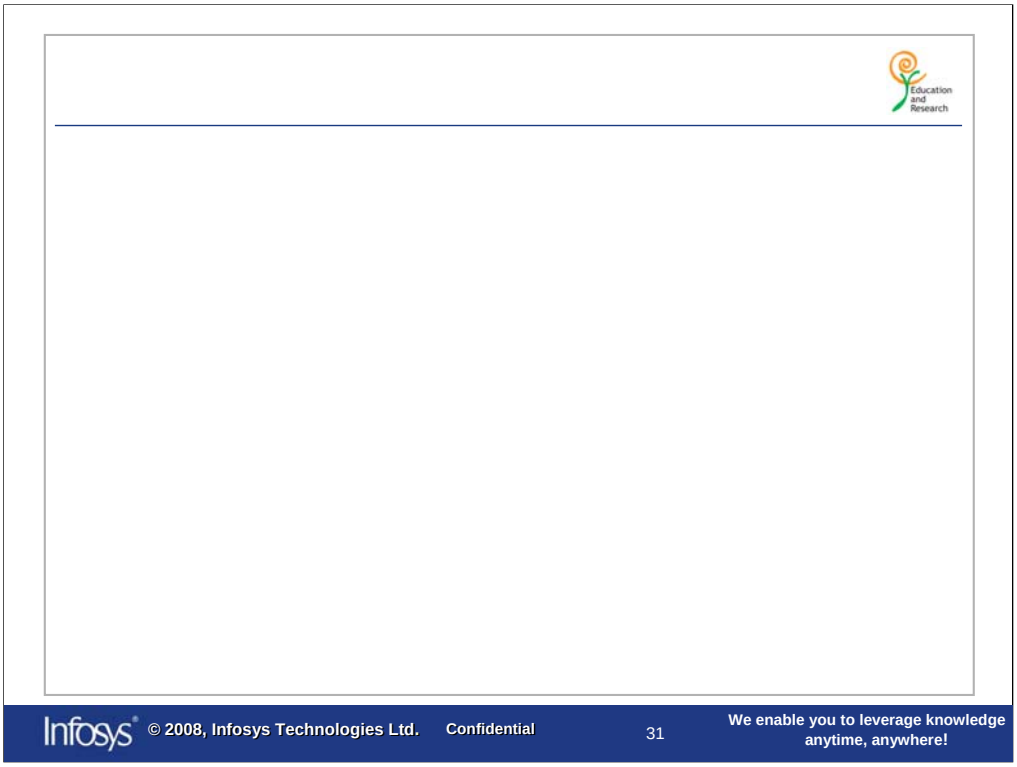
Example 3:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO 1 DISKR INDD (STEM REC. “
SAY rec.1
"EXECIO 1 DISKR INDD (STEM REC. FINIS“
SAY rec.1
```

Output is:

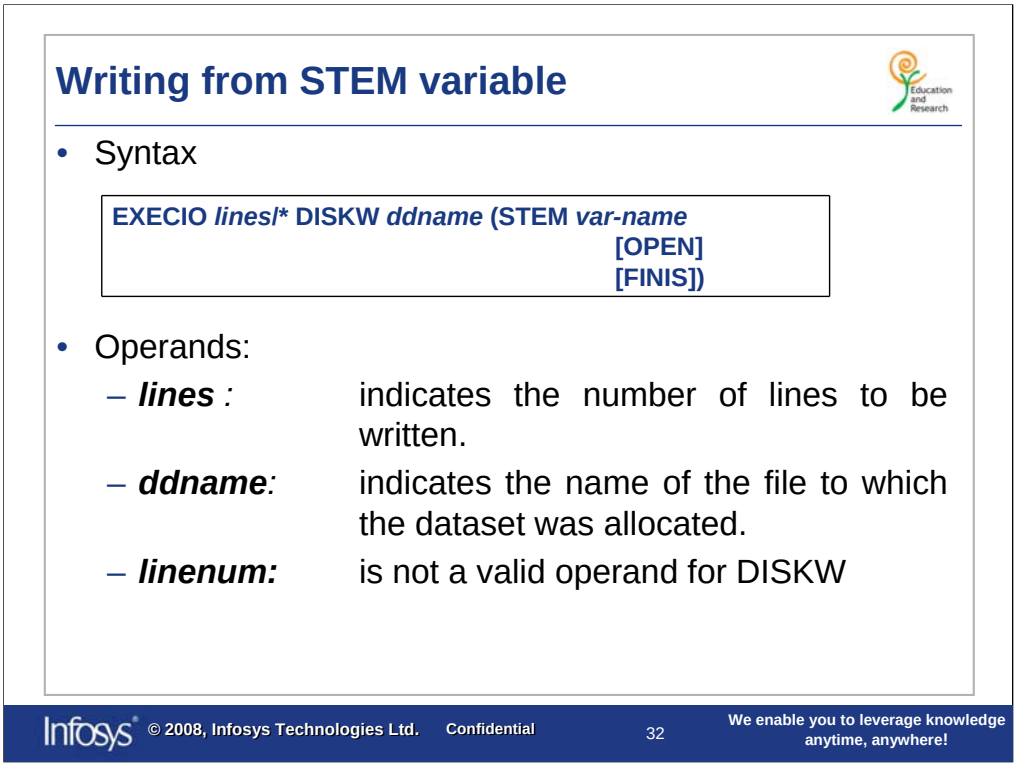
```
1000    MVS
1001    JCL
```

Even though both example 3 & 4 are similar, the outputs are different. What is missing in the 4th example???



In example 4, for the first EXECIO, FINIS is not specified, which means that the dataset is still open & the current record number after the execution of EXECIO will be set to 2 & hence the next EXECIO read statement will read the second record.

In example 3, for both EXECIO statements, FINIS is specified. After the execution of first EXECIO, dataset will be closed & current record number will be set to 0. The second EXECIO statement, however, sets the current record number to 1 & hence the same record is read by both the EXECIO statements.



Writing from STEM variable – Notes

Syntax:

EXECIO *lines*/* DISKW *ddname* (STEM *var-name* [OPEN][FINIS])

Following are the write operands:

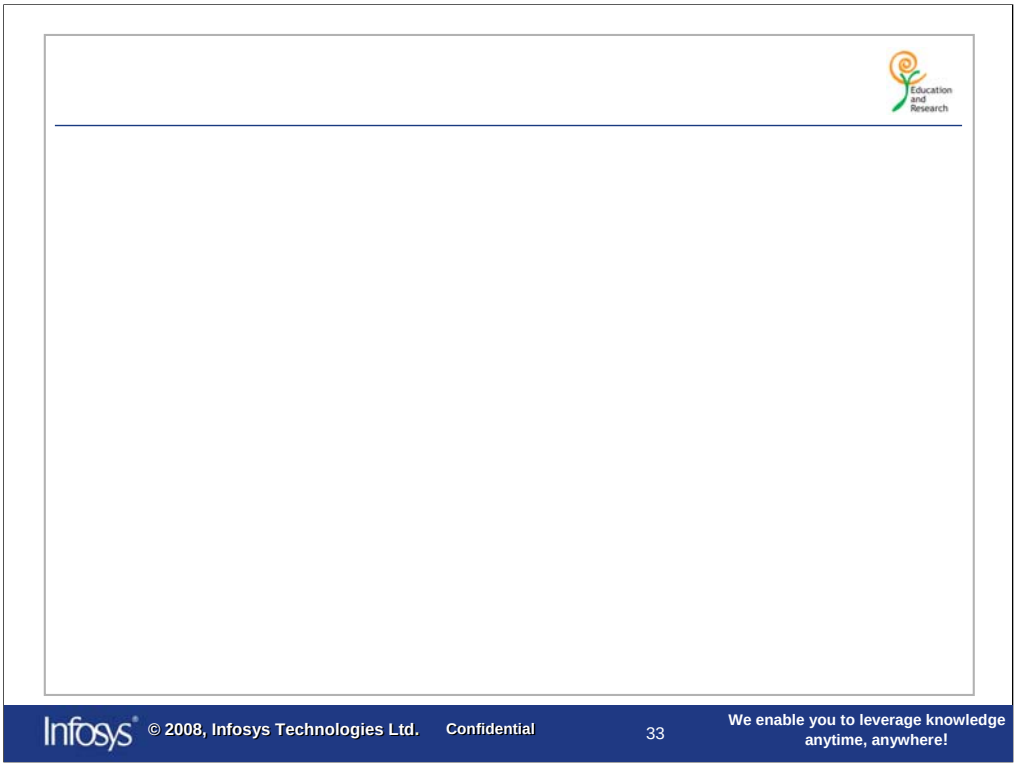
lines :

It indicates the number of lines to be written. If the value specified for *lines* is 0, no I/O operations are performed unless either OPEN or FINIS or both OPEN & FINIS are specified.

In such cases:

1. If OPEN is specified and the dataset is closed, EXECIO opens the dataset but will not write any lines. If OPEN is specified and the dataset is already open, then also EXECIO will not write any lines.
2. If FINIS is specified & the dataset is open, EXECIO will not write any lines, but it will close the dataset. If FINIS is specified & the dataset is closed already, then EXECIO will not do anything.
3. If both OPEN & FINIS are specified, then EXECIO will process OPEN first & then FINIS.

Note: When * is specified for *lines,* EXECIO will continue writing records from the compound variables until it reaches a null value or an uninitialized variable.



DISKW: This operand will open a dataset for output (if it is not already open) & will write the specified number of lines to the dataset. DISKW is also used to rewrite the last read record in case of updation.

ddname: This indicates the name of the file to which the dataset was allocated. The dataset can either be a sequential file or a member of a PDS. The file has to be allocated before issuing EXECIO. ALLOCATE command can be used for this.

FINIS: This is used to close the dataset after the EXECIO command completes. It is a good programming practice to explicitly close the datasets using FINIS. If FINIS is not specified, dataset is closed when one of the following occurs:

a. The task under which the dataset was opened, is terminated

b. The language processor environment associated with the task, under which the dataset was opened, is terminated

To close an open dataset without writing any record into a dataset, specify *lines* as 0 & also specify FINIS.

OPEN: This is used to open the specified dataset if it is not open already.

To open a closed dataset without writing any record into the dataset, specify *lines* as 0 along with OPEN.

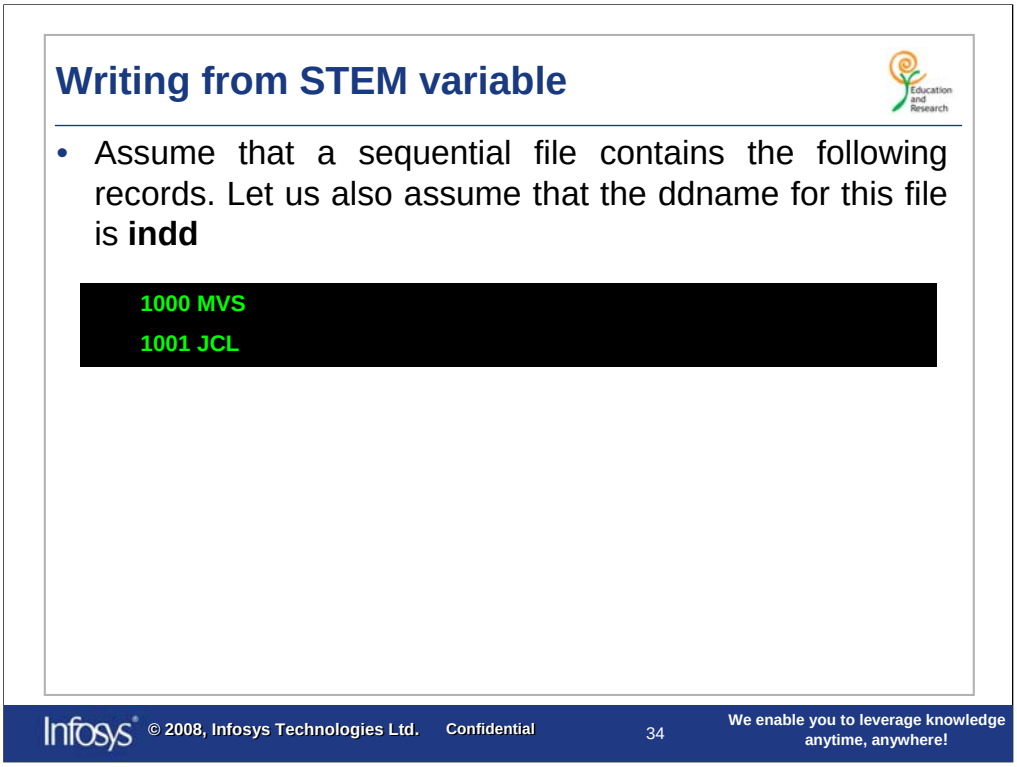
STEM *var-name*: It specifies the stem of the variables from which information is to be written into the dataset.

If records are to appended to the existing set of records then ALLOCATE a dataset using MOD parameter.

If records are to overwritten then ALLOCATE a dataset using SHR parameter.

Note:

SKIP & *linenum* are not valid operands for write operation.



Writing from STEM variable – Notes

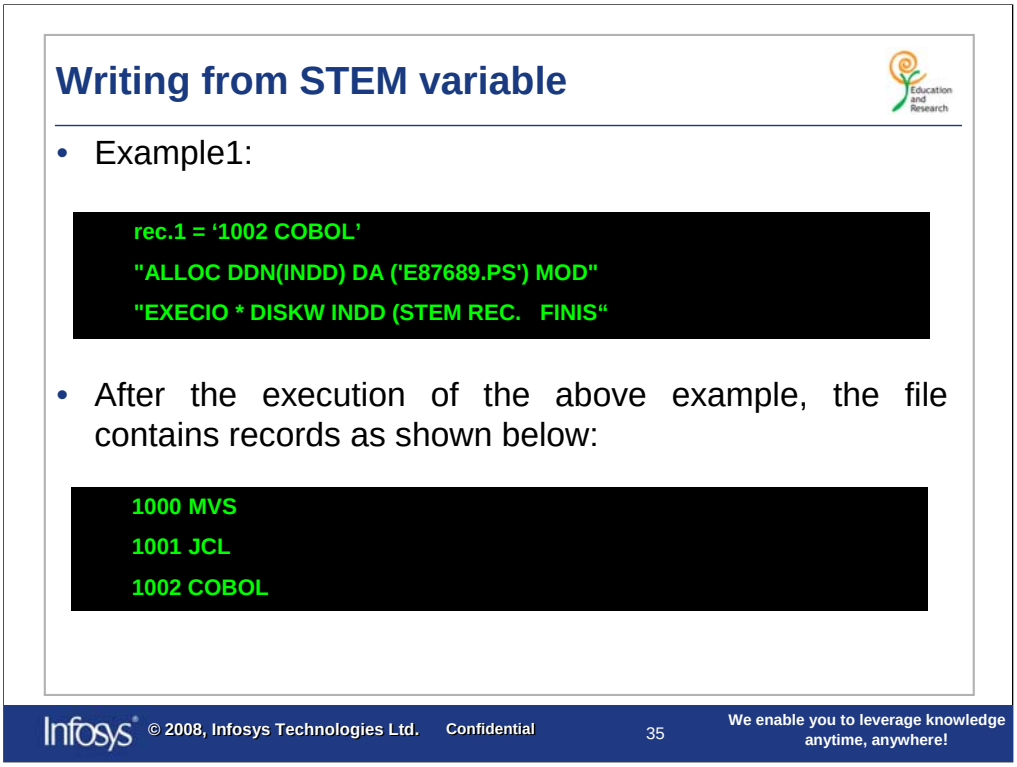
Assume that a sequential file contains the following records.

Let us also assume that the ddname for this file is **indd.** This ddname should be the same as specified in the ALLOCATE command.

Example for allocate: assuming that the sequential file name is userid.ps, below shown is the usage of allocate command.

"ALLOC DDN(indd) DA ('userid.ps') SHR" (if records are to be overwritten)

"ALLOC DDN(indd) DA ('userid.ps') MOD" (if records are to be appended to the existing set of records)



Writing from STEM variable – Notes

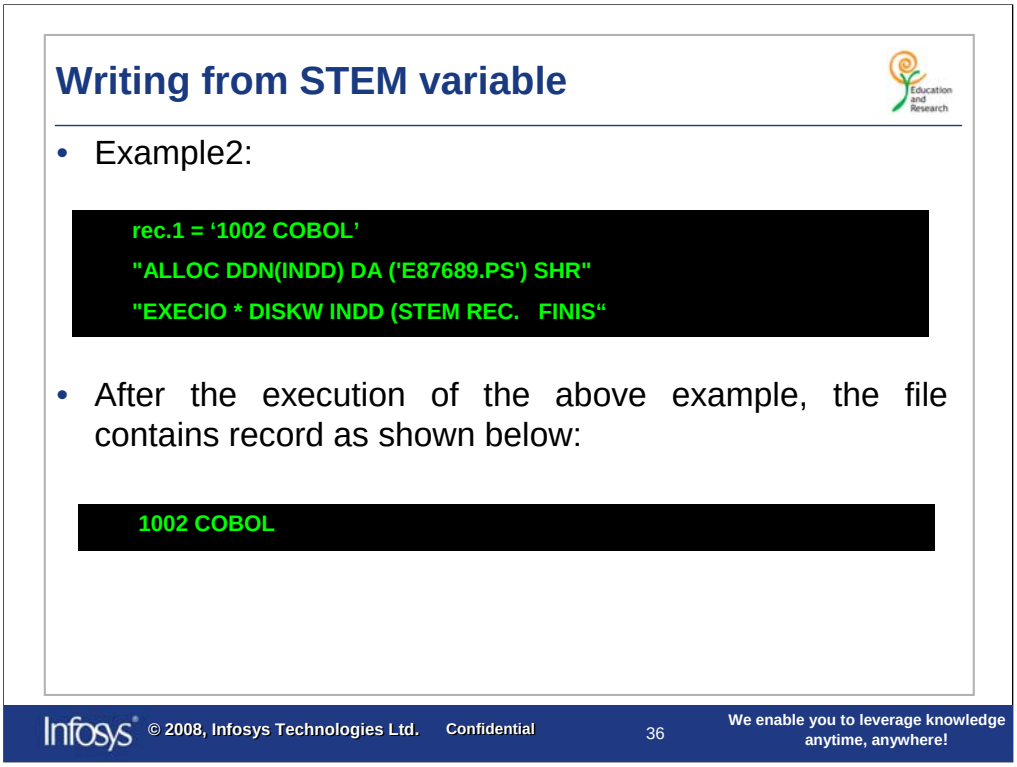
Example 1:

rec.1 = '1002 COBOL'

"ALLOC DDN(INDD) DA ('E87689.PS') MOD"

"EXECIO * DISKW INDD (STEM REC. FINIS"

Here, dataset is allocated with MOD parameter. Hence record '1002 COBOL' will be appended with the existing 2 records.



Writing from STEM variable – Notes

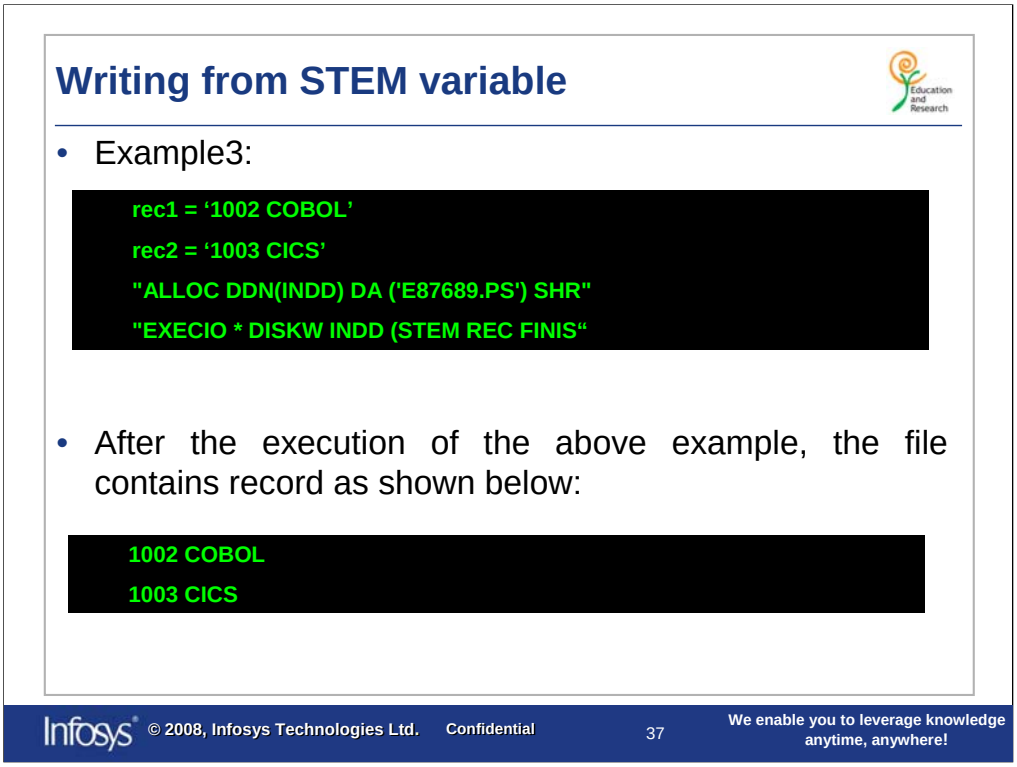
Example 2:

rec.1 = '1002 COBOL'

"ALLOC DDN(INDD) DA ('E87689.PS') SHR"

"EXECIO * DISKW INDD (STEM REC. FINIS"

Here, dataset is allocated with SHR parameter. Hence record '1002 COBOL' will overwrite the existing 2 records.



Writing from STEM variable – Notes

Example 3:

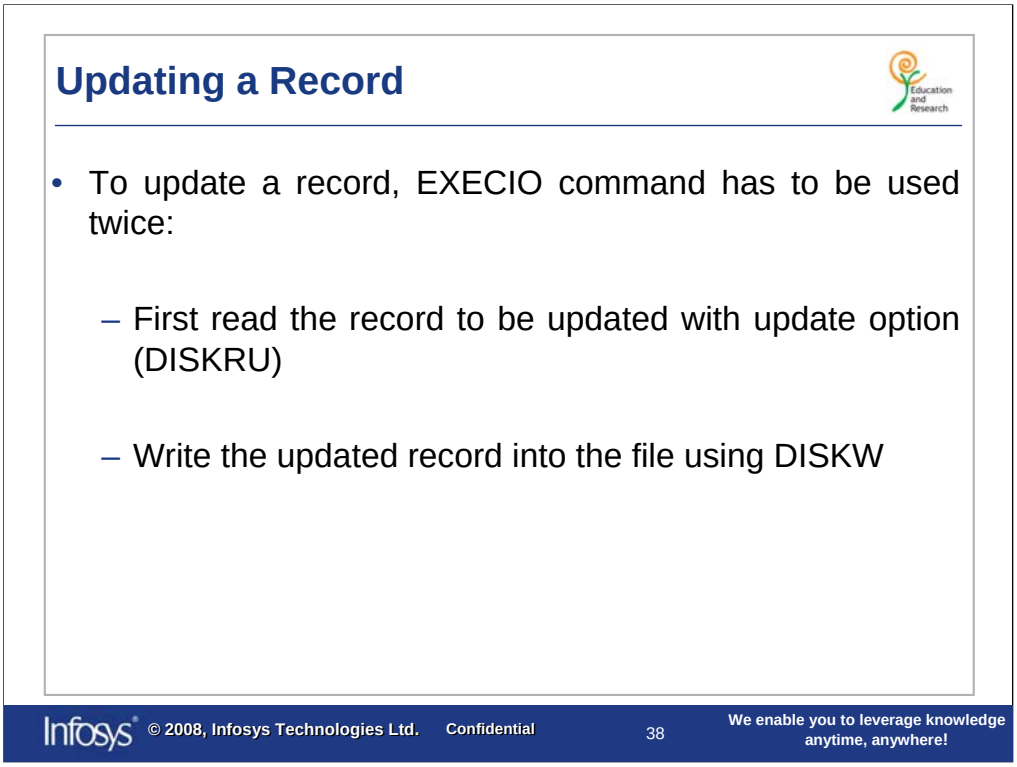
rec1 = '1002 COBOL'

rec2 = '1003 CICS'

"ALLOC DDN(INDD) DA ('E87689.PS') SHR"

"EXECIO * DISKW INDD (STEM REC FINIS"

In the above example, the records to be written are present in variables REC1 & REC2 instead of compound variables REC.1 and REC.2. However, even this is valid. Just specify 'STEM REC' instead of 'STEM REC.' in the EXECIO statement.



Updating a Record – Notes

To update a record, EXECIO command has to be used twice:

First read the record to be updated with update option (DISKRU)

Write the updated record into the file using DISKW

Updating a Record

- Syntax

```
EXECIO lines DISKRU ddname [linenum] (STEM var-name
                                      [OPEN]
                                      [FINIS]
                                      [SKIP] )
```

- Operands
  - ***lines*** *:* indicates the number of lines to be read.
  - ***ddname****:* indicates the name of the file to which the dataset was allocated.
  - ***linenum****:* indicates the line number in the dataset at which the EXECIO is to start reading

Updating a Record – Notes

EXECIO *lines* DISKRU *ddname* [*linenum*] (STEM var-name [OPEN][FINIS][SKIP])

Operands

lines *:*

It indicates the number of lines to be read.

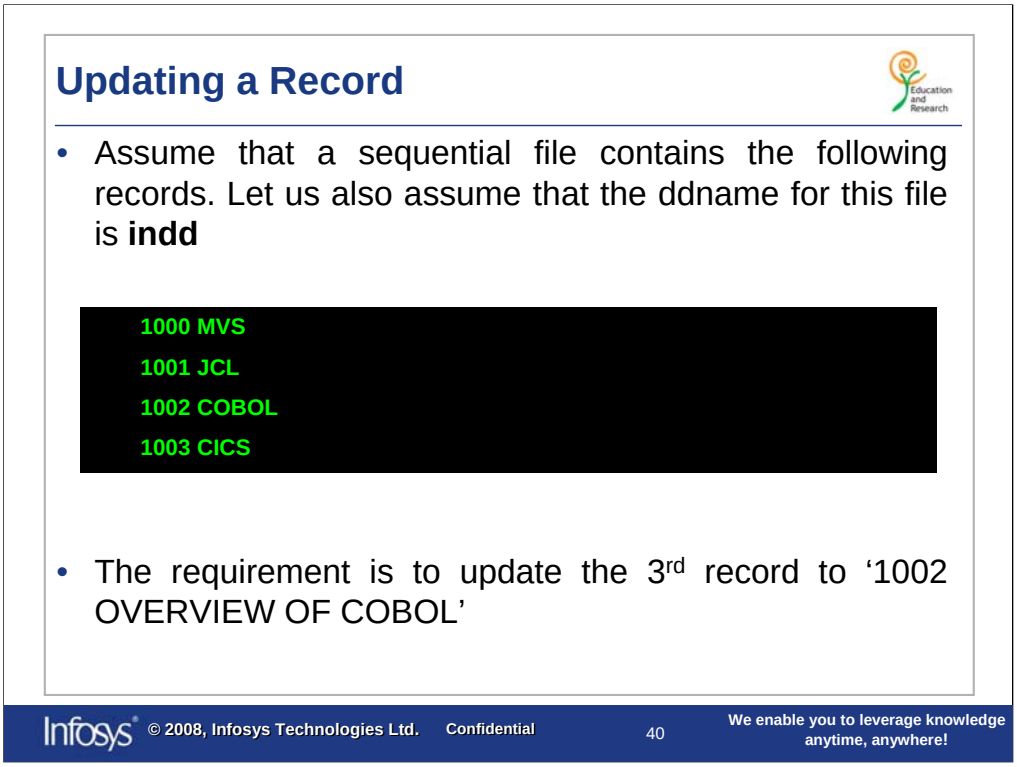
ddname*:*

It indicates the name of the file to which the dataset was allocated.

linenum*:*

It indicates the line number in the dataset at which the EXECIO is to start reading

DISKRU: Opens the dataset for update (if not already open) and places the number of records specified ina list of compound variables. When a dataset is opened for update, the last read record can be modified & written back using the EXECIO DISKW.



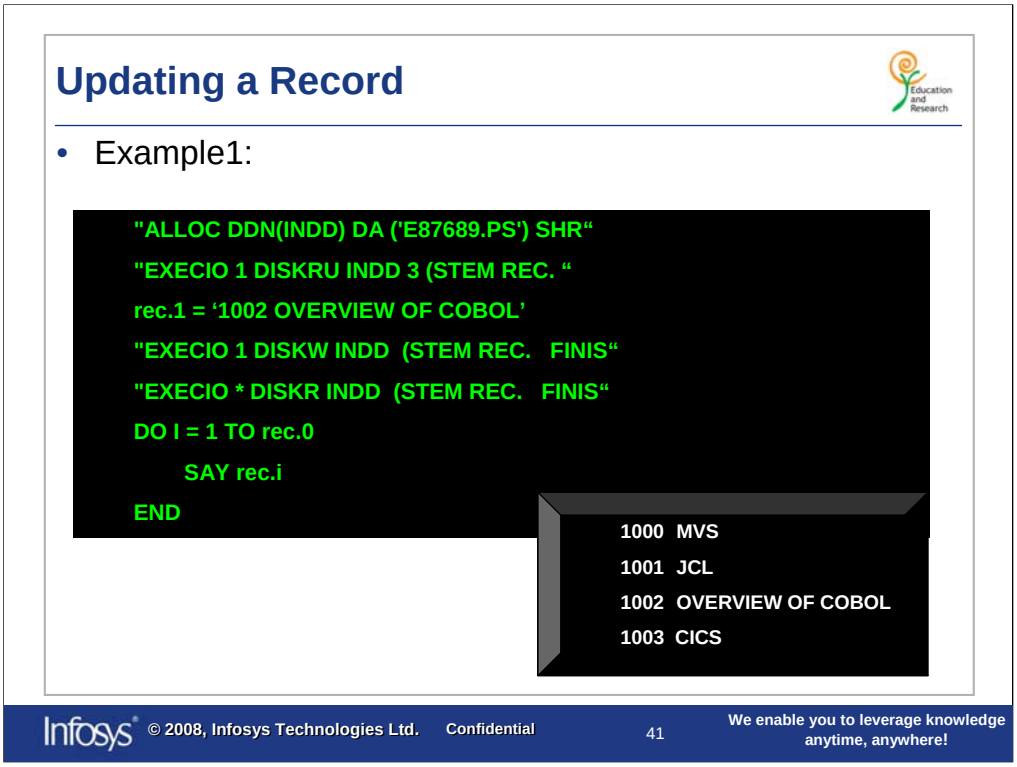
Updating a Record – Notes

Assume that a sequential file contains the following records.

Let us also assume that the ddname for this file is **indd.** This ddname should be the same as specified in the ALLOCATE command.

Example for allocate: assuming that the sequential file name is userid.ps, below shown is the usage of allocate command.

"ALLOC DDN(indd) DA ('userid.ps') shr"



Updating a Record – Notes

Example 1:

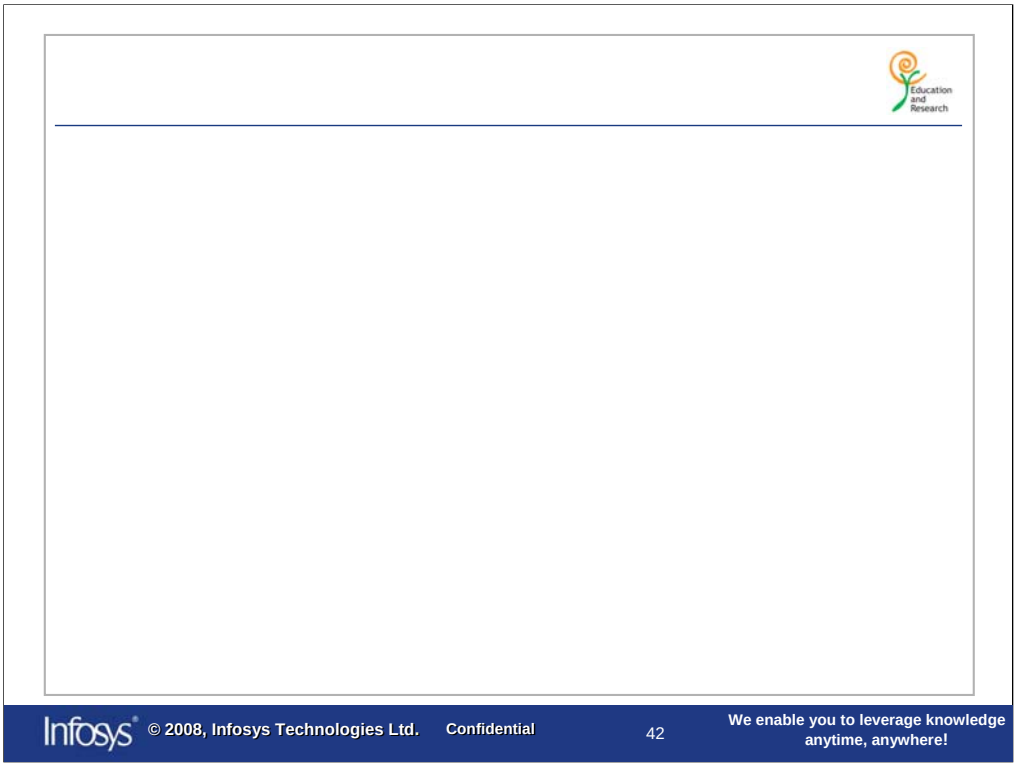
```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR“
"EXECIO 1 DISKRU INDD 3 (STEM REC. “
rec.1 = ‘1002 OVERVIEW OF COBOL’
"EXECIO 1 DISKW INDD (STEM REC. FINIS“
"EXECIO * DISKR INDD (STEM REC. FINIS“
DO I = 1 TO rec.0
        SAY rec.i
END
```

The output will be

```
1000    MVS
1001    JCL
1002    OVERVIEW OF COBOL
1003  CICS
```

Here, the 3rd record will be updated.

In the above example, FINIS is not specified for the EXECIO DISKRU statement.



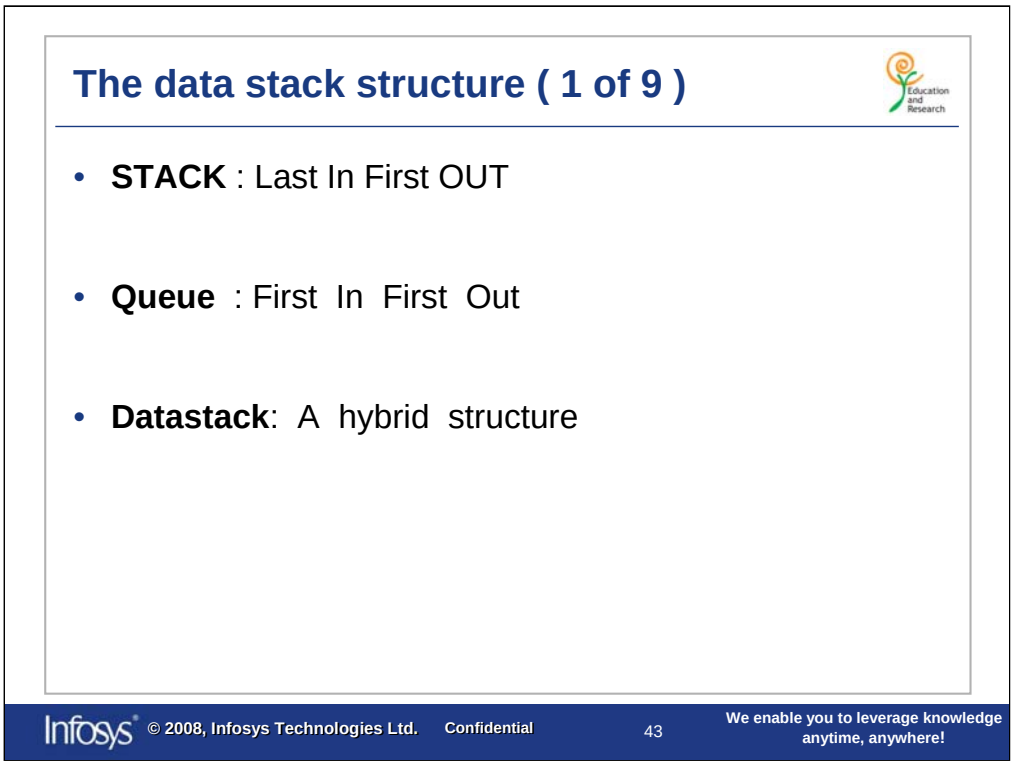
Example 2:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR“
"EXECIO 1 DISKRU INDD 3 (STEM REC. FINIS “
rec.1 = ‘1002 OVERVIEW OF COBOL’
"EXECIO 1 DISKW INDD  (STEM REC. FINIS“
"EXECIO * DISKR INDD  (STEM REC. FINIS“
DO I = 1 TO rec.0
        SAY rec.i
END
```

The output will be

```
1002     OVERVIEW OF COBOL
```

If FINIS is specified in the EXECIO DISKRU statement, then the dataset will be closed after the execution of the statement & instead of updating a record, it will execute the DISKW as an independent write statement & record will be either appended or overwritten (based on SHR or MOD). In the above example, mode is SHR and hence the records will be overwritten.



The data stack structure (1 of 9) – Notes

A stack is a data structure that is described as Last In First Out. i.e. Items are pushed onto the top of a stack and are also removed from the top.

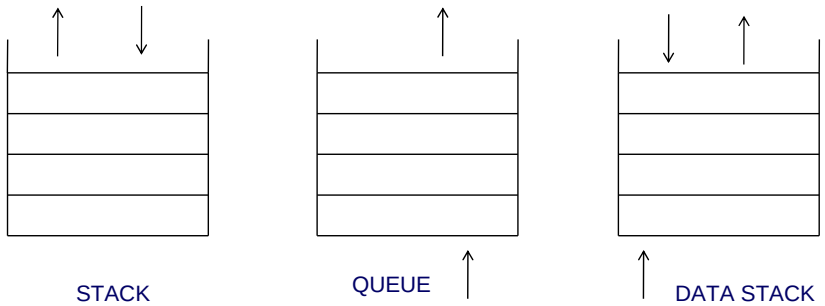
A queue is described as First In First Out . i.e. Items are pushed onto the bottom of a queue and removed from the front .

A data stack is a hybrid structure. It is neither a stack nor a queue , but a mix of the two.

Understanding the data stack structure in REXX becomes easier if we already have an understanding of the traditional stack and queue data structures. The stack data structure is also called a LIFO (Last In First Out) structure . Items are pushed onto the top of a stack and are also removed from the top. We can think of a stack as a pile of dishes. The latest dish we put on the pile will be the first one we take out of the pile . The first dish we put in the pile would obviously be the last one we take out when we empty the box. The queue data structure on the other hand is a FIFO (First In First Out) structure. Items are pushed onto the bottom of the queue and pulled off the top. A queue works just like a line-up for a movie . If we are the first person in the queue , we will be the first one to get the ticket at the counter and leave the queue. New people who want tickets join the queue and stand at the end/tail of the queue . The data stack structure is neither a queue nor a stack . It is a hybrid of the two as explained in the next slide.

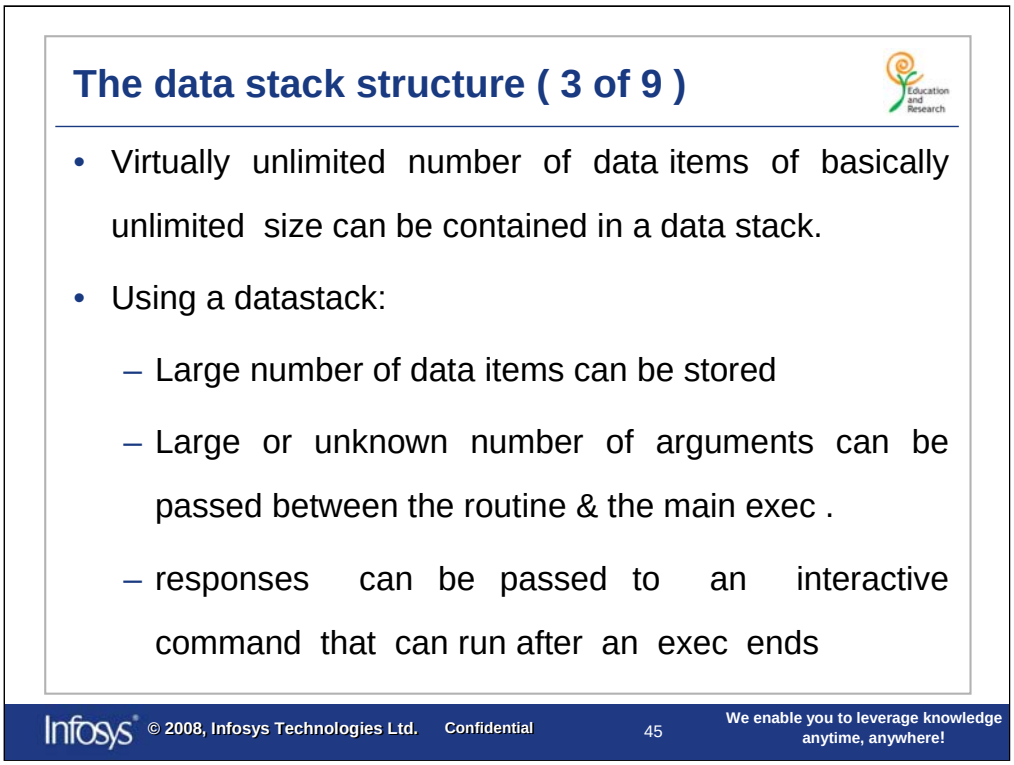
The data stack structure (2 of 9)

- Items can be pushed onto the top or bottom of a data stack . However items can only be pulled off the top exactly as with a stack or a queue.



The data stack structure (2 of 9) - Notes

The data stack structure is a combination of the stack and queue data structure. Items can be pushed onto the top or added to the bottom(or tail) of the data stack . However items can be pulled off or removed only from the top just as in a stack or a queue. We can use the data stack either as a stack or a queue , depending on how we process the items.



The data stack structure (3 of 9) - Notes

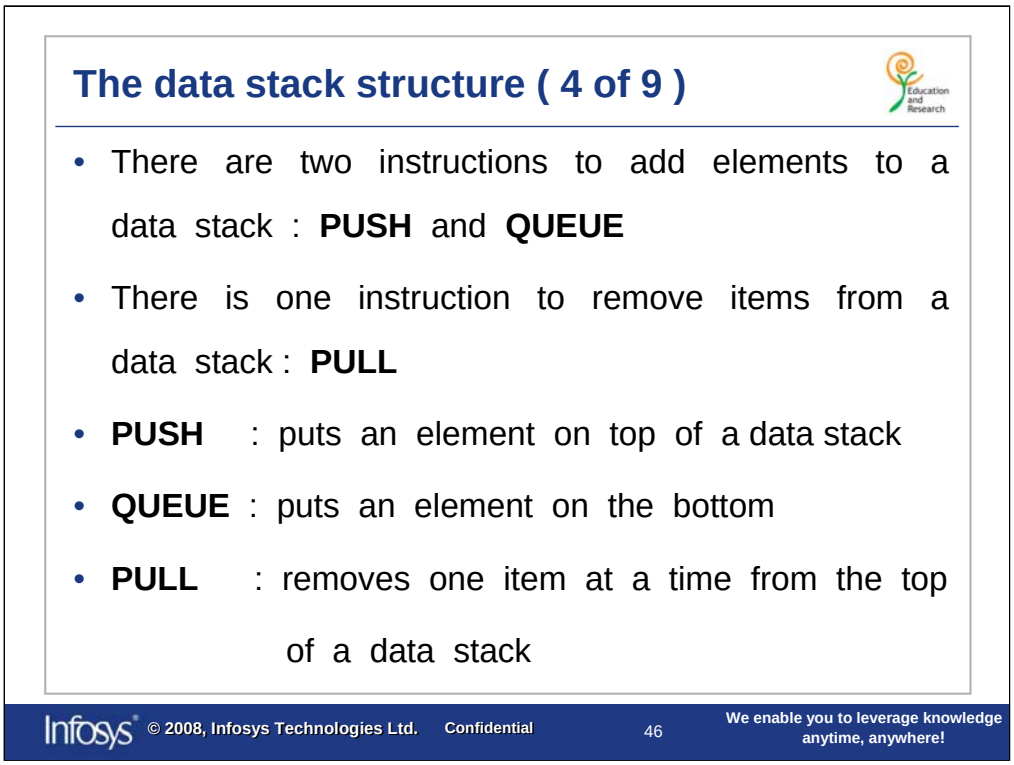
The data stack is a unique hybrid of the conventional stack and queue data structures but it also has a number of other unique characteristics. The data stack is a very flexible data structure with many applications. A data stack can contain virtually limitless number of data items of basically unlimited size. The data items can be commands that are to be executed when the exec ends. The flexibility of the data stack lends it to many potential uses/applications.

Using a datastack:

Large number of data items can be stored for a single exec's use.

Large or unknown number of arguments can be passed between the routine & the main exec .

responses can be passed to an interactive command that can run after an exec ends



The data stack structure (4 of 9) - Notes

We now take a look at the instructions for adding and removing elements from a data stack. There are two instructions for adding elements to a data stack:

The PUSH instruction adds an element on top of a data stack.

The QUEUE instruction puts an element on the bottom/rear of a data stack.

There is one instruction for removing elements from a data stack:

The PULL instruction removes one item at a time from the top of a data stack.

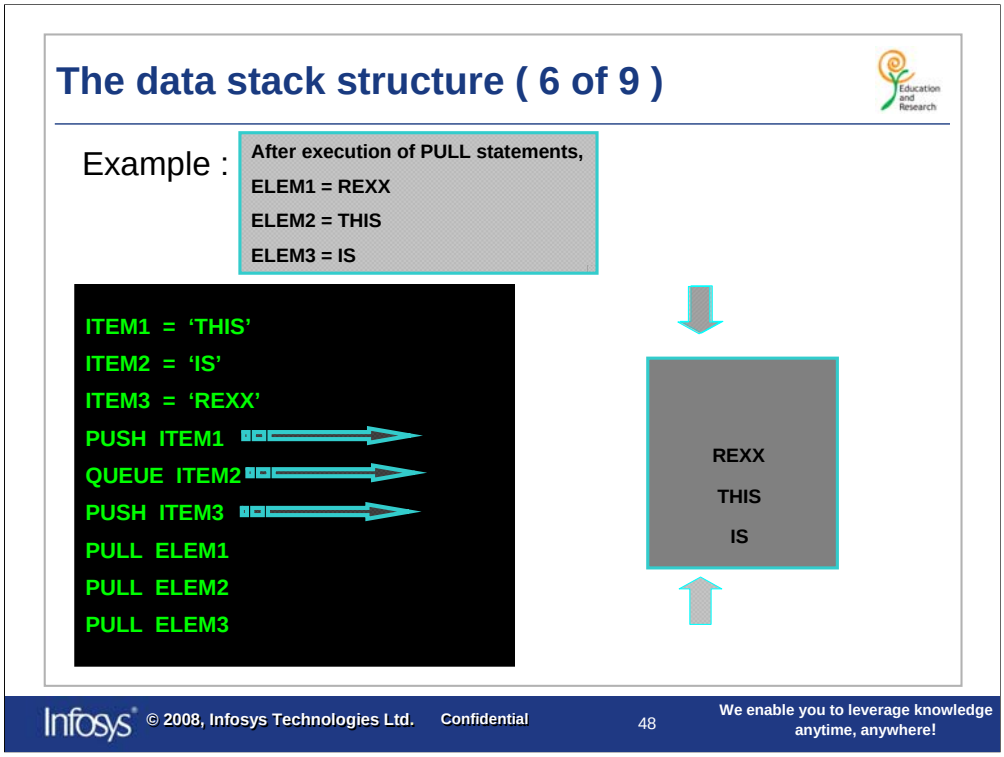
The data stack structure (5 of 9)

- PULL instruction is used to retrieve data from the terminal
- PULL instruction is also used to get information from a data stack.
- PULL instruction first checks the data stack & only if the datastack is empty, it will look at the terminal for data

The data stack structure (5 of 9) - Notes

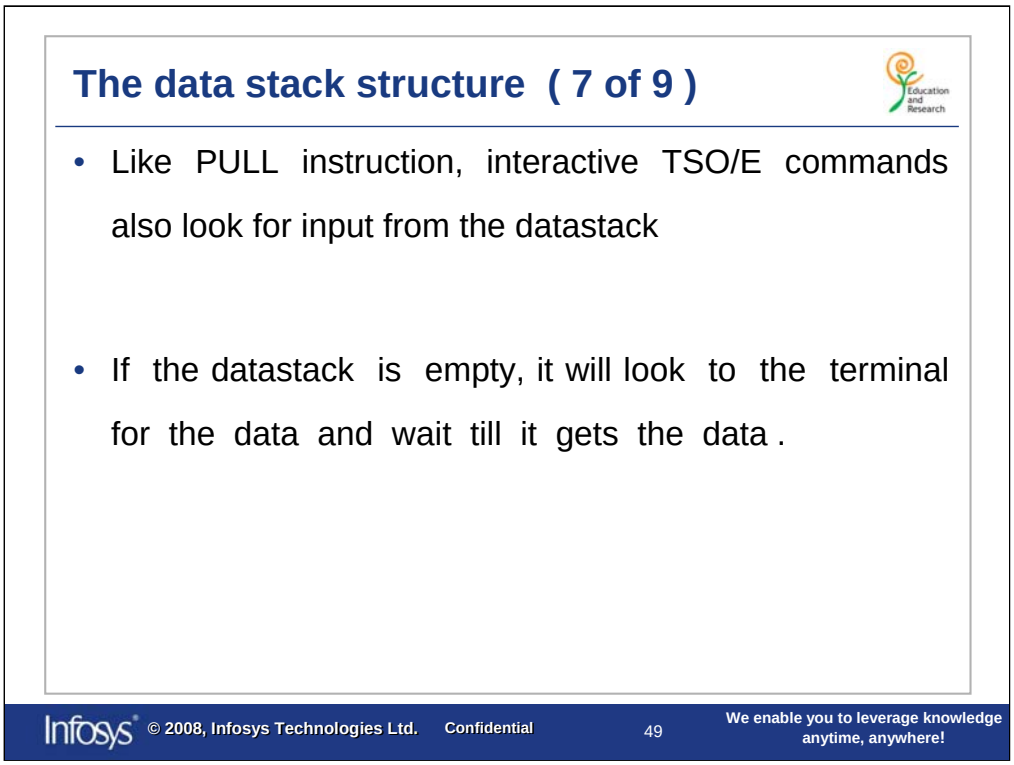
We had come across the PULL instruction before also while discussing about terminal I/O. The PULL instruction is used to retrieve data from the terminal . However here we mentioned that the PULL instruction is used to get information from a data stack . How can the PULL instruction be used to retrieve data both from the data stack as well as the terminal ?

When a PULL instruction is executed, it first checks the data stack . If it is empty , it will look to the terminal for input and wait till it gets some.



The data stack structure (6 of 9) - Notes

In the above code snippet, the first PUSH instruction will add 'THIS' to the data stack. The next QUEUE instruction will add 'IS' to the bottom of the data stack . The next instruction is a PUSH instruction which will add 'REXX' to the top of the data stack. The first PULL instruction removes the topmost item in the data stack i.e. 'REXX' . The next PULL instruction removes the next topmost item on the data stack i.e. 'THIS' and the third PULL instruction will remove the last item remaining in the data stack i.e. 'IS'



The data stack structure (7 of 9) - Notes

Just as in the case of the PULL instruction that first looks for input in the data stack and then if the data stack is empty proceeds to get the input from the terminal , interactive TSO/E commands e.g. ALLOCATE etc also first look for input in the data stack . Only if the data stack is empty , it will look to the terminal for data and wait till it gets some .

The data stack structure (8 of 9)

- The **QUEUED()** function is used to return the number of items in a data stack.

  Example :

```
---
------
---------
--------------
NUM = QUEUED()
SAY  NUM
```


The data stack structure (8 of 9) - Notes

There are times when we might need to know the number of items in a data stack , perhaps to set up a loop counter.

The built-in function QUEUED() is used to return the number of items in a data stack.

The data stack structure (9 of 9)

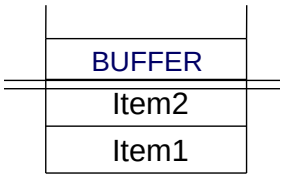
- A data stack to can be used to share information between an exec and it's routines only when it is alright to do so
- Unintentional sharing of data however may lead to errors and the routines could fail .
- We shall see later how to protect information existing in a data stack.

The data stack structure (9 of 9) - Notes

A data stack is a very good way of sharing data between an exec and it's routines if it is done intentionally. If routines unintentionally share data , there could be many errors and the routines could fail . There are some ways of protecting information existing in a data stack. We shall have a look at them a little later in the course.

Buffers and related functions (1 of 7)

- A buffer can be thought of as an extension of a data stack.
- A buffer is a way for an exec to create a temporary extension to the data stack



Buffers and related functions (1 of 7) - Notes

A buffer can be thought of as an extension of a data stack. This provides a way of creating a temporary extension i.e. storage that can be easily deleted once the required processing is complete. It is to be noted that buffers do not automatically protect elements below them. We shall discuss about this in detail a little later in this course.

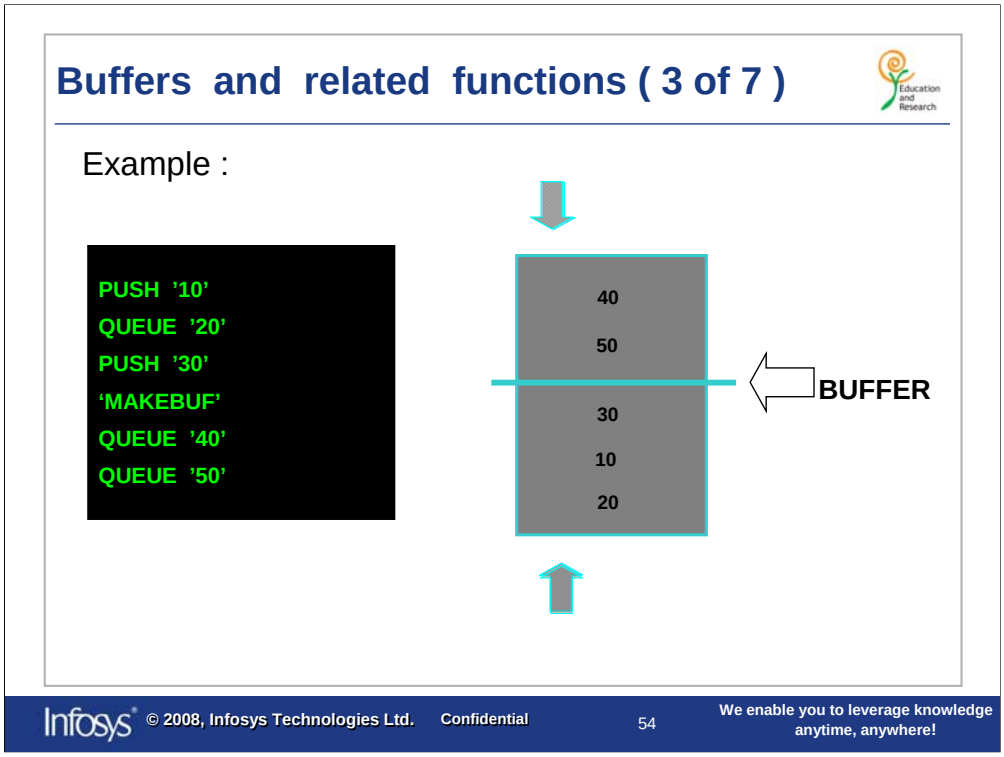
Buffers and related functions (2 of 7)

- **MAKEBUF** : command to create a buffer.
- **DROPBUF** : command to drop or delete buffers
- **Note** : Datastack & Buffer commands are actually TSO/E REXX commands . So they have to be enclosed in single quotes.

Buffers and related functions (2 of 7) - Notes

All buffer commands are actually TSO/E REXX commands . For this reason, they have to be enclosed in single quotes.

The command to create a buffer is MAKEBUF . Any number of buffers can be created on a data stack. The command to drop or delete buffers is DROPBUF. This command will drop one buffer at a time. So if we have created three buffers using the MAKEBUF command, then we have to issue the DROPBUF command three times in order to get rid of all the buffers.



Buffers and related functions (3 of 7) - Notes

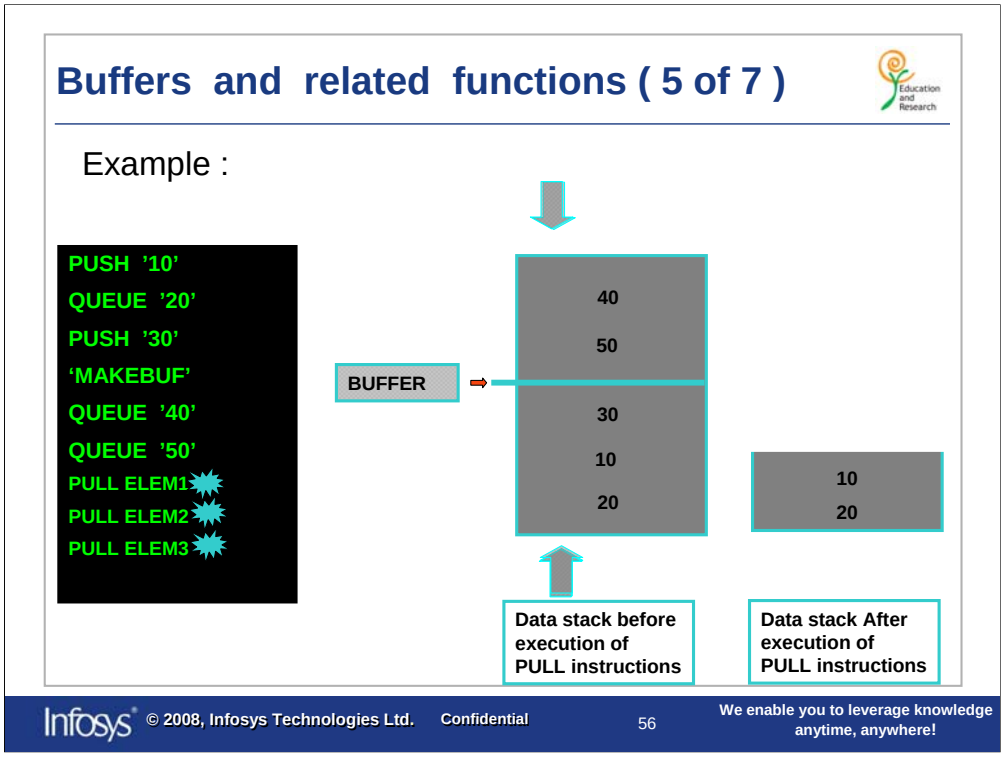
In the above code snippet , the PUSH statement adds 10 to the stack. The next QUEUE instruction adds 20 to the bottom of the stack . The next PUSH instruction adds 30 to the top of the stack. We then issue a MAKEBUF command . This will create a buffer on top of the data stack . We now proceed to add elements to this buffer . The first QUEUE instruction after the MAKEBUF command will add 40 to the buffer. The next QUEUE instruction will add 50 to the bottom of the buffer as shown in the slide .

Buffers and related functions (4 of 7)

- The **MAKEBUF** command does not protect elements below the buffer.
- Once all the elements in a buffer have been accessed by the exec , the elements added before the MAKEBUF command become accessible.

Buffers and related functions (4 of 7) - Notes

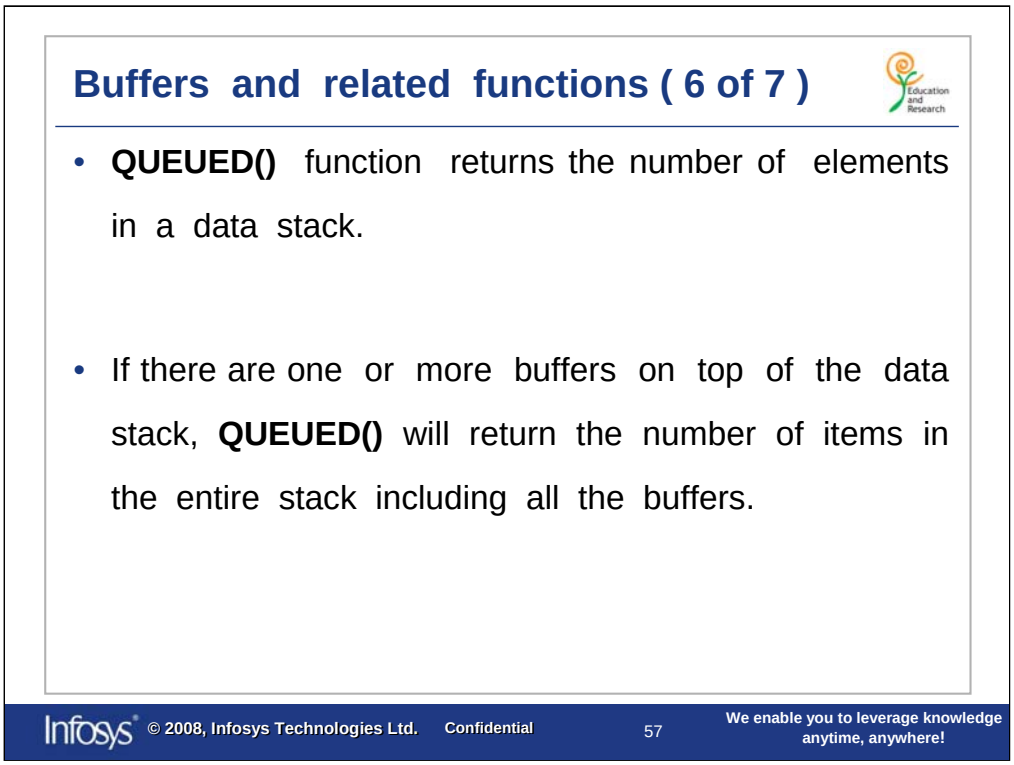
The MAKEBUF command will not protect elements below the buffer . If the elements in a buffer have all been accessed , the exec will automatically drop the buffer and proceed to access elements added before the MAKEBUF command was issued . We shall look at an example in the next slide .



Buffers and related functions (5 of 7) - Notes

In the above code snippet, 10 will be pushed first onto the data stack . Next 20 will be added to the bottom of the data stack by virtue of the QUEUE instruction . Next 30 is added to the top of the data stack by the PUSH instruction .

We then create a buffer by the MAKEBUF command . We now proceed to add 40 to the buffer . In the next instruction, we add 50 to the bottom of the buffer by virtue of the last QUEUE instruction . We then have three PULL instructions. The first PULL instruction removes 40 from the buffer . The next PULL instruction removes 50 from the buffer . Note that the buffer is now empty and has no elements in it . The next time the PULL instruction is issued , the buffer is automatically dropped and the topmost element in the data stack i.e. 30 is removed from the data stack.



Buffers and related functions (6 of 7) - Notes

Earlier we saw that the QUEUED() function returns the number of elements in a data stack. Please note that if we create one or more buffers on top of the data stack, QUEUED() will return the number of items in the entire stack including all the buffers.

The QUEUED() function will return the total number of items in the entire stack including all the buffers . For example , if the data stack has 3 elements and there are 2 buffers on top of the data stack , each with 5 elements in it , then the QUEUED() function will return 3 + 5+ 5 = 13 .

Buffers and related functions (7 of 7)

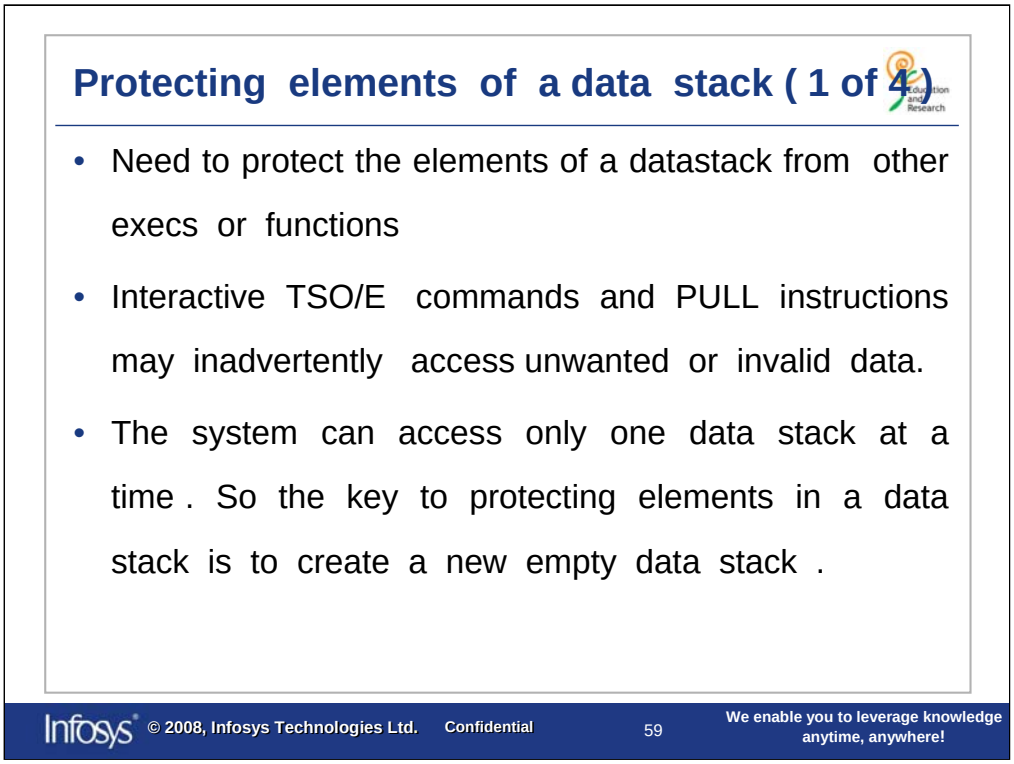
- **QBUF** : command to find out the total number of buffers
- **QELEM** : command to find out the number of elements contained in the most recent buffer.
- Note : The result of QBUF as well as QELEM will be stored in the special variable RC

Buffers and related functions (7 of 7) - Notes

There are occasions where we might need to know the following :

a) How many buffers are there ? E.g. Suppose an exec wants to drop all the buffers so as to access the elements in the original data stack . Therefore it would need to know how many buffers are there. The QBUF command gives the number of existing buffers . The result of the QBUF command is returned in the REXX special variable RC

b) How many elements are there in the most recent buffer ? E.g. Suppose we want our exec to process only the elements in the most recent buffer . So we need to know how many elements are there in the most recent buffer. The QELEM command gives the number of elements contained in the most recent buffer . The result of the QELEM command is also returned in the REXX special variable RC.



Protecting elements of a data stack (1 of 4) - Notes

Elements of a data stack may need to be protected from other execs or from other functions . If we do not protect the elements of a data stack , interactive TSO/E commands and PULL instructions may inadvertently access unwanted or invalid data . This is because as we have already discussed before , PULL instructions and interactive TSO/E commands first look for input in the data stack . Only if the data stack is empty, it will look at the terminal for input. Now if the data stack contains some invalid or irrelevant data, then the PULL instruction or interactive TSO/E command will access that invalid data. This is why we need to protect the elements of a data stack usually from other execs or functions so that PULL instructions and interactive TSO/E commands do not inadvertently access unwanted or invalid data.

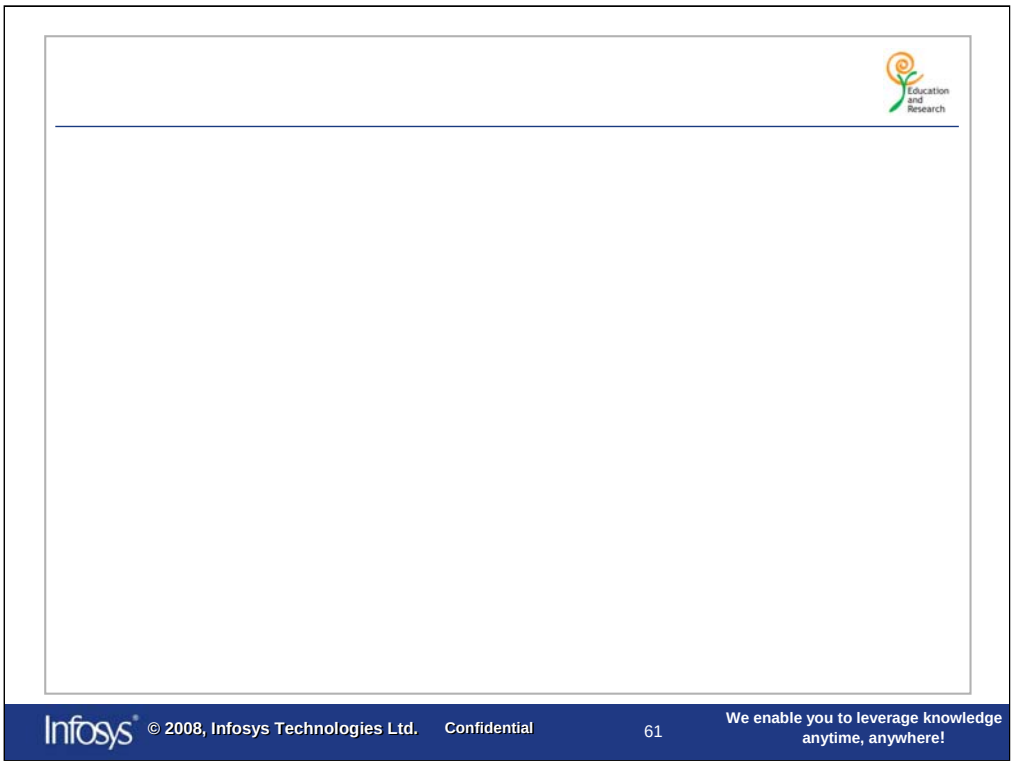
The system can access only one data stack at a time .The key to protecting elements in a data stack is to create a new empty data stack. We shall discuss more on this in the next few slides.

Protecting elements of a data stack (2 of 4)

- Creating a new data stack with nothing in it will effectively “lock” the elements in the original data stack.
- An interactive TSO/E command or PULL instruction will first look at this newly created data stack for input .
- On finding an empty data stack, it will proceed to get the input from the terminal

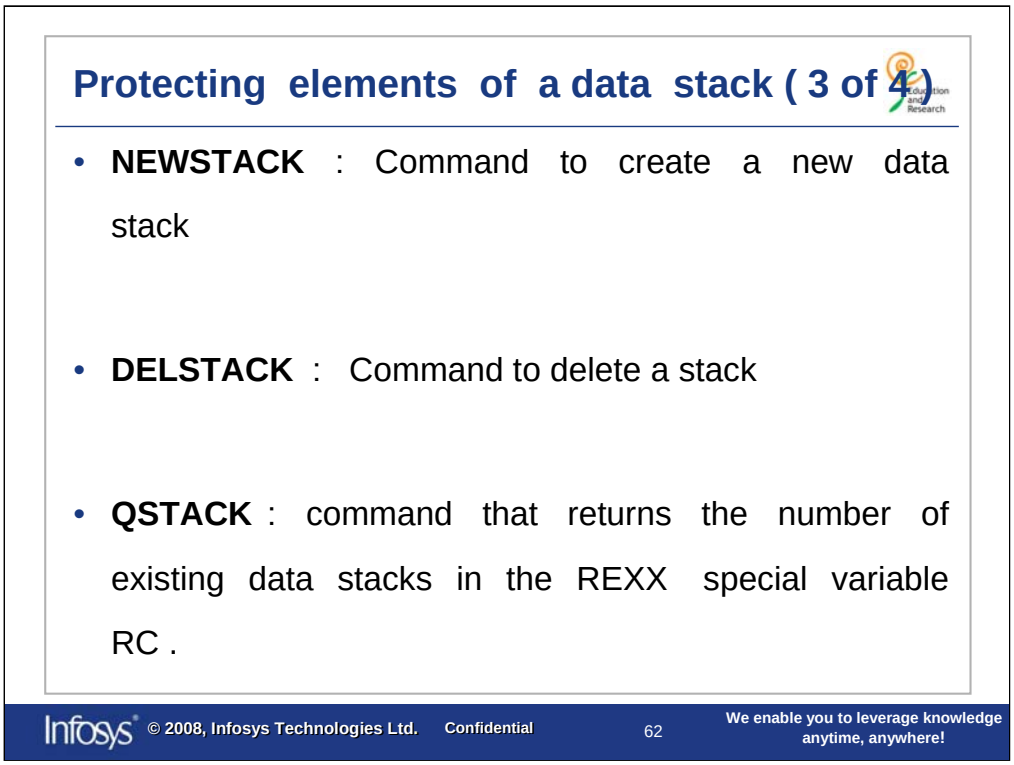
Protecting elements of a data stack (2 of 4) - Notes

The key to remember is that the system can access only one data stack at a time. Now suppose we create a new data stack with nothing in it . When an interactive TSO/E command is issued , it will first look at this newly created data stack for input. On finding an empty data stack, it will proceed to get the input from the terminal. Thus by creating a new data stack (with nothing in it), we can effectively “lock” or protect the elements in the original data stack.



Thus we have seen that the key to protecting elements on a data stack is the fact that the system can access only one data stack at a time. If we create a new empty data stack and an interactive TSO/E command or PULL instruction is issued, then the system first looks at this newly created data stack for input. On finding it empty, it proceeds to obtain input from the terminal. This effectively "locks" or protects the elements on the original data stack because the interactive TSO/E commands or PULL instructions are prevented from inadvertently accessing invalid or unwanted data on the original stack.

We shall look at the commands for creating and deleting a new data stack in the next slide.



Protecting elements of a data stack (3 of 4) - Notes

The command to create a new data stack is NEWSTACK . The data stack that is newly created is completely isolated (or different) from the original data stack. Unless & until this new datastack is deleted by a DELSTACK command issued by the exec, elements on the original data stack cannot be accessed by any other exec or the routines it calls.

When a DELSTACK command is issued from an exec, it will erase all the items on the most recently created data stack and then delete the data stack. Once the most recently created data stack is deleted , the elements in the previous stack become accessible.

It is to be noted that if there is no new stack created using the NEWSTACK command, the DELSTACK command will remove all the elements of the original stack.

We may often come across situations where we need to know the number of existing data stacks . The QSTACK command will return the total number of existing data stacks in the REXX special variable RC.

Note : It is to be noted that QSTACK will return not only the number of data stacks created by the exec from which it was issued. QSTACK will return the total number of stacks in existence.

Protecting elements of a data stack (4 of 4)

- **Example** : Suppose the main exec routine creates two stacks and each of it's 3 subroutines create one stack each .

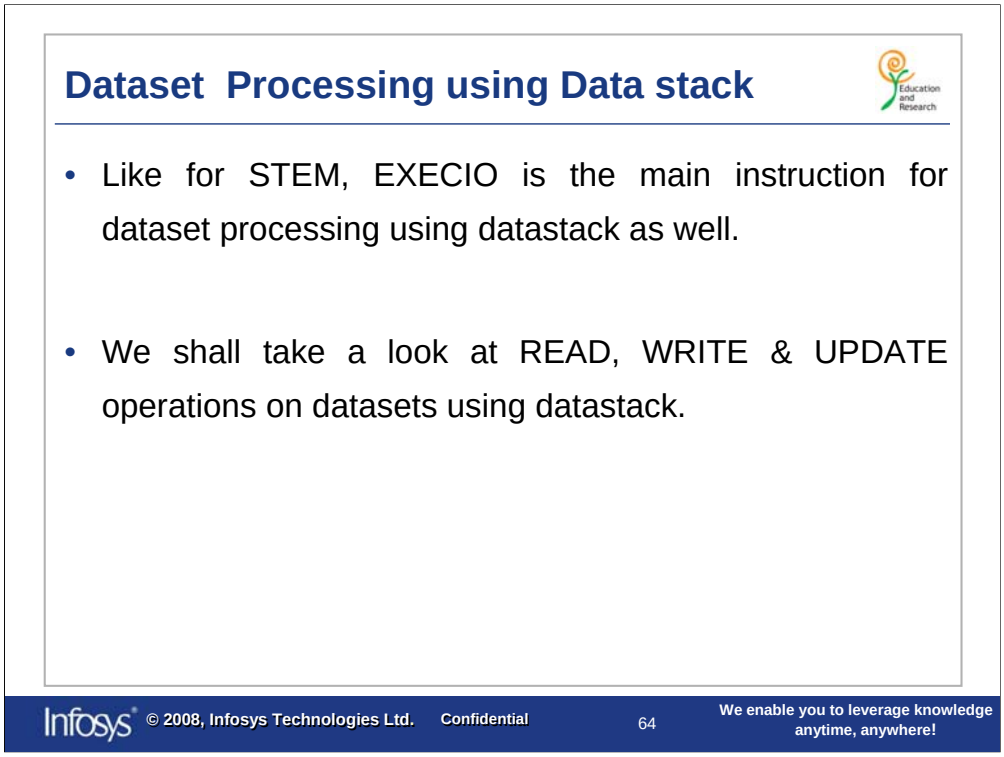
  What will be the return value from QSTACK command?

6

Protecting elements of a data stack (4 of 4) - Notes

In the above example , we assume the main exec to have created two stacks (i.e. It has issued the NEWSTACK command two times) . Suppose it has 3 subroutines , each of which creates one stack each. Now if we issue a QSTACK command, the value returned in the RC variable will be 6 regardless of where we have issued that QSTACK command.

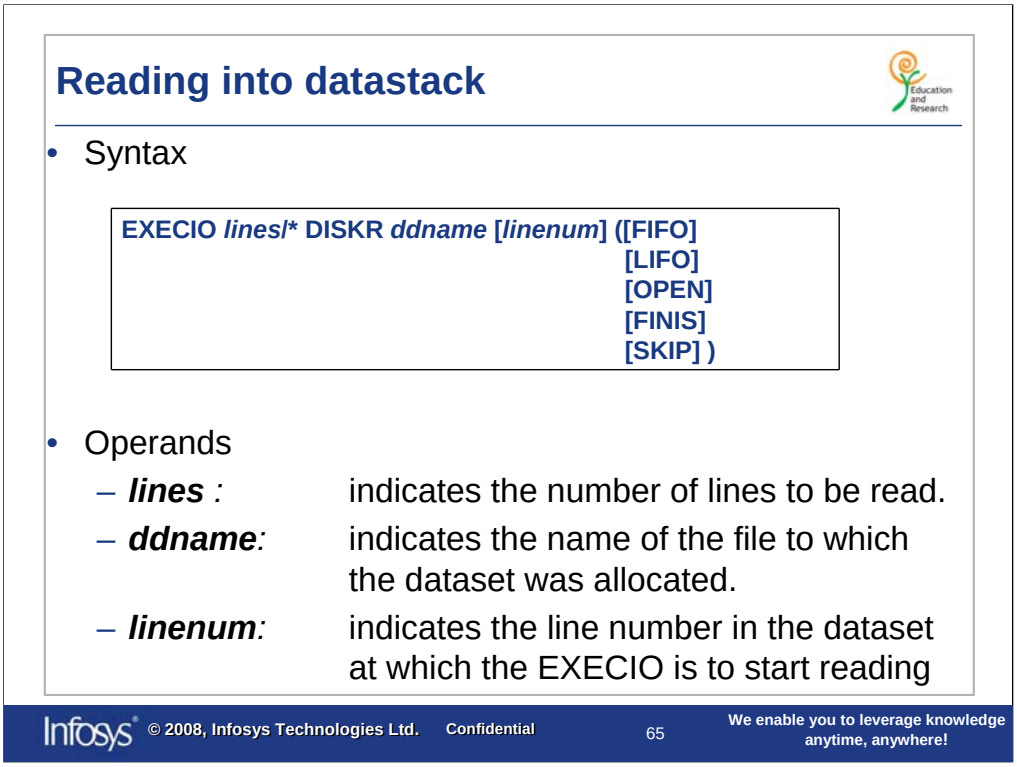
RC will have the value 6 and not 5 because one stack is automatically created by REXX and this is also taken into consideration by the QSTACK command.



Dataset Processing using Data stack - Notes

Like for STEM, EXECIO is the main instruction for dataset processing using datastack as well.

We shall take a look at READ, WRITE & UPDATE operations on datasets using datastack



Reading into datastack – Notes

EXECIO *lines*/* DISKR *ddname* [*linenum*] ([FIFO] [LIFO][OPEN][FINIS][SKIP])

Following are the read operands:

lines :

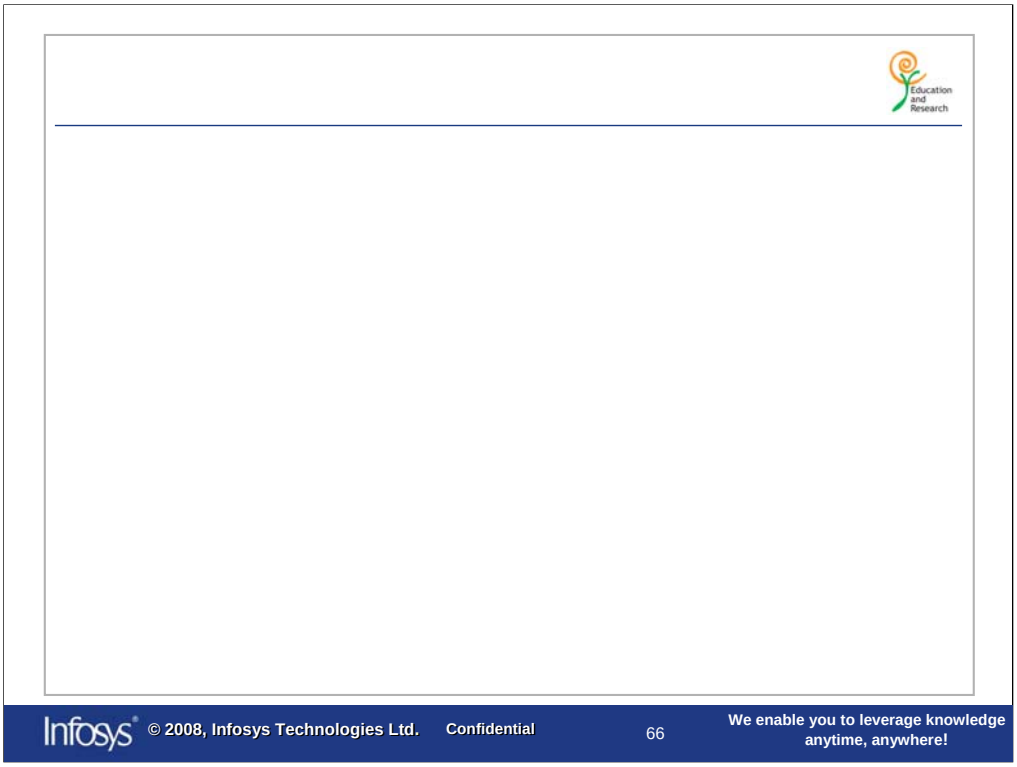
It indicates the number of lines to be read. To read all the lines, specify ***** instead of a number. If the value specified for *lines* is 0, no I/O operations are performed unless either OPEN or FINIS or both OPEN & FINIS are specified. In

such cases:

1. If OPEN is specified and the dataset is closed, EXECIO opens the dataset but will not read any lines. If OPEN is specified and the dataset is already open, then also EXECIO will not read any lines. In both the cases, if a non-zero value is specified for the *linenum* operand, EXECIO will set the current record number to the record number specified by the *linenum* operand.

NOTE: By default, the current record number will be set to the first record. Current record can be defined as the number of the next record EXECIO will read. However, if a non-zero value is specified for *linenum* , then EXECIO sets the current record number to the record number indicated by the *linenum* operand.

2. If FINIS is specified & the dataset is open, EXECIO will not read any lines, but it will close the dataset. If FINIS is specified & the dataset is closed already, then EXECIO will not do anything.
3. If both OPEN & FINIS are specified, then EXECIO will process OPEN first & then FINIS.



DISKR: This operand will open a dataset for input (if it is not already open) & will read the specified number of lines from the dataset.

ddname: This indicates the name of the file to which the dataset was allocated. The dataset can either be a sequential file or a member of a PDS. The file has to be allocated before issuing EXECIO. ALLOCATE command can be used for this.

linenum: This indicates the line number in the dataset at which the EXECIO has to start reading. When a dataset is opened for input or update, the current record number is the number of the next record which will be read by EXECIO. If a zero value is specified for *linenum,* then it is as though *linenum* has not been specified at all.

FINIS: This is used to close the dataset after the EXECIO command completes. It is a good programming practice to explicitly close the datasets using FINIS. If FINIS is not specified, dataset is closed when one of the following occurs:

a. The task under which the dataset was opened, is terminated

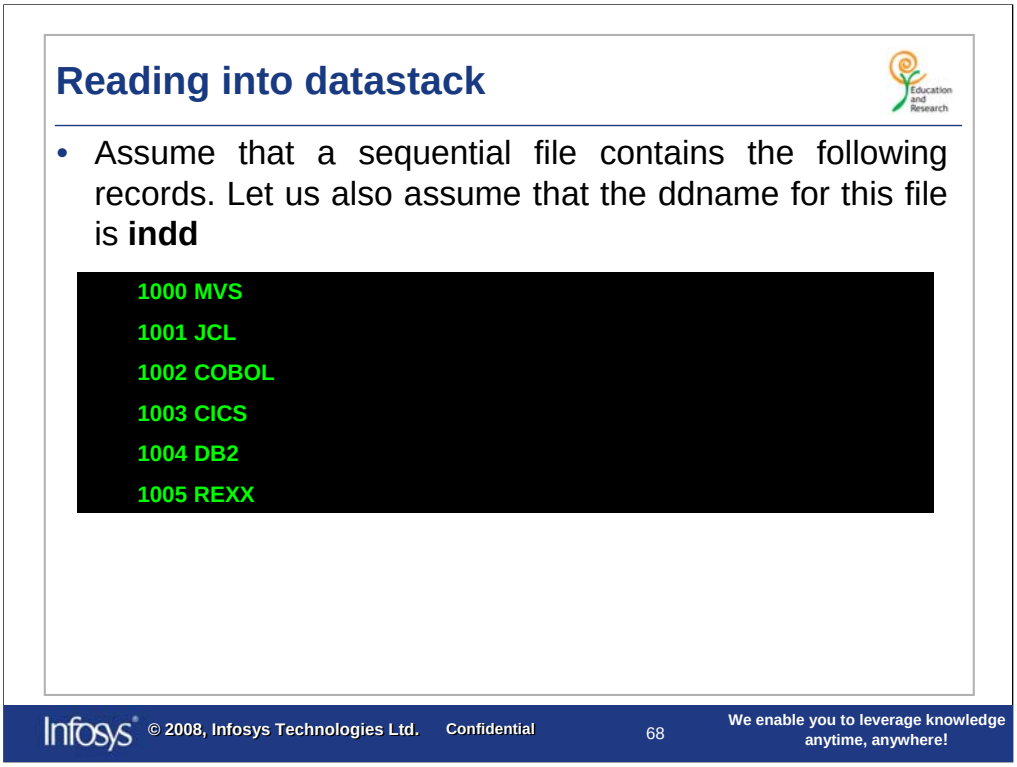
b. The language processor environment associated with the task, under which the dataset was opened, is terminated

OPEN: This is used to open the specified dataset if it is not open already.

FIFO: If FIFO is specified then, the records are placed in the datastack in First In First Out order. This is the default setting

LIFO: If LIFO is specified then, the records are placed in the datastack in Last In First Out order.

SKIP: If SKIP is specified, then the specified number of variables will be read but will not be placed in the datastack



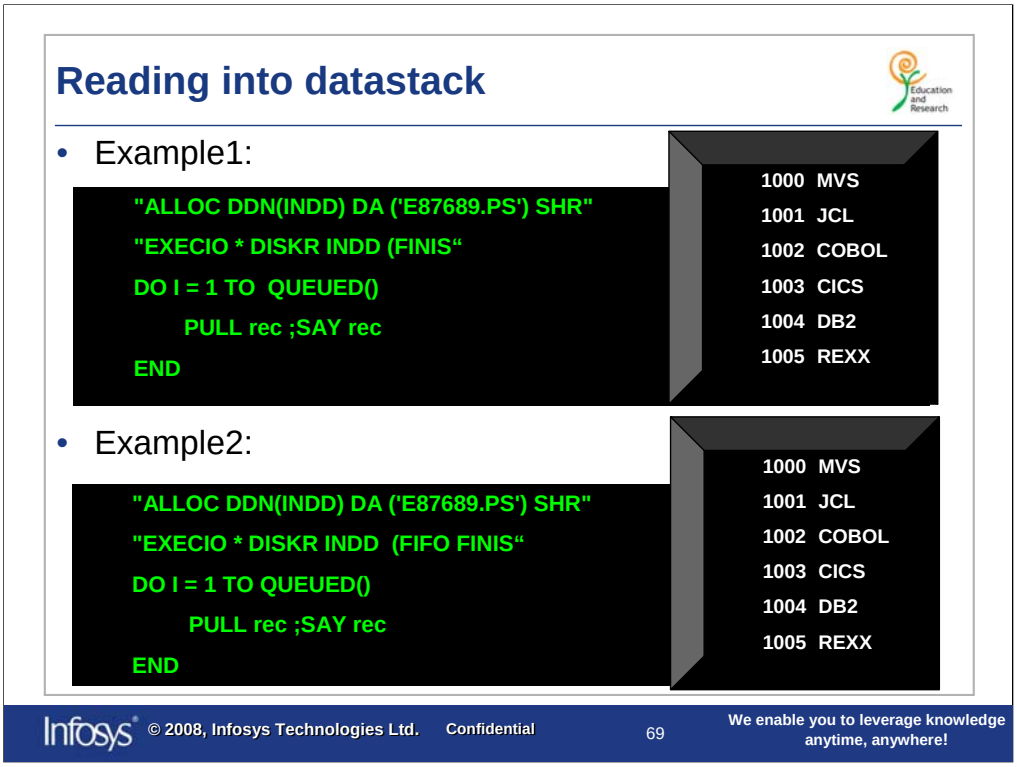
Reading into datastack – Notes

Assume that a sequential file contains the following records.

Let us also assume that the ddname for this file is **indd.** This ddname should be the same as specified in the ALLOCATE command.

Example for allocate: assuming that the sequential file name is userid.ps, below shown is the usage of allocate command.

“ALLOC DDN(indd) DA (‘userid.ps’) shr”



Reading into datastack – Notes

Example 1:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO * DISKR INDD (FINIS“
DO I = 1 TO  QUEUED()
        PULL rec ;SAY rec
END
```

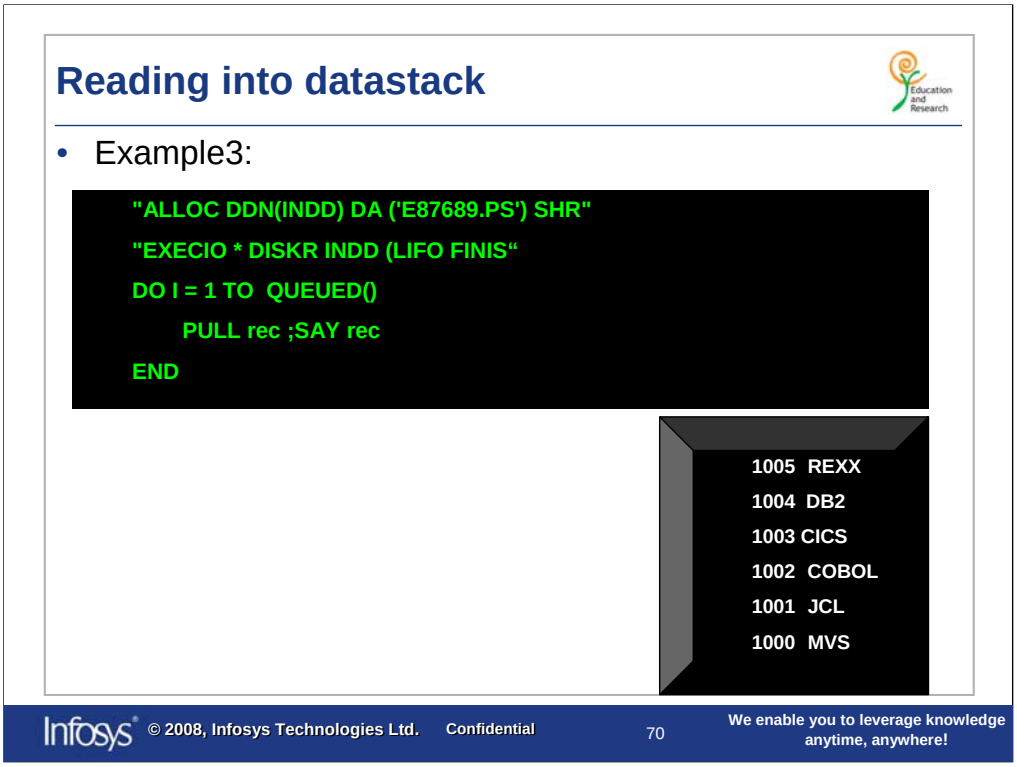
In the above example, the EXECIO statement specifies * for *lines.* Thus, all the records will be read from the dataset. Note that *linenum* has not been specified. Hence, all records starting from the first record will be read into datastack in FIFO order (as FIFO is the default).

Important thing to note is, when records are read into datastack, QUEUED() function will return the number of records read.

Example 2:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO * DISKR INDD  (FIFO FINIS“
DO I = 1 TO QUEUED()
         PULL rec ;SAY rec
END
```

In this example, FIFO is specified. Hence the output will be same as in example 1.



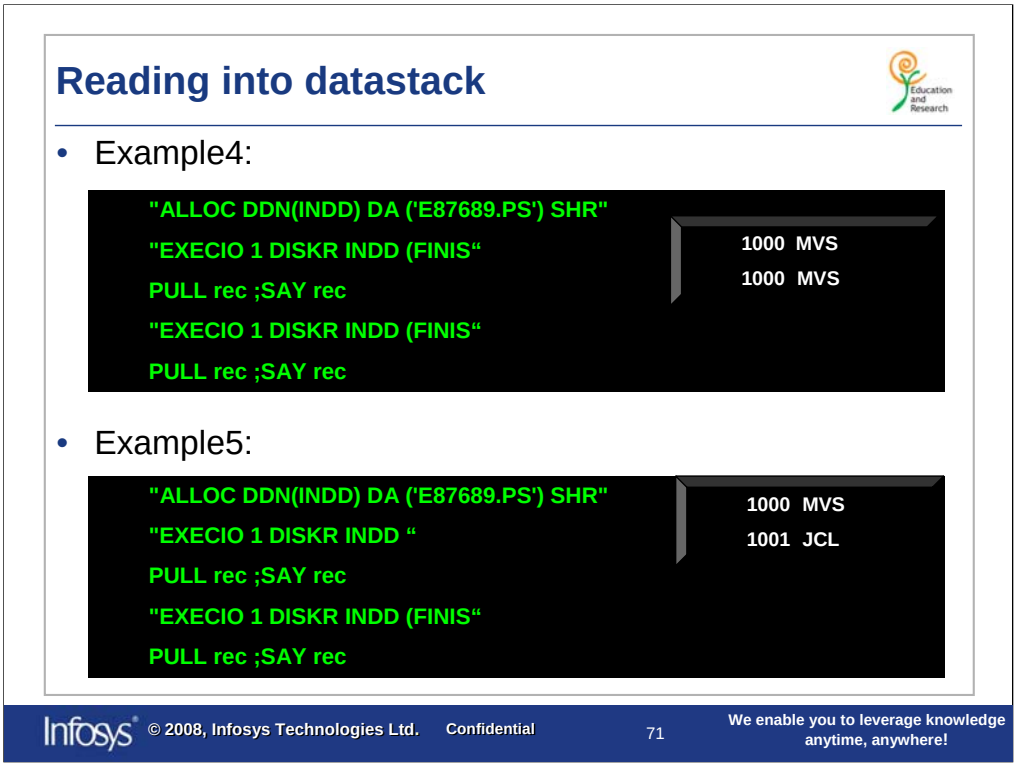
Reading into datastack – Notes

Example 3:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO * DISKR INDD (LIFO FINIS“
DO I = 1 TO  QUEUED()
        PULL rec ;SAY rec
END
```

In the above example, the EXECIO statement specifies * for *lines.* Thus, all the records will be read from the dataset. Note that *linenum* has not been specified. Hence, all records starting from the first record will be read into datastack in LIFO order.

Important thing to note is, when records are read into datastack, QUEUED() function will return the number of records read.



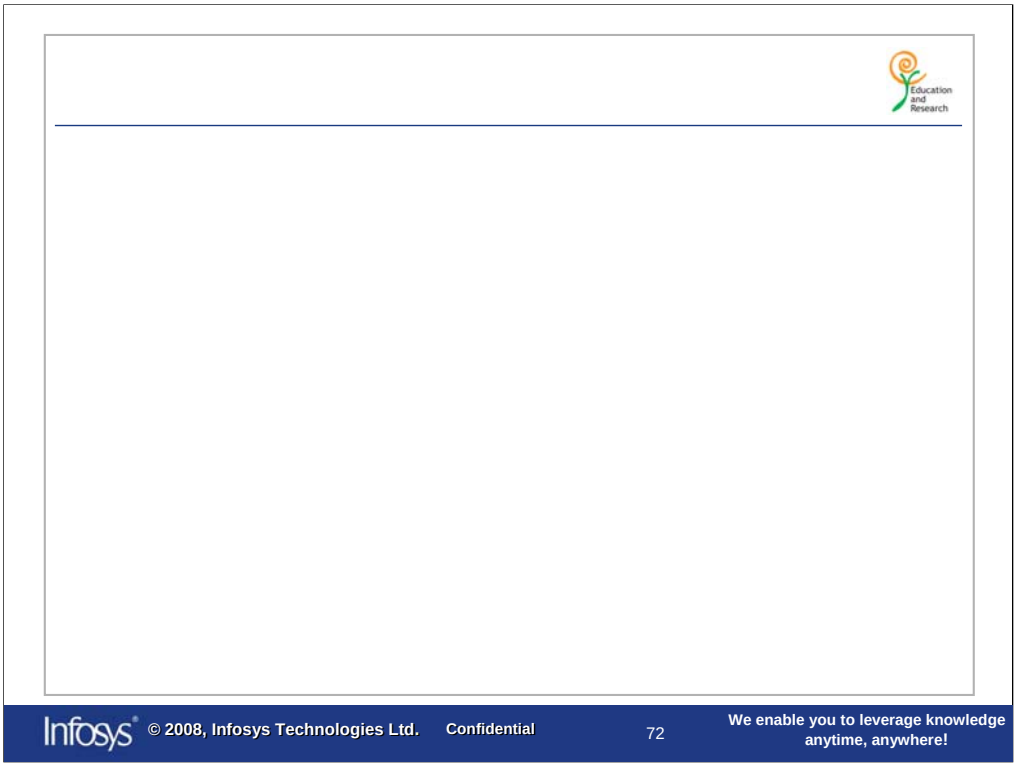
Reading into datastack – Notes

Example 4:

"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO 1 DISKR INDD (FINIS“
PULL rec ;SAY rec
"EXECIO 1 DISKR INDD (FINIS“
PULL rec ;SAY rec

In the above example, there are two EXECIO statements. Both of them read the same record.
The output will be:

1000 MVS
1000 MVS



Example 5:

"ALLOC DDN(INDD) DA ('E87689.PS') SHR"

"EXECIO 1 DISKR INDD “

PULL rec ;SAY rec

"EXECIO 1 DISKR INDD (FINIS“

PULL rec ;SAY rec

Output is:

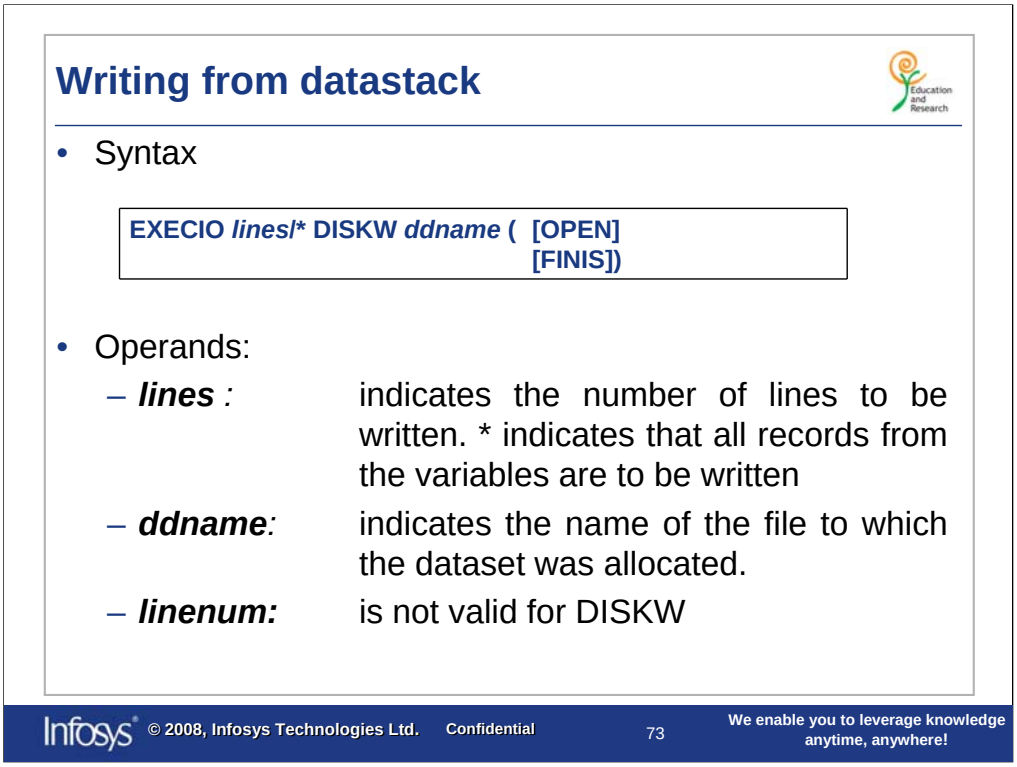
1000 MVS

1001 JCL

Even though both example 4 & 5 are similar, the outputs are different. What is missing in the 5th example???

In example 5, for the first EXECIO, FINIS is not specified, which means that the dataset is still open & the current record number after the execution of EXECIO will be set to 2 & hence the next EXECIO read statement will read the second record.

In example 4, for both EXECIO statements, FINIS is specified. After the execution of first EXECIO, dataset will be closed & current record number will be set to 0. The second EXECIO statement, however, sets the current record number to 1 & hence the same record is read by both the EXECIO statements.



Writing from datastack – Notes

Syntax:

EXECIO *lines*/* DISKW *ddname* ([OPEN][FINIS])

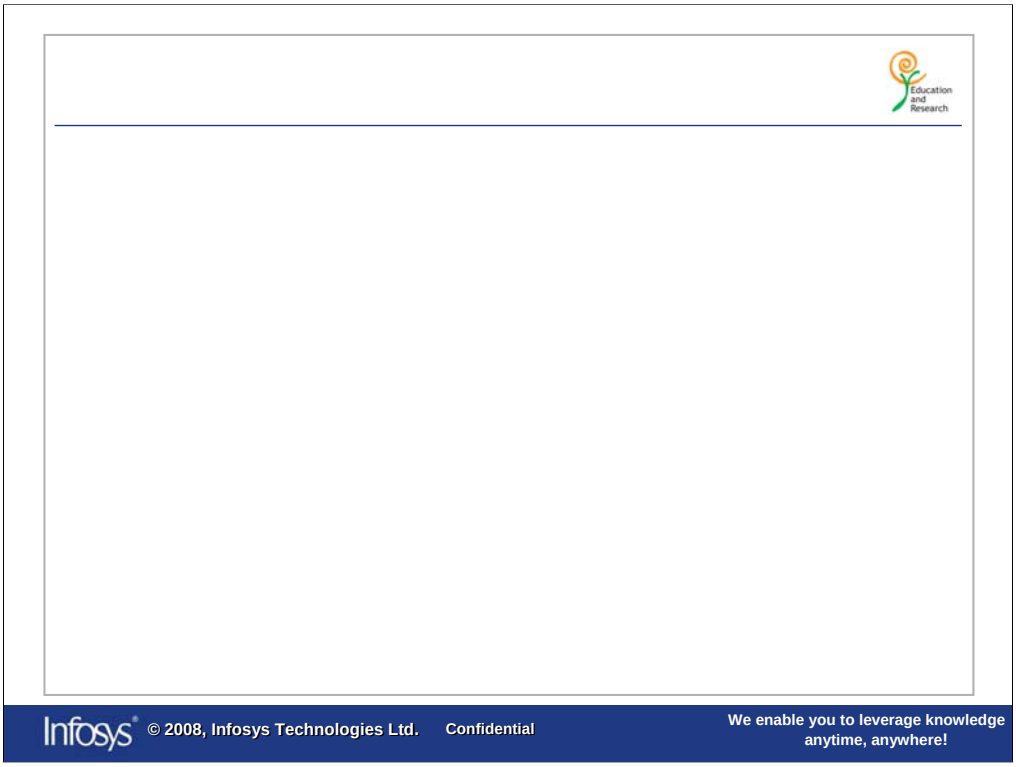
Following are the write operands:

lines :

It indicates the number of lines to be written. If the value specified for *lines* is 0, no I/O operations are performed unless either OPEN or FINIS or both OPEN & FINIS are specified. In

such cases:

1. If OPEN is specified and the dataset is closed, EXECIO opens the dataset but will not write any lines. If OPEN is specified and the dataset is already open, then also EXECIO will not write any lines.
2. If FINIS is specified & the dataset is open, EXECIO will not write any lines, but it will close the dataset. If FINIS is specified & the dataset is closed already, then EXECIO will not do anything.
3. If both OPEN & FINIS are specified, then EXECIO will process OPEN first & then FINIS.

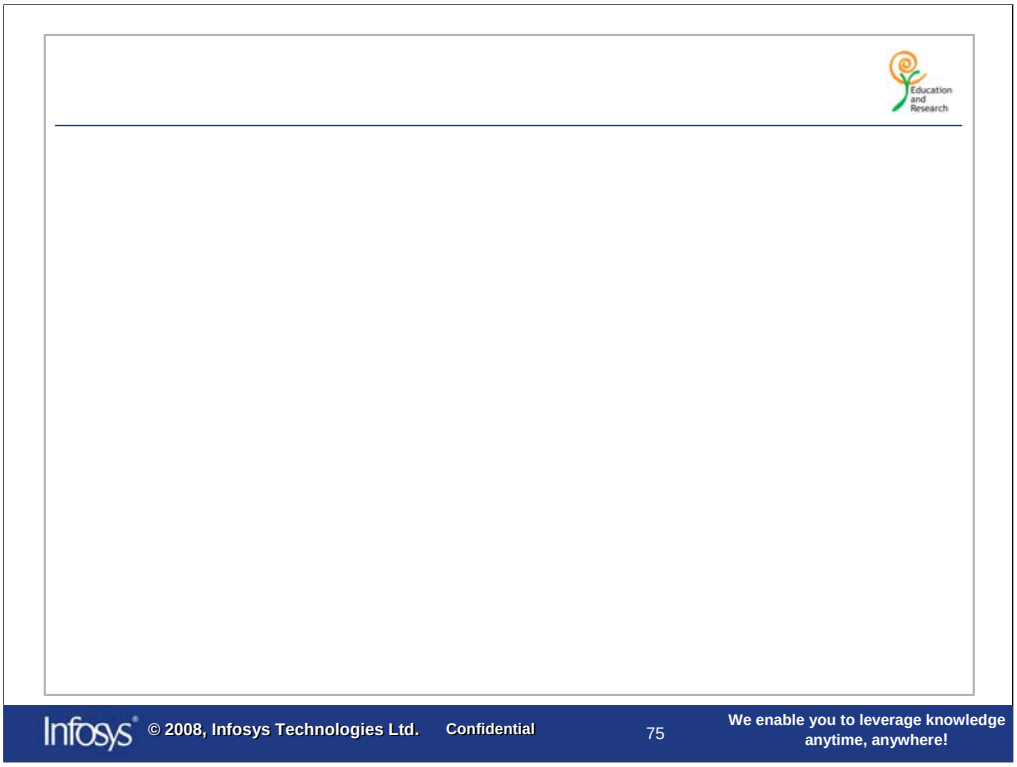


Note: When * is specified for *lines,* EXECIO will continue writing records from the datastack until it reaches a null line. If there is no null line in the datastack in an interactice TSO/E address space, EXECIO will wait for an input from the terminal & will stop only when a null line is received.

When the exec is running in TSO/E background or non-TSO/E address space & EXECIO * DISKW is specified & if the datastack becomes empty before a null line is reached, then EXECIO will look for data in the input stream.

DISKW: This operand will open a dataset for output (if it is not already open) & will write the specified number of lines to the dataset. DISKW is also used to rewrite the last read record in case of updation.

ddname: This indicates the name of the file to which the dataset was allocated. The dataset can either be a sequential file or a member of a PDS. The file has to be allocated before issuing EXECIO. ALLOCATE command can be used for this.



FINIS: This is used to close the dataset after the EXECIO command completes. It is a good programming practice to explicitly close the datasets using FINIS. If FINIS is not specified, dataset is closed when one of the following occurs:

a. The task under which the dataset was opened, is terminated

b. The language processor environment associated with the task, under which the dataset was opened, is terminated

To close an open dataset without writing any record into a dataset, specify *lines* as 0 & also specify FINIS.

OPEN: This is used to open the specified dataset if it is not open already.

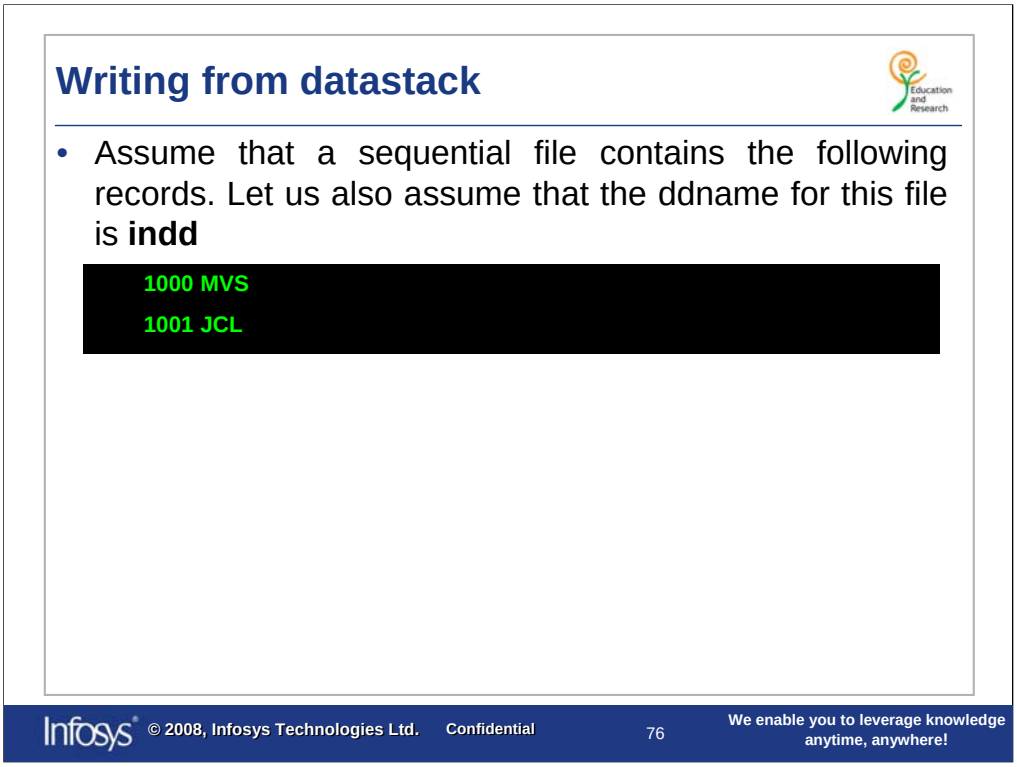
To open a closed dataset without writing any record into the dataset, specify *lines* as 0 along with OPEN.

If records are to appended to the existing set of records then ALLOCATE a dataset using MOD parameter.

If records are to overwritten then ALLOCATE a dataset using SHR parameter.

Note:

SKIP & ***linenum*** are not valid operands for write operation.



Writing from datastack – Notes

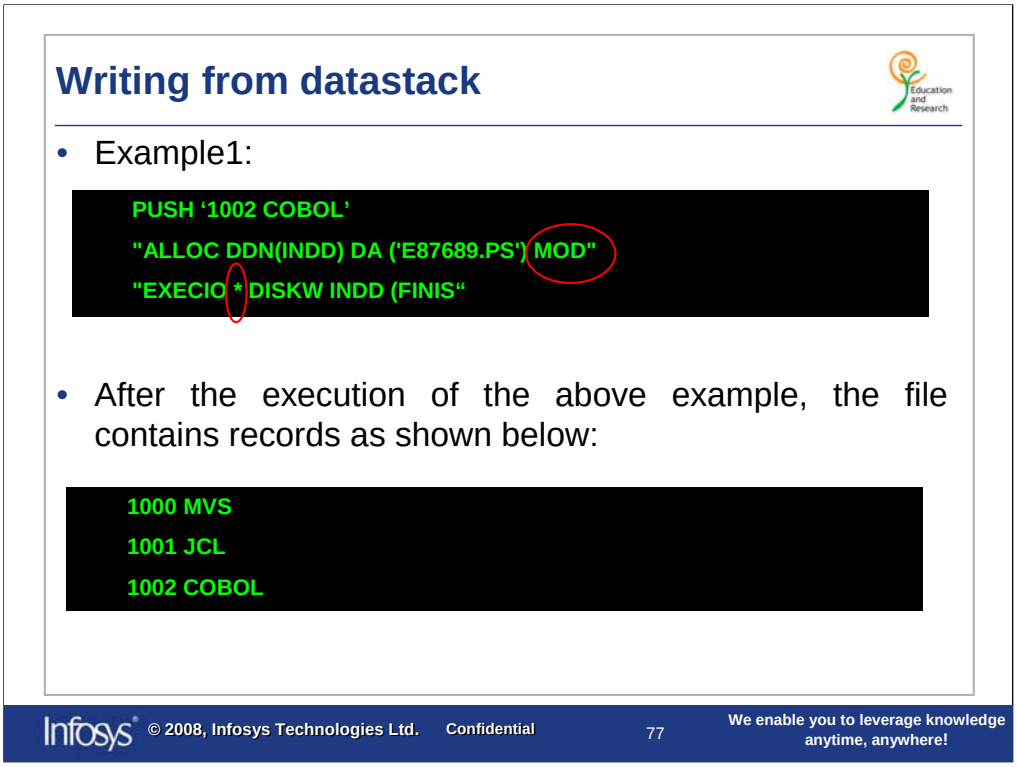
Assume that a sequential file contains the following records.

Let us also assume that the ddname for this file is **indd.** This ddname should be the same as specified in the ALLOCATE command.

Example for allocate: assuming that the sequential file name is userid.ps, below shown is the usage of allocate command.

"ALLOC DDN(indd) DA ('userid.ps') SHR" (if records are to be overwritten)

"ALLOC DDN(indd) DA ('userid.ps') MOD" (if records are to be appended to the existing set of records)



Writing from datastack – Notes

Example 1:

PUSH ‘1002 COBOL’

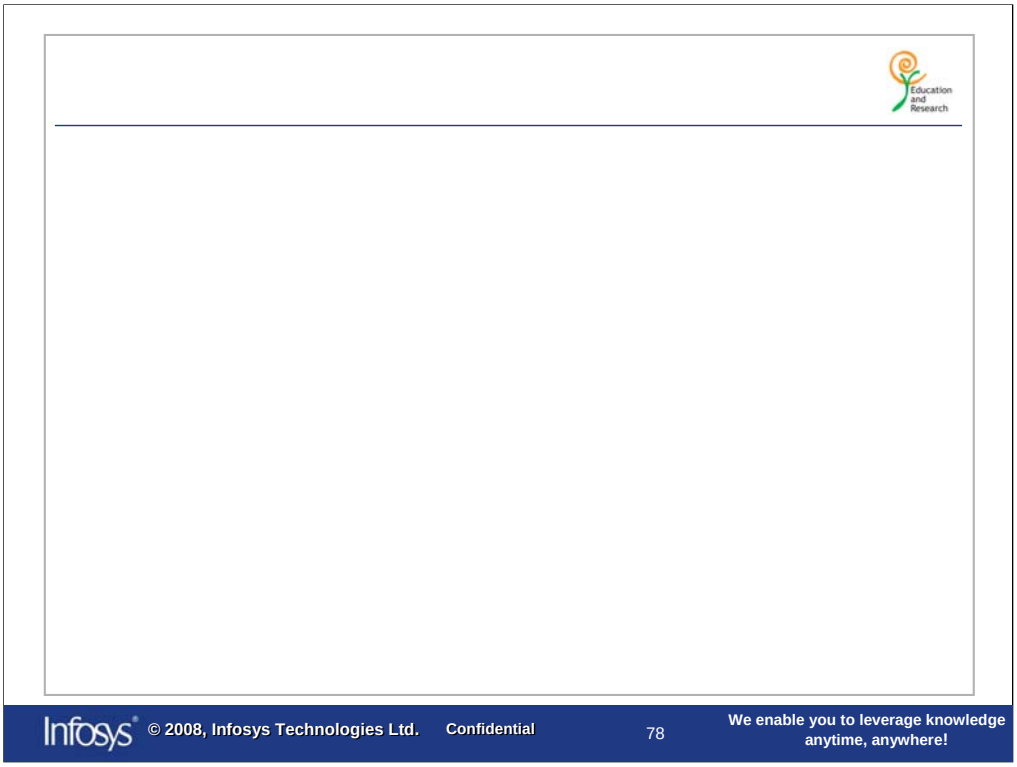
"ALLOC DDN(INDD) DA ('E87689.PS') MOD"

"EXECIO * DISKW INDD (FINIS“

Here, dataset is allocated with MOD parameter. Hence record ‘1002 COBOL’ will be appended with the existing 2 records.

Note:

In this case, the datastack will contain only ‘1002 COBOL’. It does not contain a null line. Hence, EXECIO will wait for terminal input from the user. If this input is a null, then only ‘1002 COBOL’ is appended & the exec terminates. However, if the user specifies another record as input from the terminal, then that record will also be appended with the existing records. Thus, if we do not put a null line into the datatstack, then the EXECIO will ask for input from the terminal. Let us look at the effect of not putting a null line into the datastack in the following cases:



Case 1: When EXECIO * DISKW is specified.

In this case, all the lines from the datastack are written into the dataset & then EXECIO will look for input from the terminal. It will keep accepting the input from the terminal until a null line is given as input.

Example:

PUSH '1002 COBOL'

"ALLOC DDN(INDD) DA ('E87689.PS') MOD"

"EXECIO * DISKW INDD (FINIS"

After '1002 COBOL' is written into the dataset, EXECIO will look for input from the terminal.

Let us assume that the user enters '1003 CICS'

'1004 DB2'

'' (null)

In this case, the file would contain the following records:

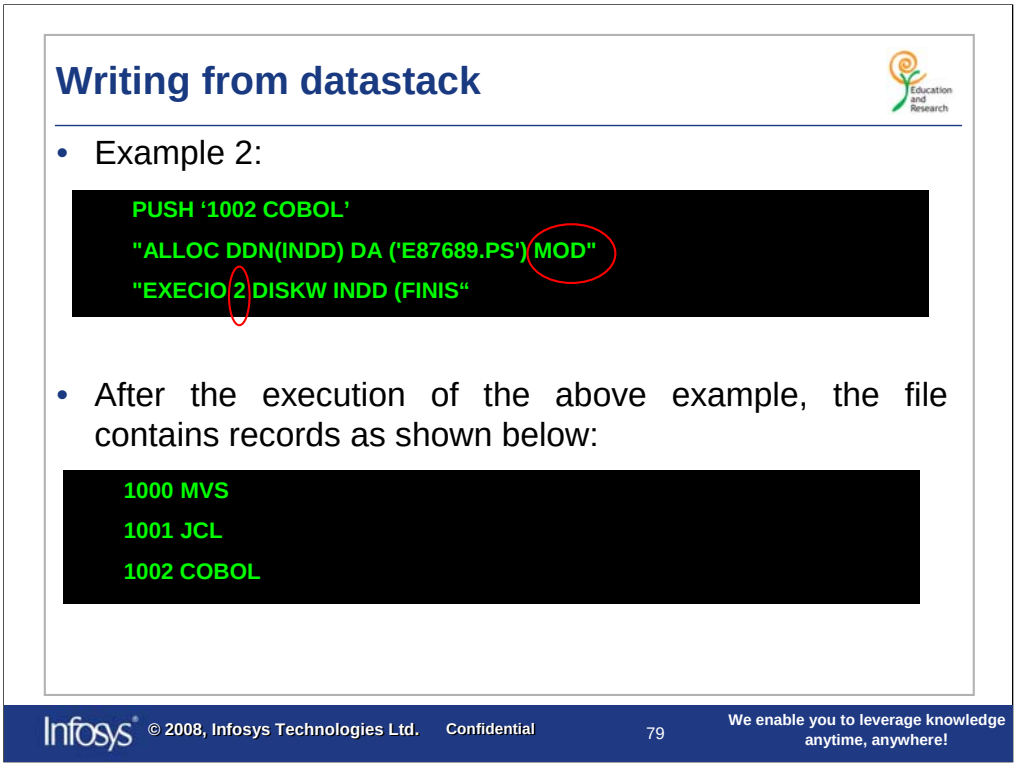
1000 MVS

1001 JCL

1002 COBOL

1003 CICS

1004 DB2



Writing from datastack – Notes

Case 2: When EXECIO *linenum* DISKW is specified and *linenum* specified is greater than the actual number of lines in the datastack

Example 2:

PUSH ‘1002 COBOL’

"ALLOC DDN(INDD) DA ('E87689.PS') MOD"

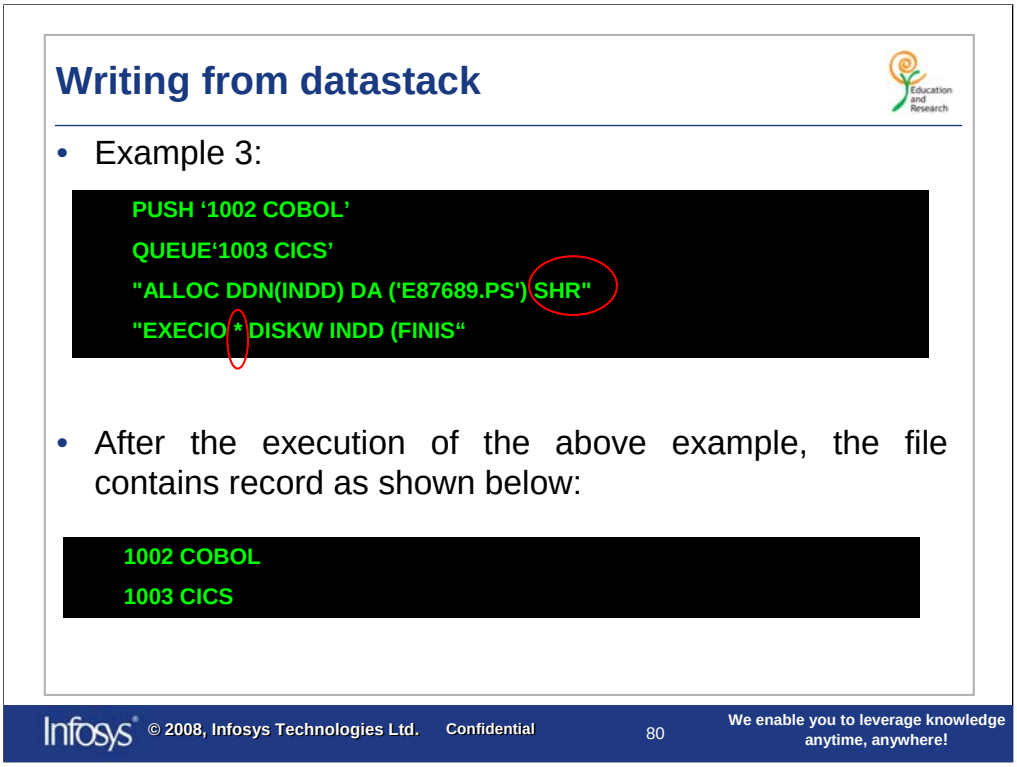
"EXECIO 2 DISKW INDD (FINIS“

Here, dataset is allocated with MOD parameter. Hence record ‘1002 COBOL’ will be appended with the existing 2 records.

Note:

In this case, the datastack will contain only ‘1002 COBOL’. It does not contain a null line. Hence, EXECIO will wait for terminal input from the user. If this input is a null, then only ‘1002 COBOL’ is appended & the exec terminates.

The value specified for *linenum* is 2 which is greater than the actual number of lines in the datastack. In this case, the EXECIO will wait for only one more input (the difference between *linenum* specified and the actual number of lines in the datastack, in this case 2-1 = 1). If the user enters some record, then that record will also be written into the dataset. But if the user enters nothing (just presses ENTER key), then a blank line will be written.



Writing from datastack – Notes

Example 3:

PUSH ‘1002 COBOL’
QUEUE‘1003 CICS’
"ALLOC DDN(INDD) DA ('E87689.PS') SHR"
"EXECIO * DISKW INDD (FINIS“

Here, dataset is allocated with SHR parameter. Hence records ‘1002 COBOL’ & ‘1003 CICS’ will overwrite the existing 2 records.

Updating a Record

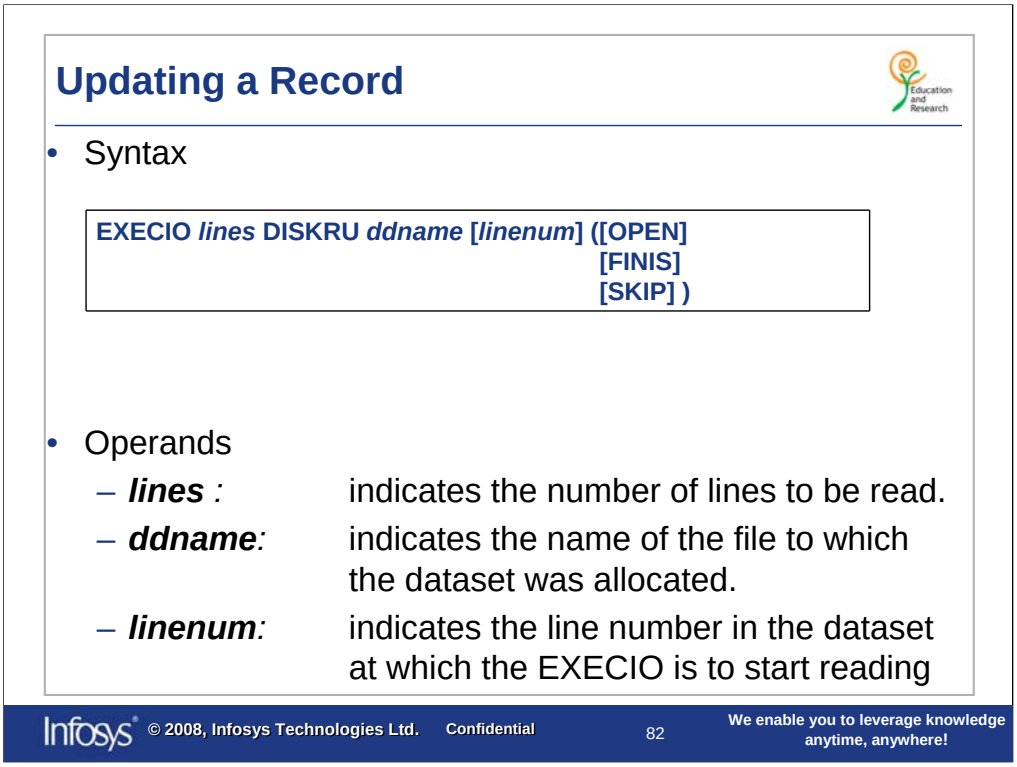
- To update a record, EXECIO command has to be used twice:
  - First read the record to be updated with update option (DISKRU)
  - Write the updated record into the file using DISKW

Updating a Record – Notes

To update a record, EXECIO command has to be used twice:

First read the record to be updated with update option (DISKRU)

Write the updated record into the file using DISKW



Updating a Record – Notes

EXECIO *lines* DISKRU *ddname* [*linenum*] (STEM var-name [OPEN][FINIS][SKIP])

Operands

lines *:*

It indicates the number of lines to be read.

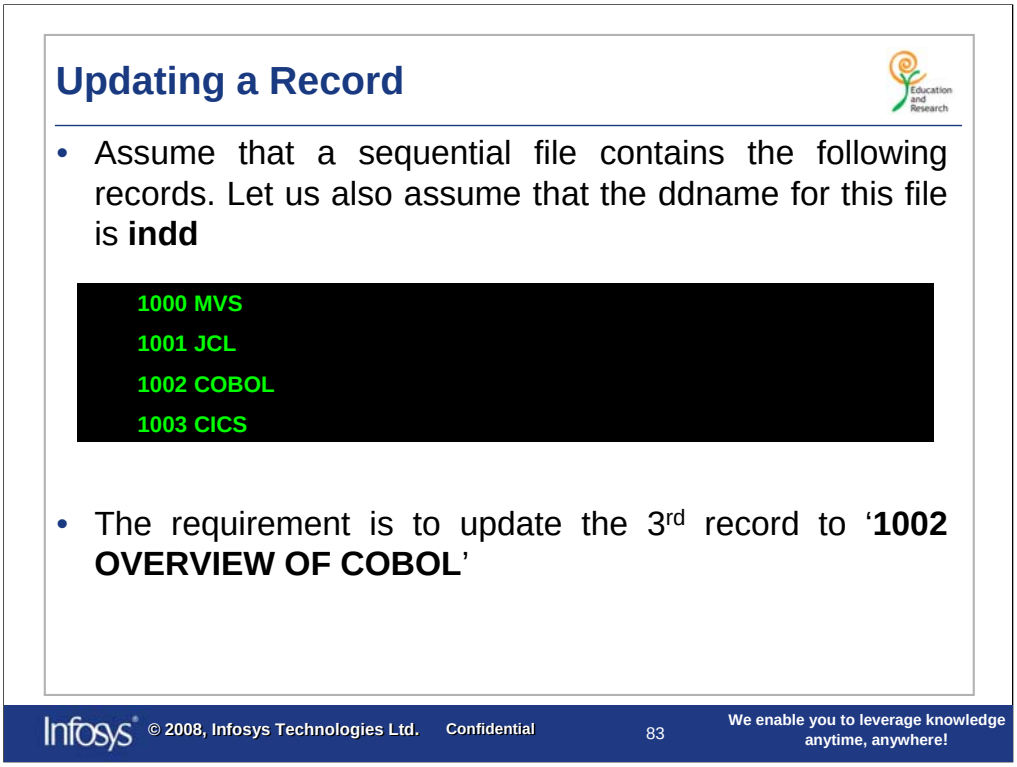
ddname*:*

It indicates the name of the file to which the dataset was allocated.

linenum*:*

It indicates the line number in the dataset at which the EXECIO is to start reading

DISKRU: Opens the dataset for update (if not already open) and places the number of records specified in the datastack. When a dataset is opened for update, the last read record can be modified & written back using the EXECIO DISKW.



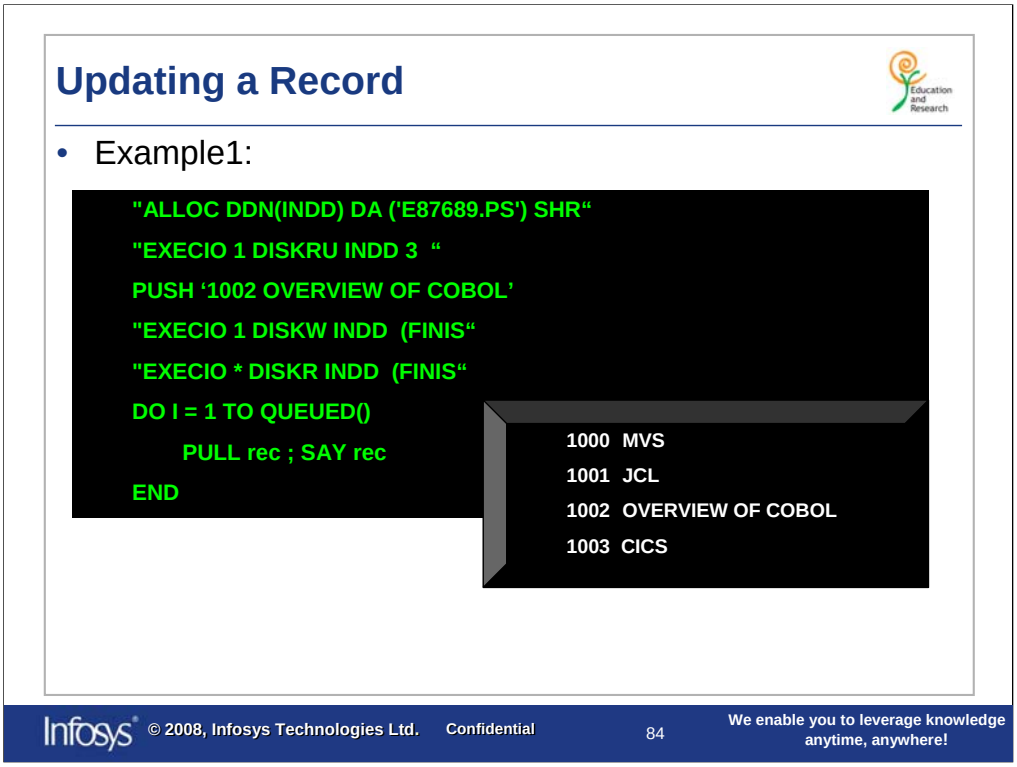
Updating a Record – Notes

Assume that a sequential file contains the following records.

Let us also assume that the ddname for this file is **indd.** This ddname should be the same as specified in the ALLOCATE command.

Example for allocate: assuming that the sequential file name is userid.ps, below shown is the usage of allocate command.

"ALLOC DDN(indd) DA ('userid.ps') shr"



Updating a Record – Notes

Example 1:

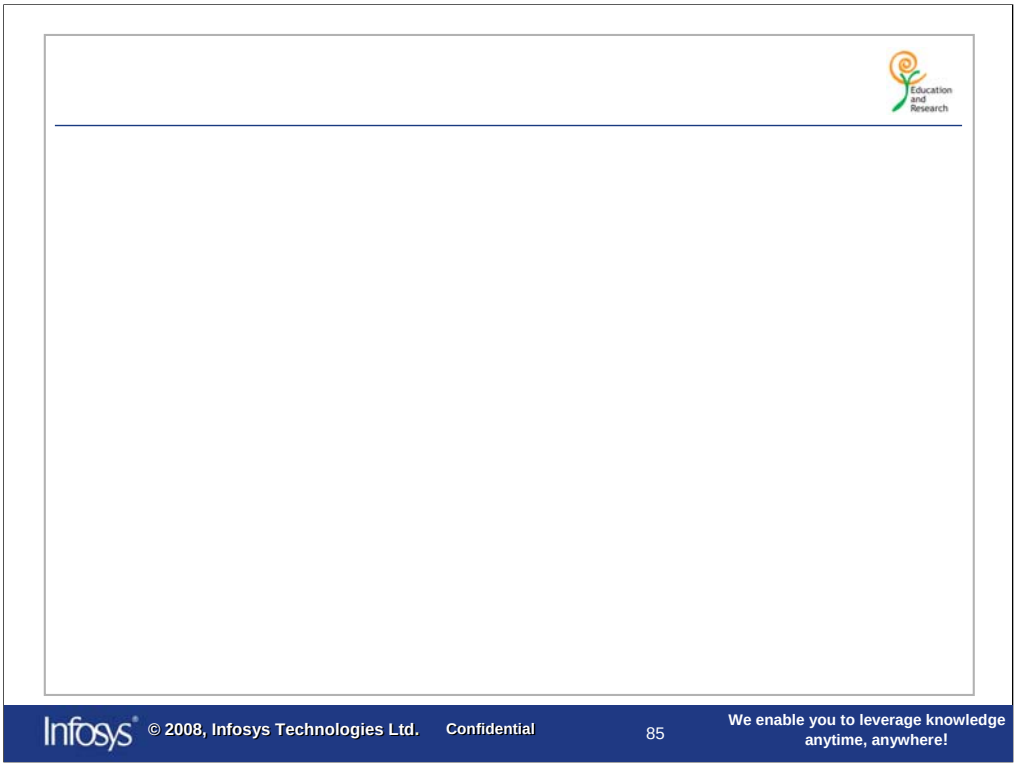
```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR“
"EXECIO 1 DISKRU INDD 3 “
PUSH ‘1002 OVERVIEW OF COBOL’
"EXECIO 1 DISKW INDD (FINIS“
"EXECIO * DISKR INDD (FINIS“
DO I = 1 TO QUEUED()
        PULL rec ; SAY rec
END
```

The output will be

```
1000    MVS
1001    JCL
1002    OVERVIEW OF COBOL
1003  CICS
```

Here, the 3rd record will be updated.

In the above example, FINIS is not specified for the EXECIO DISKRU statement.



Example 2:

```
"ALLOC DDN(INDD) DA ('E87689.PS') SHR“
"EXECIO 1 DISKRU INDD 3  (FINIS “
PUSH ‘1002 OVERVIEW OF COBOL’
"EXECIO 1 DISKW INDD  (FINIS“
"EXECIO * DISKR INDD  (FINIS“
DO I = 1 TO QUEUED()
        PULL rec ; SAY rec
END
```

The output will be

```
1002     OVERVIEW OF COBOL
```

If FINIS is specified in the EXECIO DISKRU statement, then the dataset will be closed after the execution of the statement & instead of updating a record, it will execute the DISKW as an independent write statement & record will be either appended or overwritten (based on SHR or MOD). In the above example, mode is SHR and hence the records will be overwritten.

REXX Host Environments (1 of 5)

- The default command environment is TSO. All commands are sent to TSO for processing, unless specified otherwise in the exec .
- Some other available host environments are ISPEXEC, MVS, ISREDIT etc. An installation can add or delete environments.

REXX Host Environments (1 of 5) - Notes

Various operating (or host environments) are available to REXX . It is possible to perform a task in one environment and then easily move to another environment within the same exec. The default command environment is TSO. Unless an exec specifies a different environment, all commands are sent to TSO for processing. Some other available host environments are ISPEXEC, MVS etc . An installation can add or delete environments.

REXX Host Environments (2 of 5)

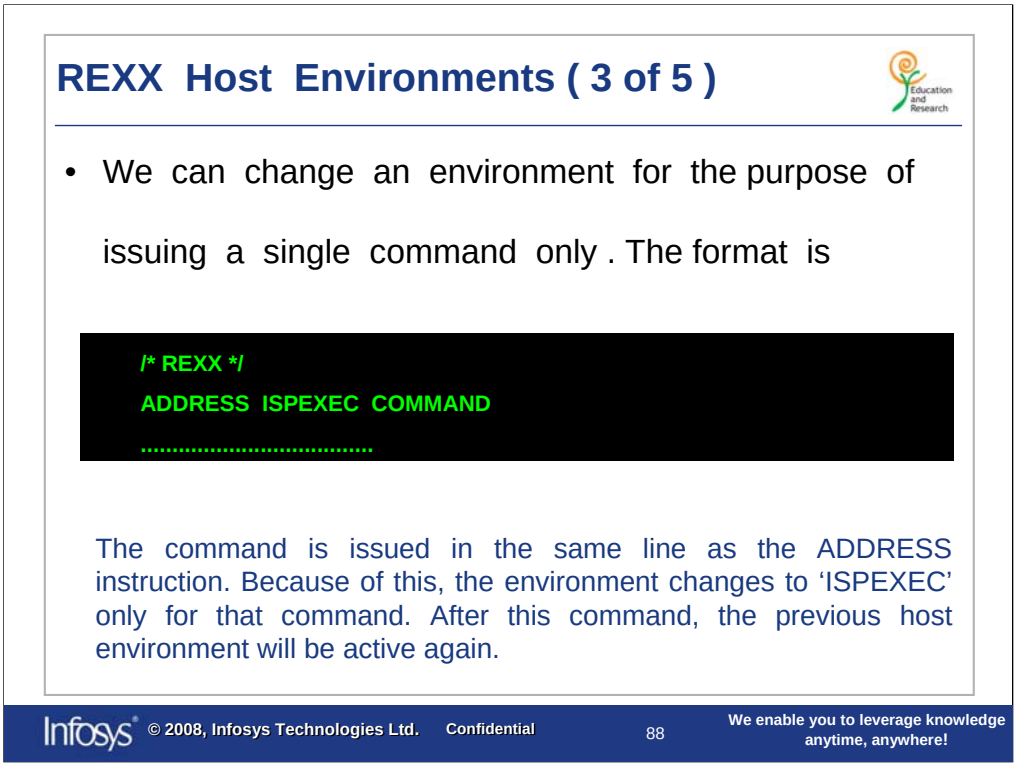
- Changing the host environment : use the **ADDRESS** instruction followed by the name of the new environment .

```
/* REXX */
ADDRESS  ISPEXEC
COMMAND
```

The command is issued in a new line after the ADDRESS instruction. Because of this, the same 'ISPEXEC' environment will be active until another ADDRESS instruction changes the environment.

REXX Host Environments (2 of 5) - Notes

A single exec can issue commands in more than one environment . To do this , it is obvious that we need to be able to change the host environment at any time . For changing the host environment, the ADDRESS instruction can be used followed by the name of the new environment . In the above example, ADDRESS ISPEXEC changes the host environment to ISPF . The command appears on the next line . We will now continue in ISPF till we issue a new host environment command .



REXX Host Environments (3 of 5) - Notes

We can also change a host environment for the purpose of issuing a single command only . As shown in the slide, in this case, the command follows the environment call on the same line. After that (i.e. From the next line onwards), the former host environment becomes active again.

REXX Host Environments (4 of 5)

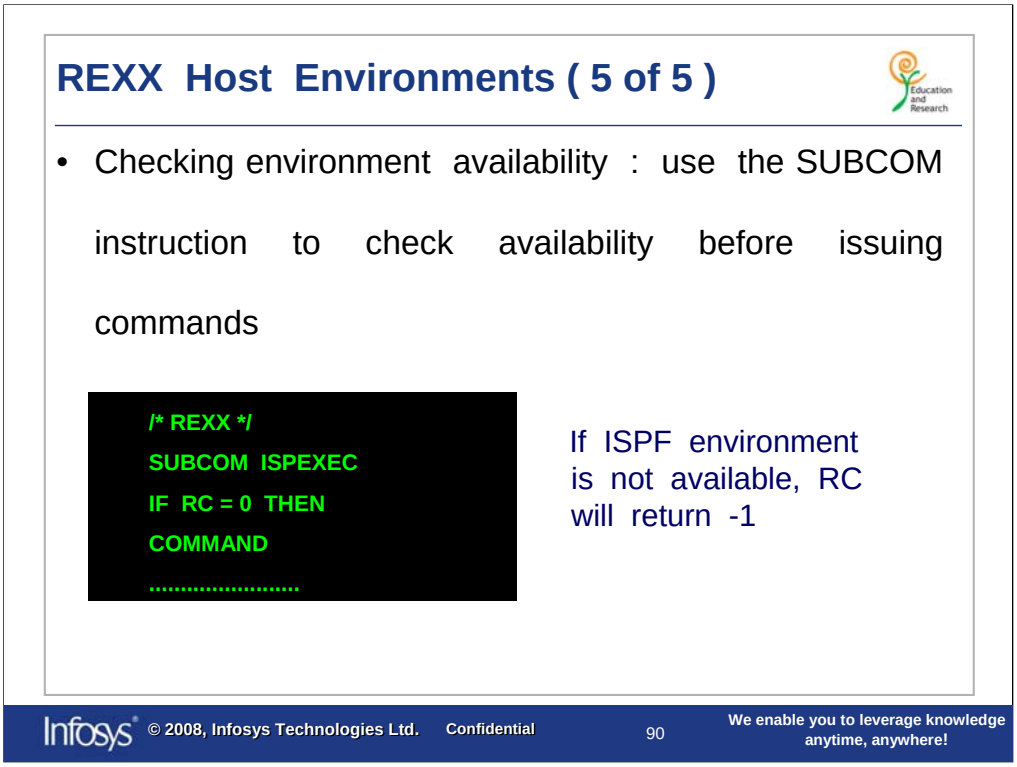
- To determine the active environment : use the ADDRESS built-in function

```
/* REXX */
X = ADDRESS()
SAY X
.....
```

Using the SAY instruction, the current environment can be displayed

REXX Host Environments (4 of 5) - Notes

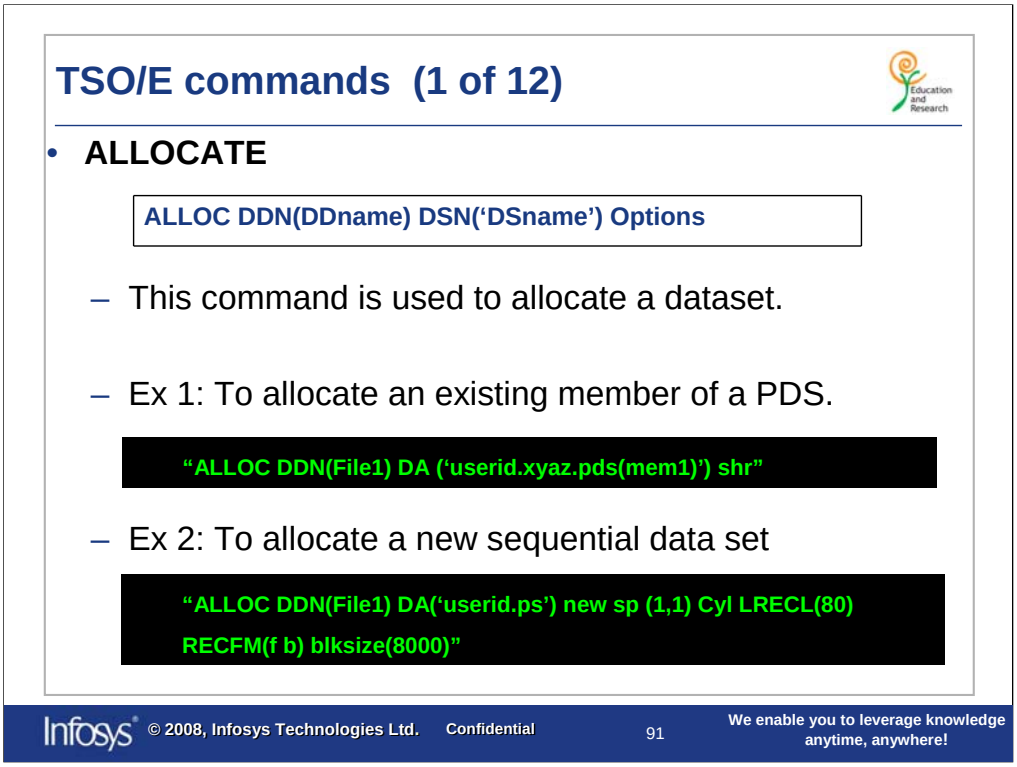
It maybe important that before attempting a specific task , an exec is “aware” of the currently active host environment . To determine the active environment, the ADDRESS built-in function can be used as shown above in the slide .



REXX Host Environments (5 of 5) - Notes

There may be times when a particular environment is not available. Since we would not want our REXX exec to end prematurely, we would naturally want to check for environment availability. The SUBCOM instruction (as shown in the slide above) can be used to check for environment availability before we issue commands . In the example given in the slide, we check for availability of the ISPF environment , before we issue any further commands. If ISPF is present , RC returns 0 and if ISPF is not present , RC returns a 1.

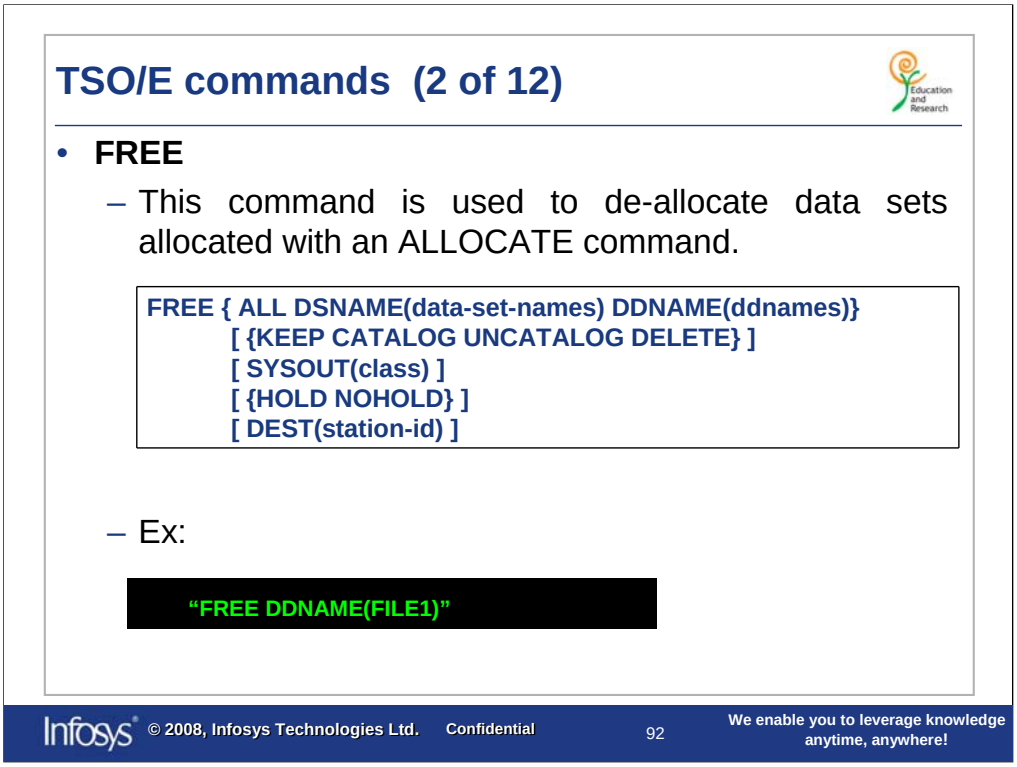
Now, let us have a look at few useful TSO/E commands.



TSO/E commands (1 of 12) – Notes

Syntax: ALLOC DDN(DDname) DSN('DSname') Options

ALLOCATE command is used to allocate a new or existing dataset. It is to associate the ddnames to dataset names. ddnames are the names used in programs to refer to files. Dataset names are names of actual datasets in disk or tape. The other function of this command is to specify the disposition of the dataset, file characteristics and space parameters if it is a new file. In example1 userid.xyaz.pds(mem1) is associated with the ddname File1 and the disposition is given as SHR. In example2 a new PS data set userid.ps is allocated and it is associated with the ddname File1



TSO/E commands (2 of 12) – Notes

FREE command is used to de allocate data sets allocated with an ALLOCATE command. The disposition specified on the ALLOCATE command will not be processed until de-allocation is done. All the data sets are implicitly de-allocated when the terminal session is ended by LOGOFF command.

Example1:

“FREE ALL”

Example2:

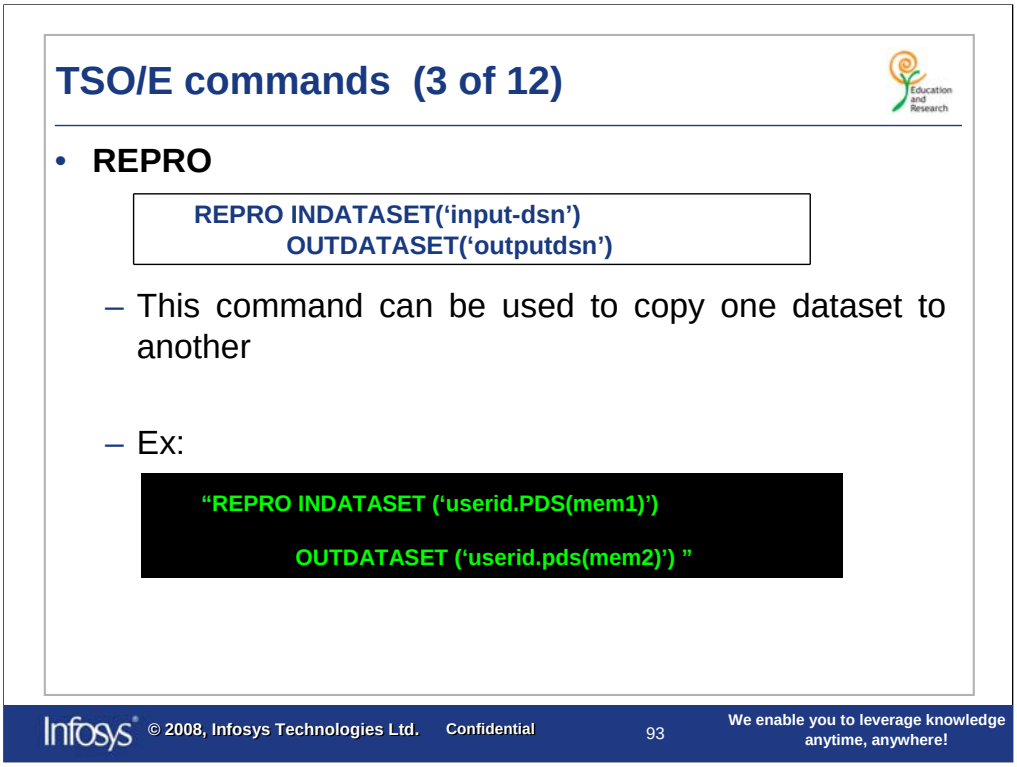
“FREE DSNAME(ERXXXXX.COBOL.PDS1 ERXXXXX.PS1)”

Example3:

“FREE DDNAME(FILE1)”

Example4:

“FREE DDNAME(FILE1) DELETE”

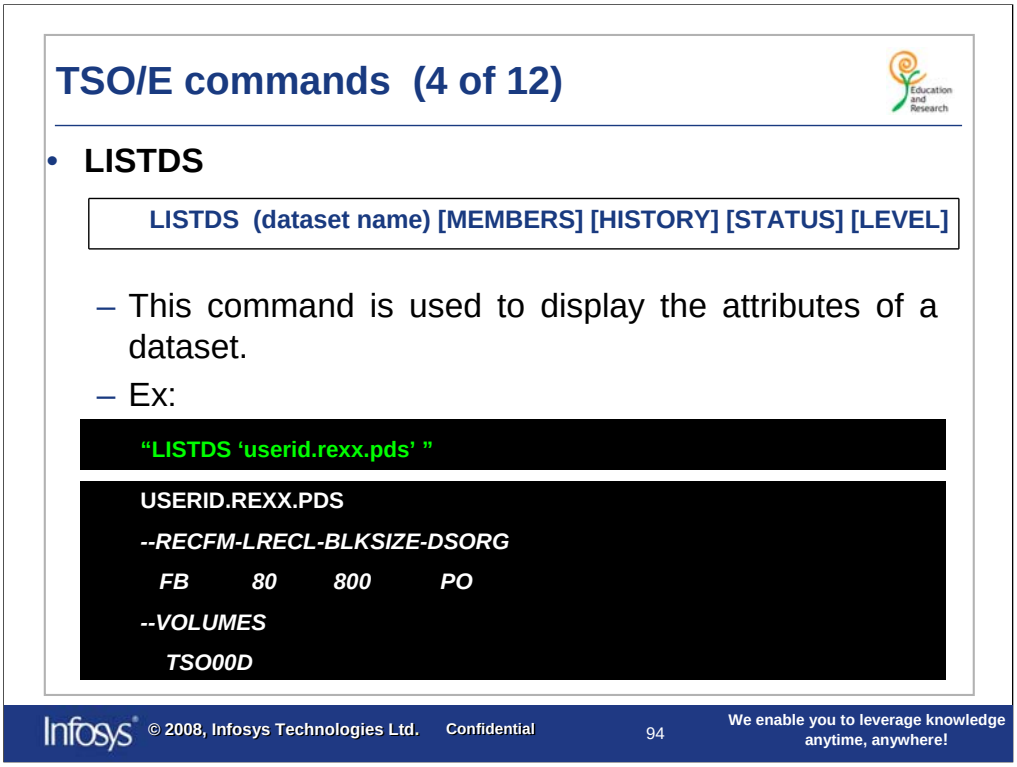


TSO/E commands (3 of 12) – Notes

REPRO INDATASET*('input-dsn')* OUTDATASET('*outputdsn*')

This command can be used to copy one dataset to another

In the above example, **mem1** of userid.PDS is copied into the same pds with a different member name **'mem2'**



TSO/E commands (4 of 12) – Notes

This command is used to display the attributes of a dataset.

Example1: **"*LISTDS 'userid.rexx.pds'* "**

The output will be:

USERID.REXX.PDS

--RECFM-LRECL-BLKSIZE-DSORG

FB 80 800 PO

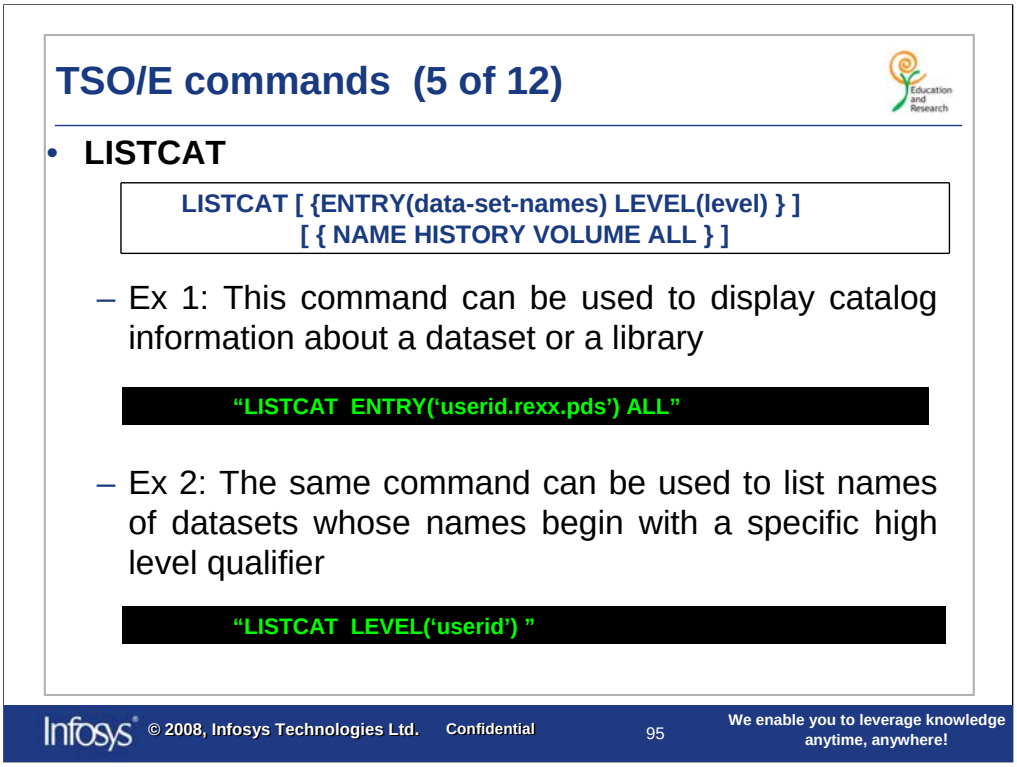
--VOLUMES

TSO00D

Using the OUTTRAP function, it is possible to trap this output and process it in the REXX exec. OUTTRAP

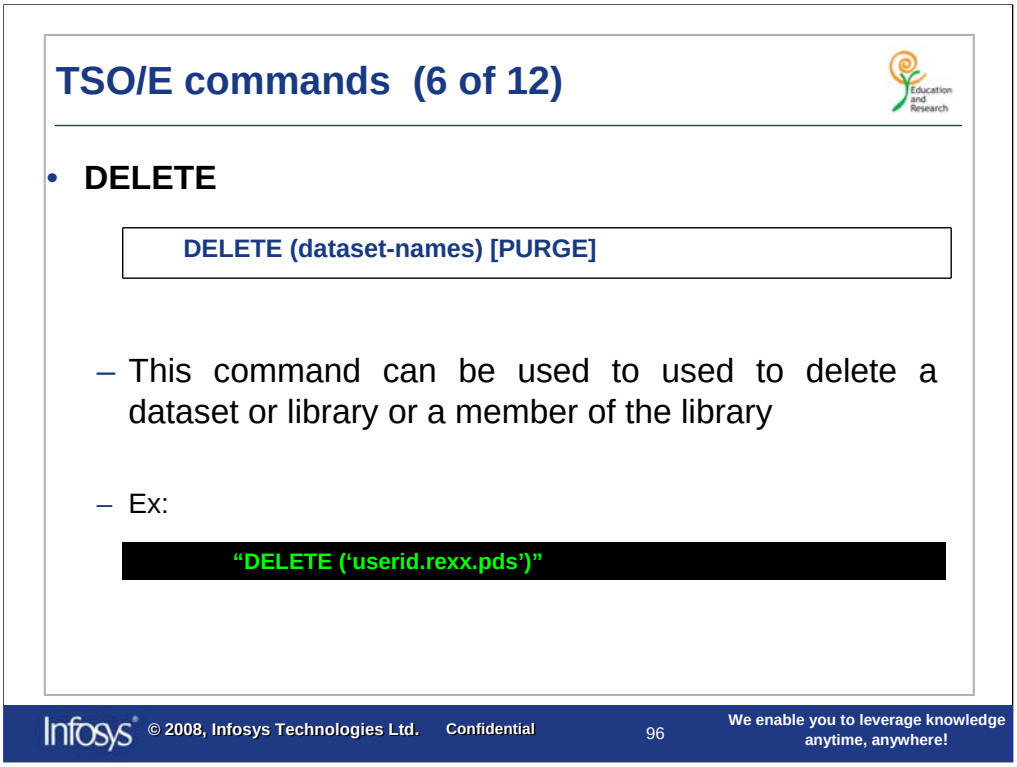
Example2: **"*LISTDS 'userid.rexx.pds'* MEMBERS"**

In the above example, MEMBERS option is specified. This will cause the command to display the members of the PDS along with the attributes of the PDS.



TSO/E commands (5 of 12) – Notes

LISTCAT is used to list entries in an operating system catalog. The number of entries to be displayed is controlled by the options LEVEL and ENTRIES or ENTRY. The amount of information displayed for each entry is controlled by NAME, HISTORY, VOLUME and ALL. In example1, all catalog information about the data set 'userid.rexx.pds' will be displayed. In example2, the names of datasets begins with high level qualifier 'userid' will be displayed.



TSO/E commands (6 of 12) – Notes

This command is used to delete a data set or a member of a PDS. You can specify a list of data set names separated by space inside the parenthesis. If you specify only one data set you may omit the parenthesis. PURGE option can be coded to scratch the data set even if its expiration date has not passed.

Example1:

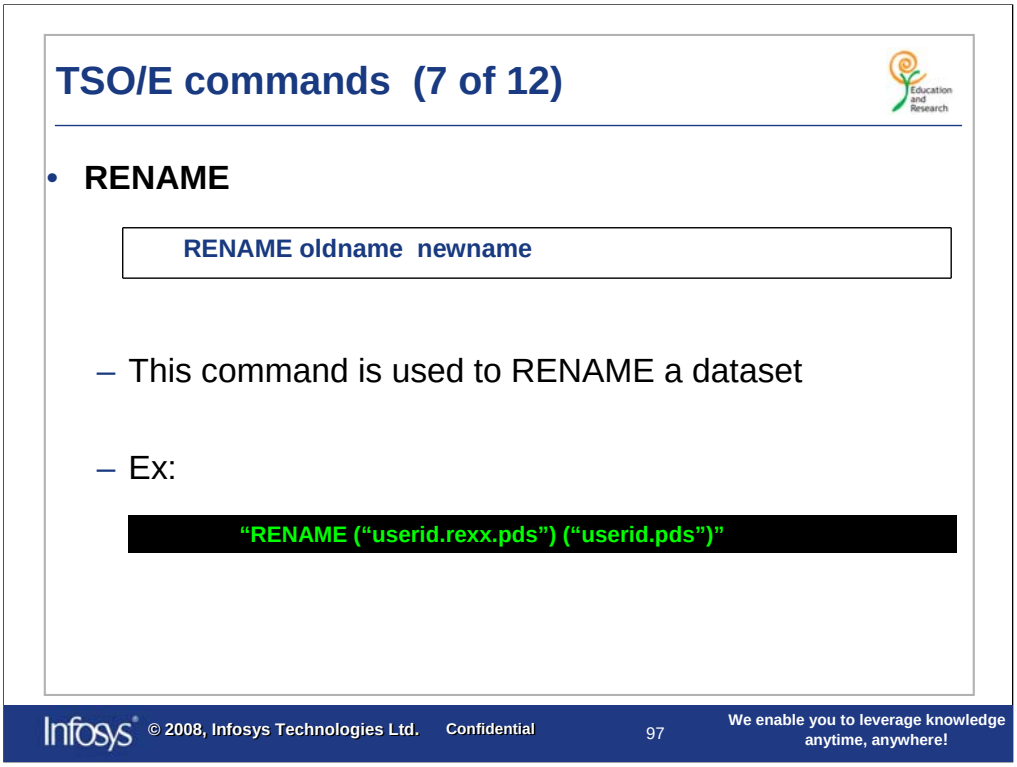
“DELETE ERXXXXX.COBOL.PDS”

Example2:

“DELETE (ERXXXXX.COBOL.PDS1 ERXXXXX.PS1 ERXXXXX.PS2)”

Example3:

“DELETE ERXXXX.COBOL.*”



TSO/E commands (7 of 12) – Notes

This command is used to rename a dataset. You need to specify the old name first and the new name second.

TSO/E commands (8 of 12)

- **XMIT**

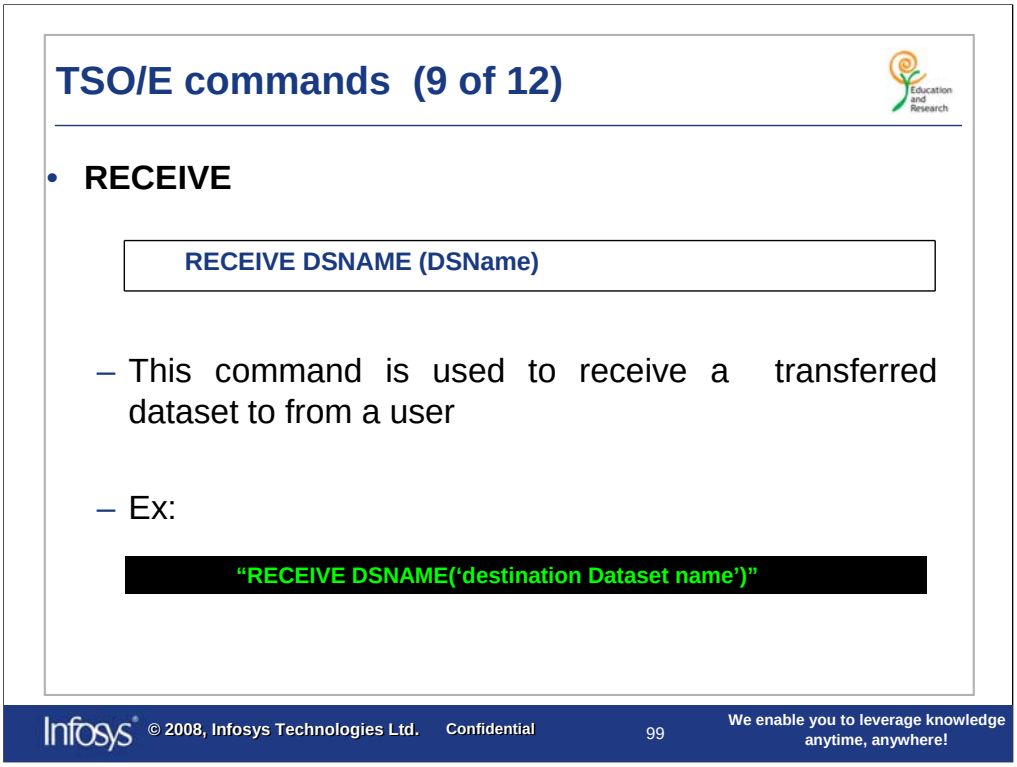
```
XMIT node.touserid dsn(dsname)
```

  - This command can be used to transfer a dataset to another user
  - Ex:

```
"XMIT ZOSYSTEM.TOUSERID DSN('USERID.PDS')"
```

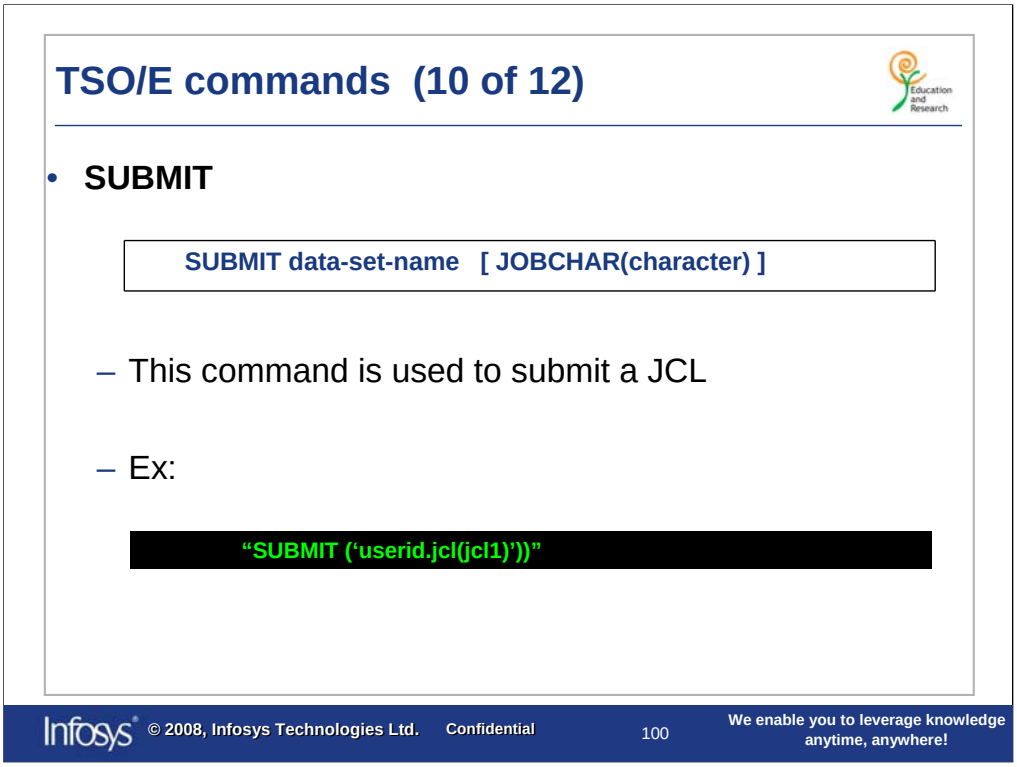

TSO/E commands (8 of 12) – Notes

This command can be used to transfer a dataset to another user. Here, ZOSYSTEM node name of the destination user.



TSO/E commands (9 of 12) – Notes

This command is used to receive a transferred dataset to from a user. The dataset transferred from a user using XMIT is received here.



TSO/E commands (10 of 12) – Notes

SUBMIT command is used to submit a JCL. In the above example jcl1 which is the member of 'userid.jcl' is submitted. In JOBCHAR option a single character or digit is coded so that it is appended to the user-id to form a job name. JOBCHAR is used if job stream doesn't contain a JOB statement.

TSO/E commands (11 of 12)

- **CALL**

```
CALL dataset-name ['parameter-string']
```

  - This command is used to execute a compiled program.
  - Ex:

```
"CALL ('LOADLIB(PGM1)')"
```


TSO/E commands (11 of 12) – Notes

The name of the load module to be executed is coded here. The parameter-string can be coded which contains the parameters passed to the program.

TSO/E commands (12 of 12)

- **CANCEL**

```
CANCEL { job-name job-name(job-id) }
       [ { PURGE NOPURGE }]
```

  - This command is used to cancel a batch job.
  - If purge option is specified, the job is cancelled as well as purged
  - Ex:

```
"CANCEL e87689a(job000548)  PURGE"
```


TSO/E commands (12 of 12) – Notes

This command is used to cancel a batch job. PURGE option is coded to remove the job's output from the SYSOUT queue. NOPURGE is the default. It means the job should be cancelled it is executing and the output should not be removed from the job queue.

Condition Traps (1 of 7)

- Condition is a specified state or an event which can be trapped by CALL ON or SIGNAL ON
- Using the condition traps, flow of execution within a REXX program can be modified
- ON & OFF sub keywords of SIGNAL & CALL instructions can be used to turn the trap on or off
- The default setting for all condition traps is OFF

Condition Traps (2 of 7)

- Syntax

```
CALL    OFF condition    NAME trapname
        ON condition

SIGNAL  OFF condition    NAME trapname
        On condition
```

- When a trap is enabled for a *condition* & if that *condition* occurs, control will be passed to the routine named
  - ***trapname*** if *trapname* is specified.
  - ***Condition*** if *trapname* is not specified.

Condition Traps (2 of 7) – Notes

If CALL is used, an internal label, a built-in function, or an external routine can be specified as the *trapname*.

If SIGNAL is used, only an internal label can be specified as the *trapname*.

Condition Traps (3 of 7)

- The following conditions can be trapped
- **ERROR**
  - A command which indicates an error upon return raises this condition
  - A command which indicates failure upon return can also raise this condition if neither SIGNAL ON FAILURE nor CALL ON FAILURE is specified

Condition Traps (3 of 7) - Notes

A command which indicates an error upon return raises this condition. A command which indicates failure can also raise this condition if neither SIGNAL ON FAILURE nor CALL ON FAILURE is specified.

All positive return codes will be trapped by SIGNAL ON FAILURE. Negative return codes are trapped only when neither SIGNAL ON FAILURE nor CALL ON FAILURE is specified

Condition Traps (4 of 7)

- **FAILURE**
  - A command which indicates failure upon return raises this condition
  - SIGNAL ON FAILURE & CALL ON FAILURE can trap all negative return codes from commands
- **HALT**
  - Any external attempt to interrupt & end the execution of the program raises this condition
  - Ex: TSO/E REXX command HI or EXECUTIL HI commands raises a HALT condition.

Condition Traps (4 of 7) – Notes

FAILURE

A command which indicates failure upon return raises this condition

SIGNAL ON FAILURE & CALL ON FAILURE can trap all negative return codes from commands

HALT

Any external attempt to interrupt & end the execution of the program raises this condition

Ex: TSO/E REXX command HI or EXECUTIL HI commands raises a HALT condition

Condition Traps (5 of 7)

- **NOVALUE**
  - This condition is raised if a variable which is uninitialized is used:
    - As a term in an expression
    - As the variable mentioned after the PARSE instruction's VAR keyword
    - As a variable in a **parse** template, a **drop** or **procedure** instruction
  - This condition can be specified only for SIGNAL ON

Condition Traps (5 of 7) – Notes

NOVALUE

```
/* The following will not raise NOVALUE. */
signal on novalue
str.=0
say str.var1
say 'NOVALUE is not raised.'
exit
novalue:
say 'NOVALUE is raised.'
```

SIGNAL ON NOVALUE will trap any uninitialized variable except tails in compound variables

Condition Traps (6 of 7)

- **SYNTAX**
  - A language processing error detected during a program's execution raises this condition
  - Syntax errors, run time errors such as attempting arithmetic operation on non numeric data fall into this category
  - This condition can be specified only for SIGNAL ON

Condition Traps (6 of 7) – Notes

SYNTAX

A language processing error detected during a program's execution raises this condition. Syntax errors, run time errors such as attempting arithmetic operation on non numeric data fall into this category.

Condition Traps (7 of 7)

- If a specific condition occurs & the trap is in the OFF state for that condition, then default action is taken.
- This action depends on the type of condition:
  - Execution of the program is terminated & a message describing the nature of the event that occurred is given out for HALT & SYNTAX conditions
  - Any other condition will be ignored

Debugging using RC and SIGL (1 of 2)

- RC and SIGL are special system variables and cannot be used for any other purposes
- RC is set every time a command is issued.
- RC is set to 0 whenever a command is successfully executed

Debugging using RC and SIGL (1 of 2) - Notes

We now take a look at special variables RC and SIGL which help diagnose problems within execs. RC and SIGL are special system variables and should not be used for any other purposes . RC is set every time a command is issued . If a command ends without error , RC is set to 0 . Otherwise if a command ends in error, it is set to whatever return code is assigned to that error .

Debugging using RC and SIGL (2 of 2)

- When control is transferred to another routine or another part of the exec , the SIGL variable is set to the line number from which the transfer occurred.

```
10  /*REXX*/
11  CALL  RTN
12  :
13  RTN:
14  SAY SIGL
```

The value of SIGL after the CALL instruction would be 11

Debugging using RC and SIGL (2 of 2) - Notes

Whenever control is transferred to another routine or another part of the exec , the SIGL variable is set to the line number from which the transfer occurred. In the code snippet shown above in the slide , the CALL instruction in line 11 causes transfer of control to line 13 to the routine RTN . Now in that routine , the variable SIGL is set to value 11 (i.e. The line number from which the transfer of control occurred) .

Today, we have learnt the topics listed in the above slide.

References

- Gabriel Gargiul, REXX in the TSO environment
- M.F. Cowlishaw, The REXX language – A practical approach to programming , Prentice Hall PTR, 1990
- Robert P. O'Hara, David Roos Gomberg, Modern Programming Using Rexx, Prentice Hall Inc, 1988
- OS/390 TSO/E REXX Reference manual found online at http://publibz.boulder.ibm.com/epubs/pdf/ikj3a330.pdf
- OS/390 TSO/E REXX User's Guide found online at http://publibz.boulder.ibm.com/epubs/pdf/ikj3c302.pdf

References - Notes

The references are listed on the slide.

References

- Kshop links
  - http://kshop/kshop/showdoc.asp?DocNo=72424
  - http://kshop/kshop/showdoc.asp?DocNo=8601
  - http://kshop/kshop/showdoc.asp?DocNo=94325
  - http://kshop/kshop/showdoc.asp?DocNo=160272
- Also see REXX CBT at http://enrcbt/cbtcontents

References - Notes

The references are listed on the slide.