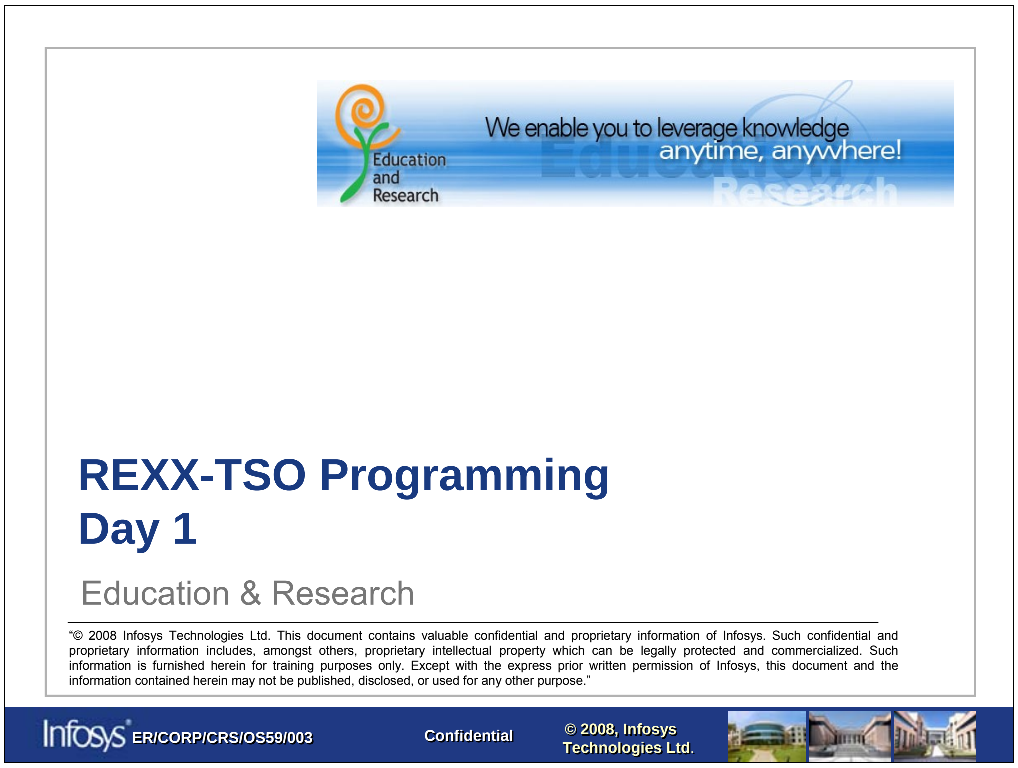


We enable you to leverage knowledge anytime, anywhere!

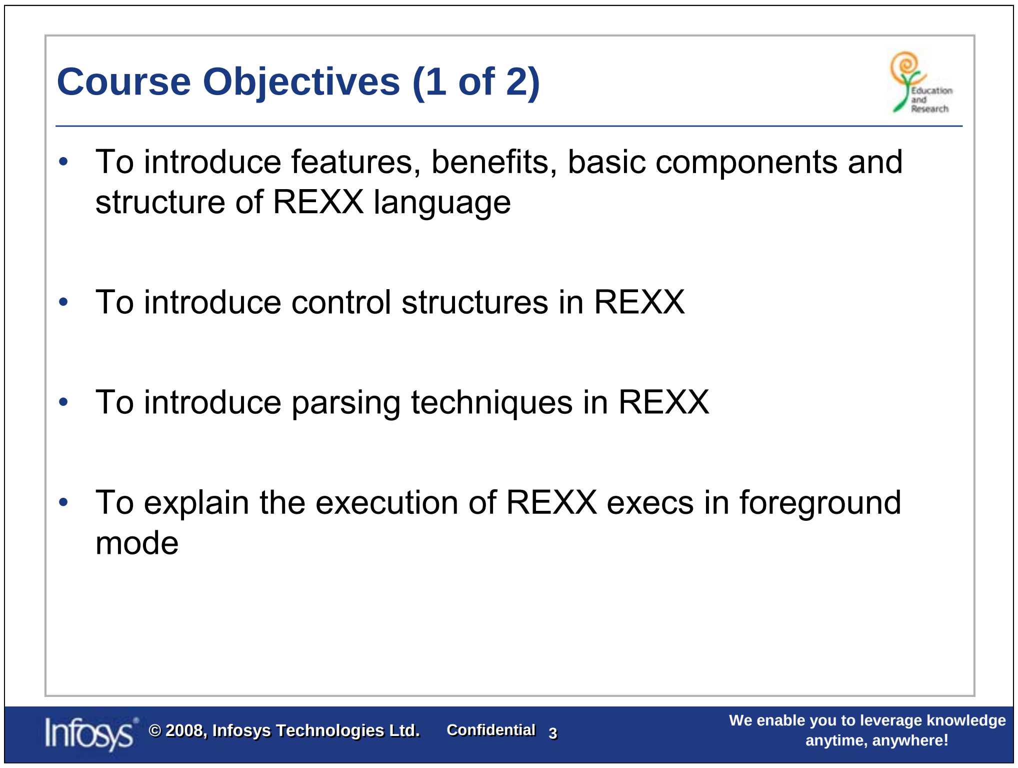
Education and Research

REXX-TSO Programming Day 1

Education & Research

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

Confidential Information

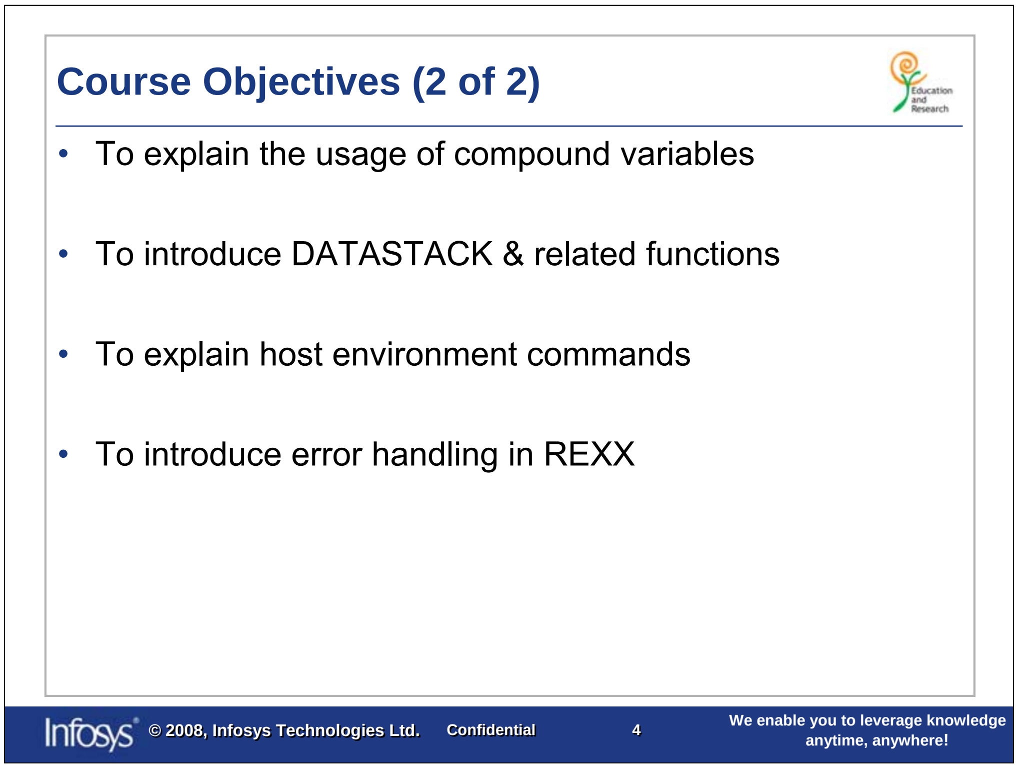


Course Objectives (1 of 2)

- To introduce features, benefits, basic components and structure of REXX language
- To introduce control structures in REXX
- To introduce parsing techniques in REXX
- To explain the execution of REXX execs in foreground mode

Course Objectives (1 of 2) - Notes

The objective of the course is to discuss the above listed broad topics . These topics will be covered across two days.

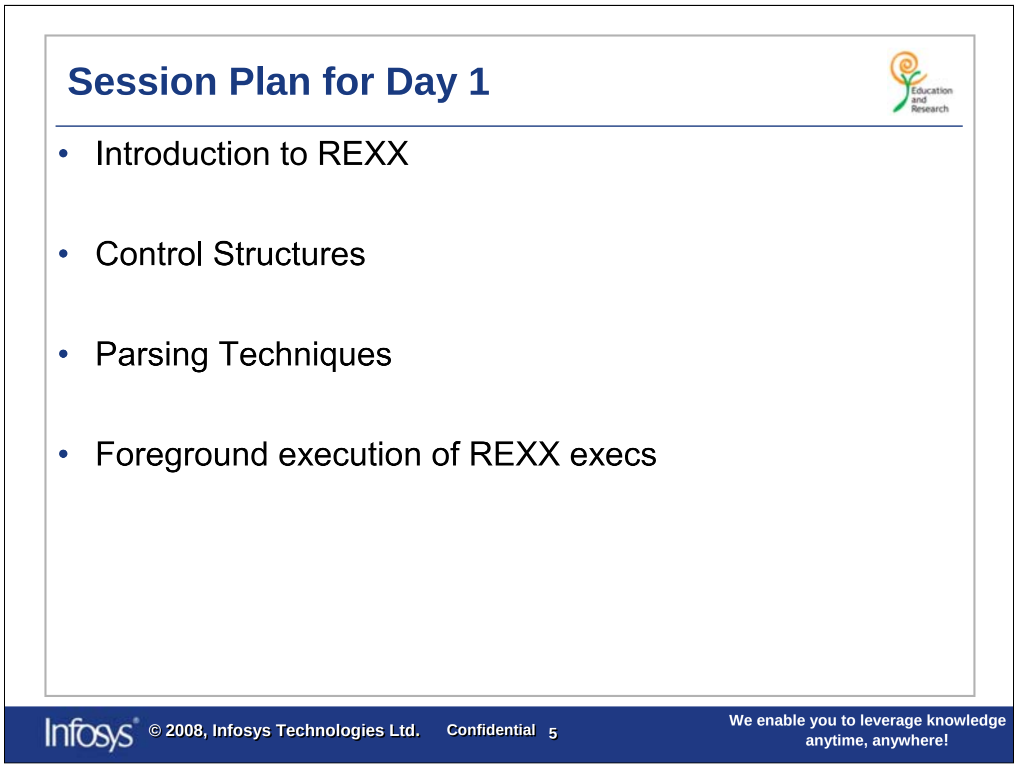


Course Objectives (2 of 2)

- To explain the usage of compound variables
- To introduce DATASTACK & related functions
- To explain host environment commands
- To introduce error handling in REXX

Course Objectives (2 of 2) - Notes

The objective of the course is to discuss the above listed broad topics . These topics will be covered across two days.

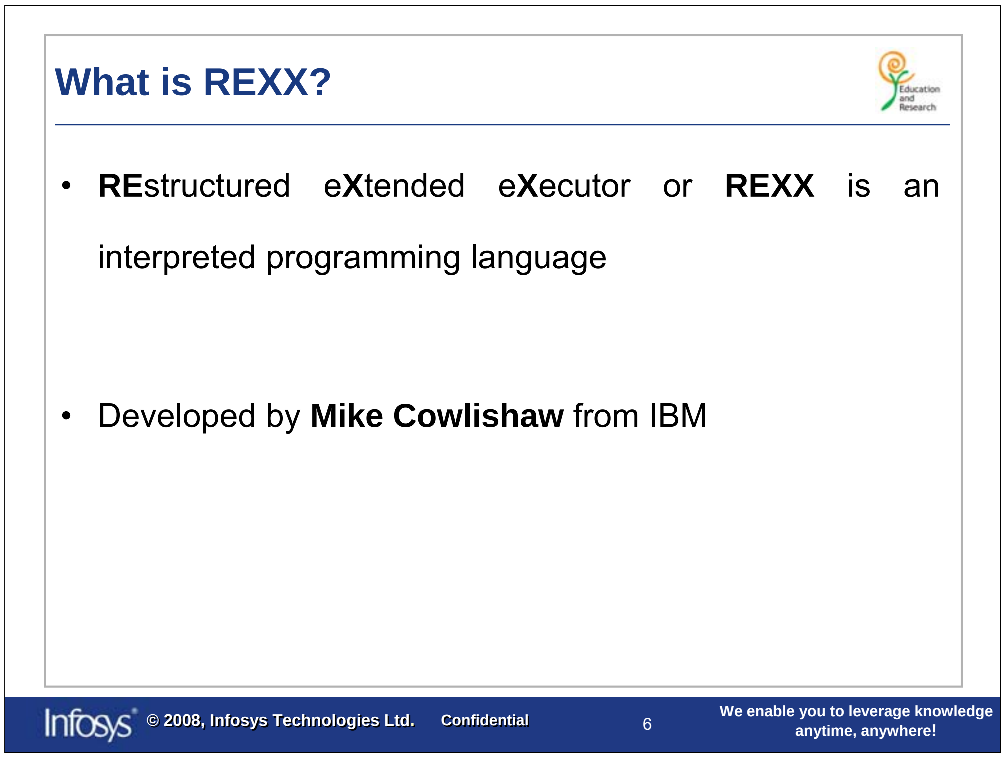


Session Plan for Day 1

- Introduction to REXX
- Control Structures
- Parsing Techniques
- Foreground execution of REXX execs

Session Plan for Day 1 - Notes

Upon completion of Day 1 of this course, you will gain an understanding of the topics that are listed here.



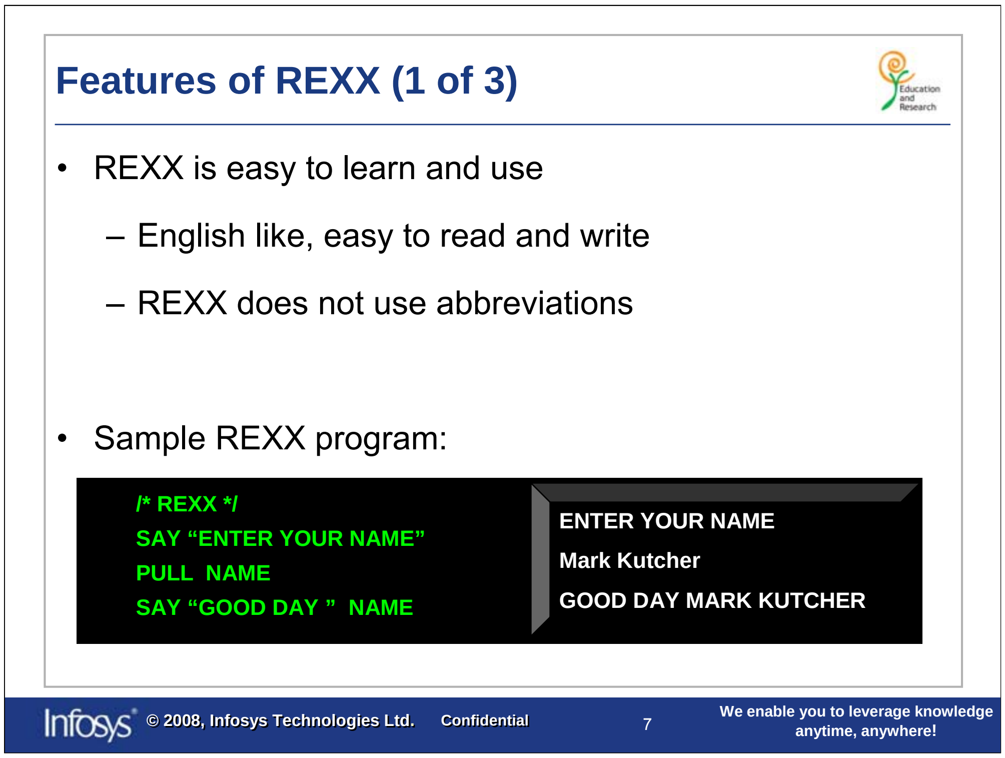
What is REXX?

- **RE**structured e**X**tended e**X**ecutor or **REXX** is an interpreted programming language
- Developed by **Mike Cowlishaw** from IBM

What is REXX - Notes

Restructured Extended Executor or REXX is a programming language developed by Mike Cowlishaw from IBM. REXX was basically developed so that programming could be made easier. This language, which was developed mainly for the system administrator, is easy to learn & hence it makes it easy for an experienced administrator as well as a novice programmer. Since REXX is an interpreted language, there is no need for a compiler.

It has many features such as free form coding, character data parsing capabilities, ability to include host environment commands etc. REXX is an easy language to use, simple enough for beginners yet also powerful and versatile enough as well for experienced application developers. In the subsequent slides, we discuss some of the important features of the language .



Features of REXX (1 of 3)

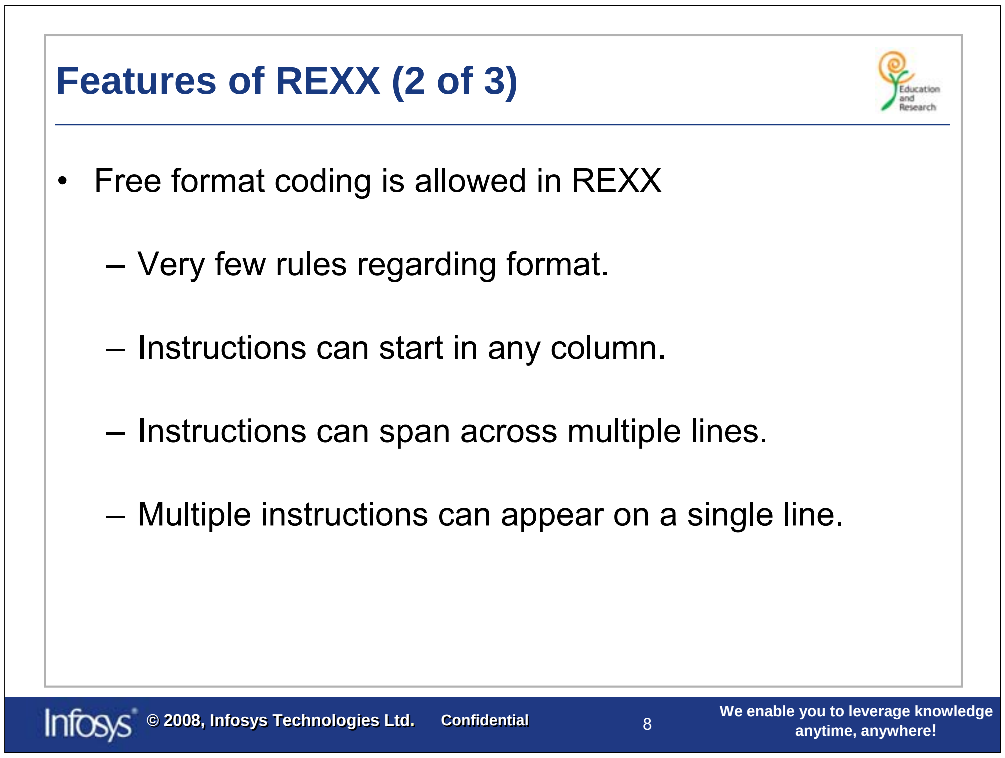
- REXX is easy to learn and use
  - English like, easy to read and write
  - REXX does not use abbreviations
- Sample REXX program:

```
/* REXX */
SAY “ENTER YOUR NAME”
PULL  NAME
SAY “GOOD DAY ”  NAME
```

```
ENTER YOUR NAME
Mark Kutcher
GOOD DAY MARK KUTCHER
```

Features of REXX (1 of 3) - Notes

Many of the instructions in REXX are simple English-like words or phrases that have similar meanings outside of computing. This enhances understandability of REXX programs. In the sample program, it is very evident that the instructions SAY and PULL send and receive data respectively from the terminal. Also REXX does not use abbreviations. This enhances the understandability of the language.

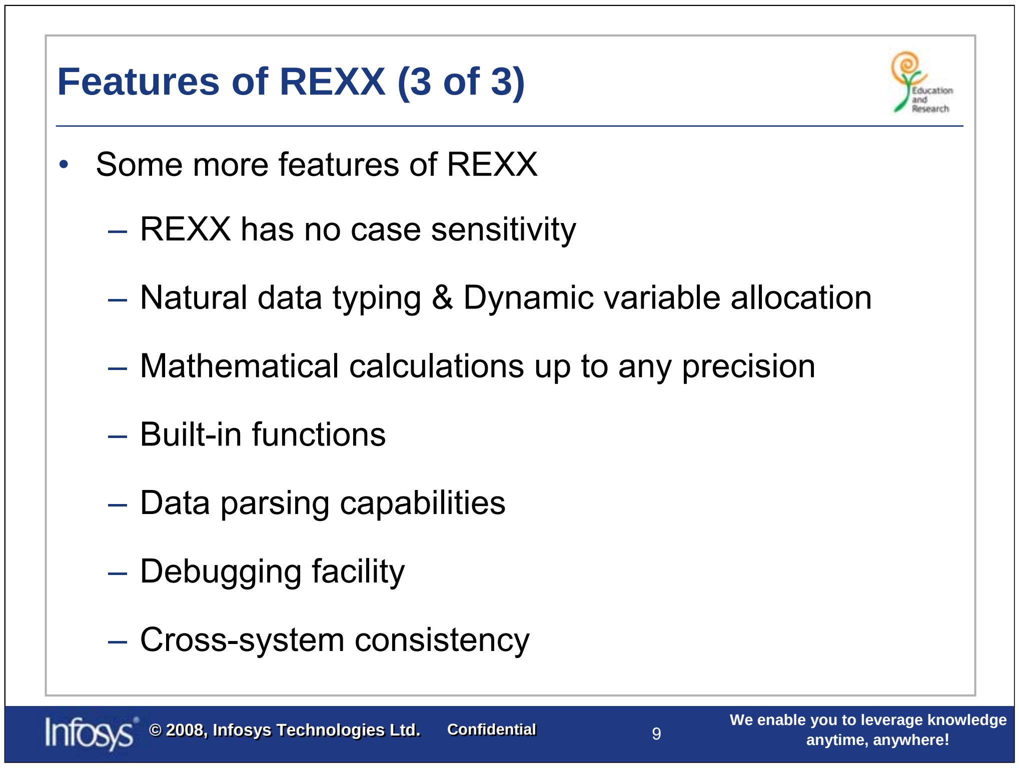


Features of REXX (2 of 3)

- Free format coding is allowed in REXX
  - Very few rules regarding format.
  - Instructions can start in any column.
  - Instructions can span across multiple lines.
  - Multiple instructions can appear on a single line.

Features of REXX (2 of 3) - Notes

There are only a few rules regarding format. Instructions can start in any column, you can skip spaces and lines, instructions can span lines and multiple instructions can appear on a single line. The essentially free format of the language allows users to layout their code in the style they feel is the most readable.



Features of REXX (3 of 3)

- Some more features of REXX
  - REXX has no case sensitivity
  - Natural data typing & Dynamic variable allocation
  - Mathematical calculations up to any precision
  - Built-in functions
  - Data parsing capabilities
  - Debugging facility
  - Cross-system consistency

Infosys © 2008, Infosys Technologies Ltd. Confidential 9 We enable you to leverage knowledge anytime, anywhere!

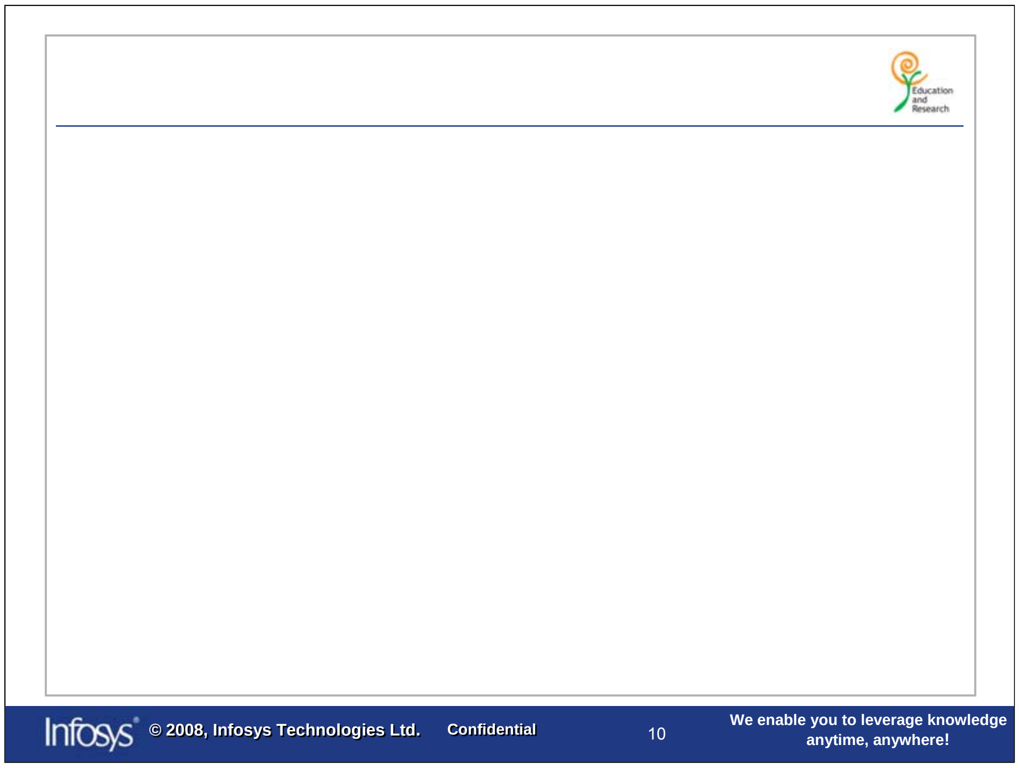
Features of REXX (3 of 3) - Notes

REXX is not case-sensitive. No distinction is made between symbols(instructions, variables etc.) written in either upper or lower case, or any combination of both . For example , VAR1, var1 , VaR1, vAR1 etc all refer to the same variable.

The “natural” data typing of REXX is one of it’s very important features. All data is treated as character strings. Numbers, including both integers and reals, are special cases of strings.

Numbers are treated as strings of characters which are made up of one or more digits (0-9). These strings may also contain a sign (plus or minus). However, this is purely optional. Also, optionally they may contain a single period (referred to as decimal point). If there is a need to represent a number in terms of power of 10, then we may do it by using the conventional exponential notation (E – either upper case or lower case). Here, again the number may have an optional sign. There can also be leading signs to the number. For example: +25E-2 (this is equal to 0.25). There can also be trailing blanks. However, blanks are not allowed in between the digits of the number or in the exponential part.

A consequence of this approach is that data declarations are never required. Data declarations in other languages are used because the computer maintains different internal data representations for different purposes and must be told which representation to use for a given data item. REXX relieves the user from all concern with internal representations and as a consequence, we do not need to predefine variables in a REXX program.



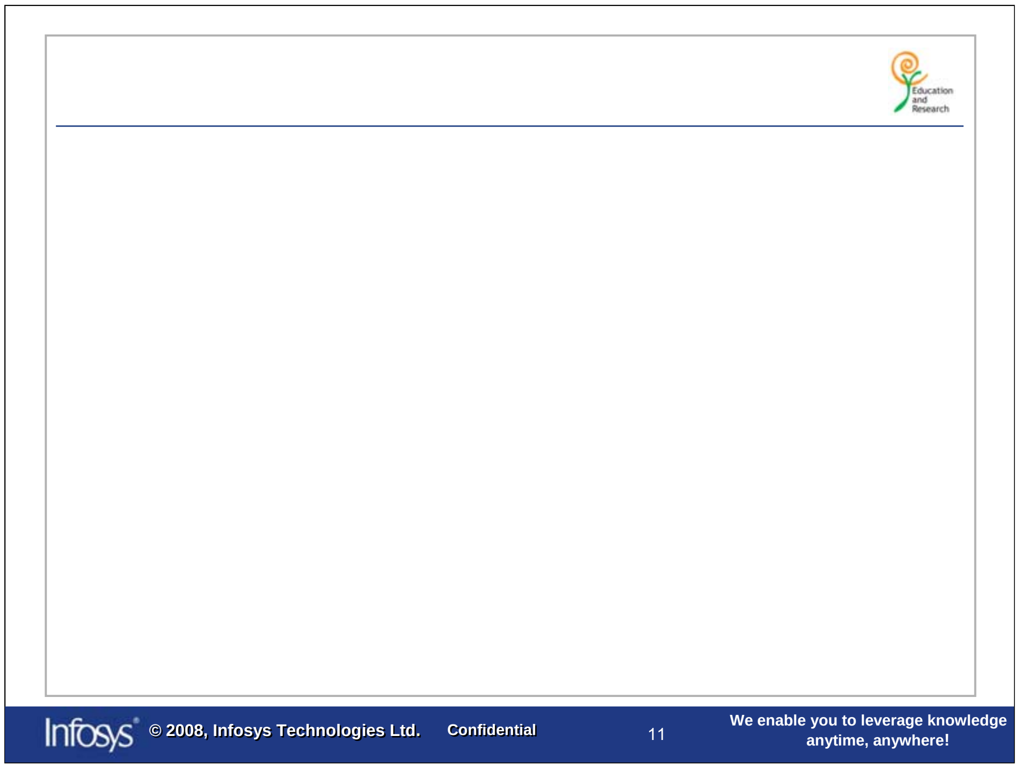
Features of REXX (3 of 3) - Notes (Contd.)

Another consequence of treating all data as character strings is that there is no inconvenient limits on the precision of numeric calculations . The default precision is 9.

A large number of character-string manipulation features including operations such as substring, replacement, searching, insertion etc are provided in REXX through a number of built-in functions. A number of formatting and arithmetic operations are also available to the REXX programmer. Example: REVERSE, MAX, etc. Some other standard built-in functions are DATE() which returns the current date in a variety of formats, SYMBOL() which indicates whether a given variable has been initialized or not etc. The extensive library of built-in functions also contain functions for file I/O.

The parsing capabilities of the language allow you to take strings from a number of possible sources and break them apart as required into constituent parts using a rather natural notation. The part of the instruction that tells how to parse the string is called parse template.

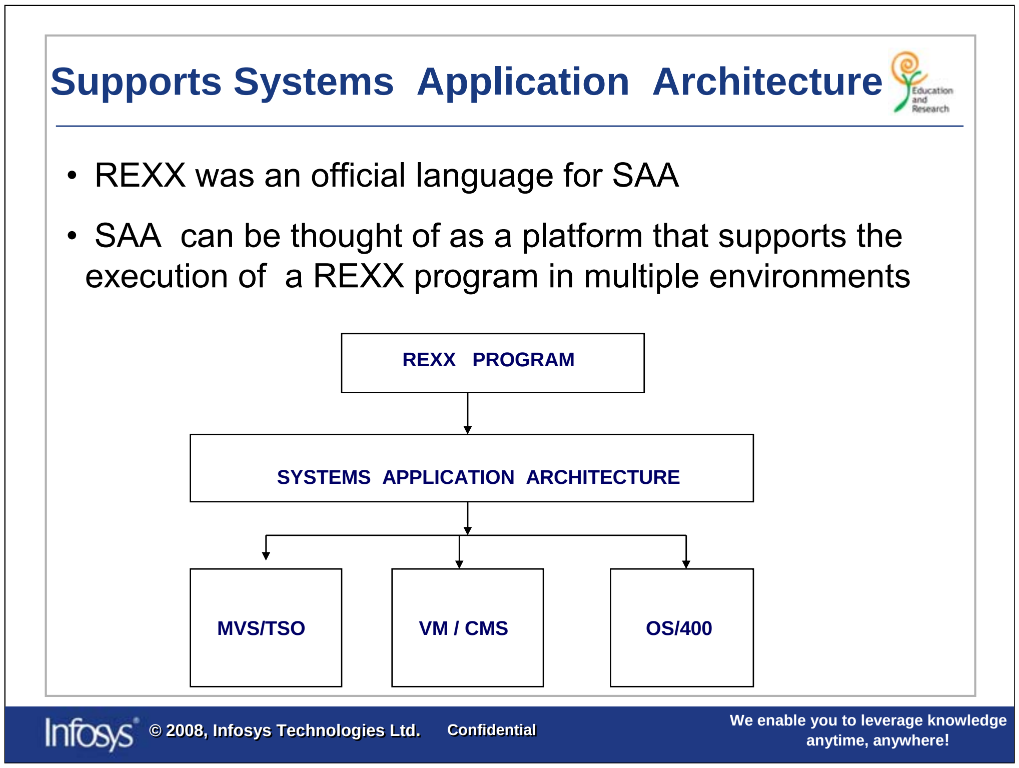
REXX has flexible debugging capabilities as part of the language definition. Through the TRACE instruction, we can trace program execution at varying levels of detail. We can trace each statement executed, the evaluated results of an expression, or even the intermediate results during expression evaluation. The trace can be either non-stop or interactive . During interactive tracing, we can execute any REXX statement, call subroutines, display and change variables and so on.



Education and Research

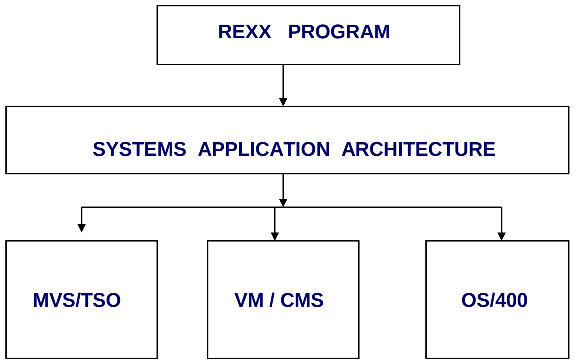
Features of REXX (3 of 3) - Notes (Contd.)

The best aspect of REXX is the cross-system consistency that it provides. REXX is spread out on mainframes as well as micros. It's implementations are available for MVS, VM/CMS,OS/2 and OS/400. Third party vendors have developed versions of REXX for Unix and for Tandem computers. A program that is written in REXX will run in a variety of environments with little or no re-coding required. This means that you can write a REXX program to run under CMS (Conversational Monitor System) and port it immediately to run in a TSO environment. There may be small changes required, but they are usually minimal.



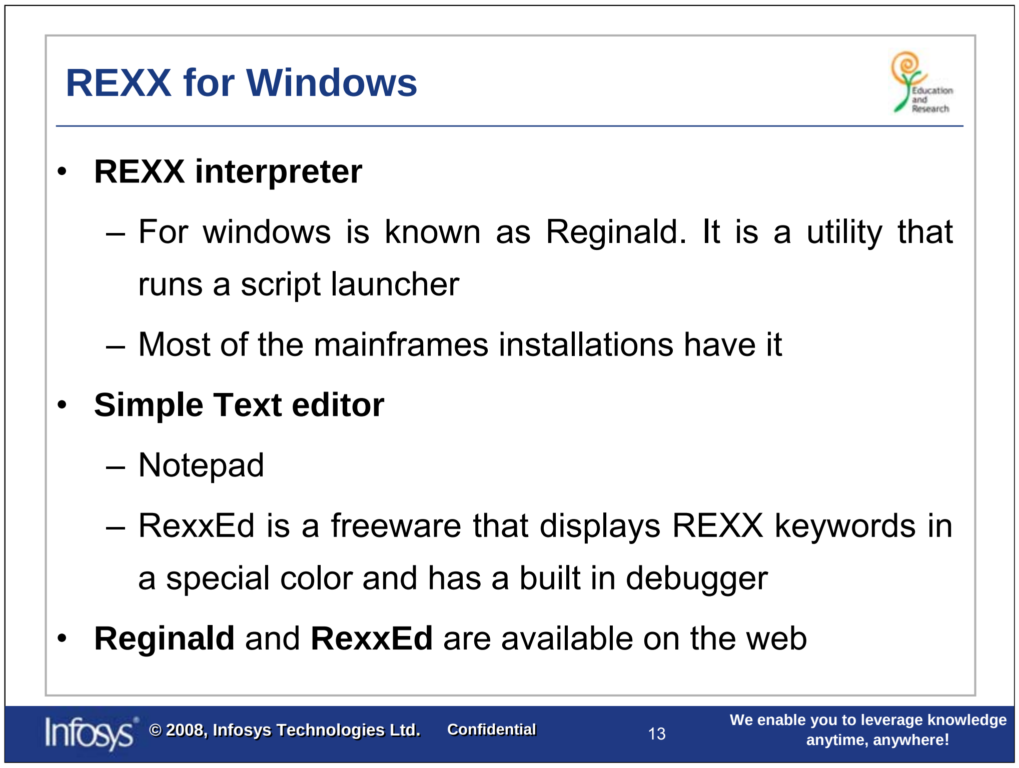
Supports Systems Application Architecture

- REXX was an official language for SAA
- SAA can be thought of as a platform that supports the execution of a REXX program in multiple environments



Systems Application Architecture - Notes

The ability to execute a REXX program in different host (or operating) environments is a result of System Application Architecture (SAA). SAA is a set of protocols and interfaces that provide a platform for developing applications with cross-system consistency. SAA defines common programming interface, communications support , user access and applications .



REXX for Windows

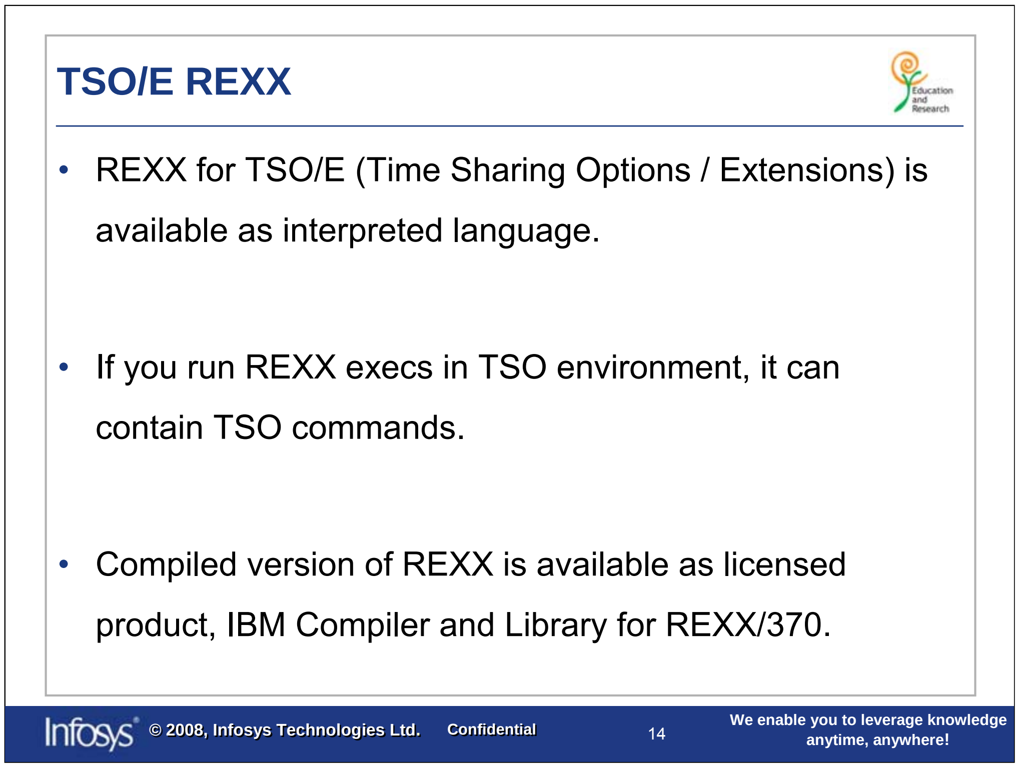
- **REXX interpreter**
  - For windows is known as Reginald. It is a utility that runs a script launcher
  - Most of the mainframes installations have it
- **Simple Text editor**
  - Notepad
  - RexxEd is a freeware that displays REXX keywords in a special color and has a built in debugger
- **Reginald** and **RexxEd** are available on the web

What do I need to get started - Notes

Usually, development tools are expensive. But the great news is that everything you need is free, and readily available. You need a REXX interpreter. This actually "runs" your REXX script just like Visual Basic runs a BASIC program. A free REXX interpreter for Windows 95/98/ME/NT/2000/XP called "Reginald" is available on web. Reginald installs a utility that runs your REXX script called the "REXX Script Launcher". The Script Launcher transparently runs your script. You do not need to manually run the Script Launcher. You can simply double-click on the icon for your REXX script, and the Script Launcher will automatically run your script.

You need a simple text editor, such as Notepad, to write your REXX scripts. A better choice is the freeware text editor, RexxEd, available on the web. It displays REXX keywords in a special color to make it easier on your eyes to read a REXX script. Plus, it has a built-in debugger to work out problems with your script and online help facility. It even runs REXX scripts directly from the editor (so you don't have to first save them to disk).

The installation program for Reginald will install and completely setup REXX on your computer. You must also install RexxEd separately, if you wish to use that text editor. If you are working in mainframes, then REXX interpreter should be available there to work on as most of the mainframe installations have this.

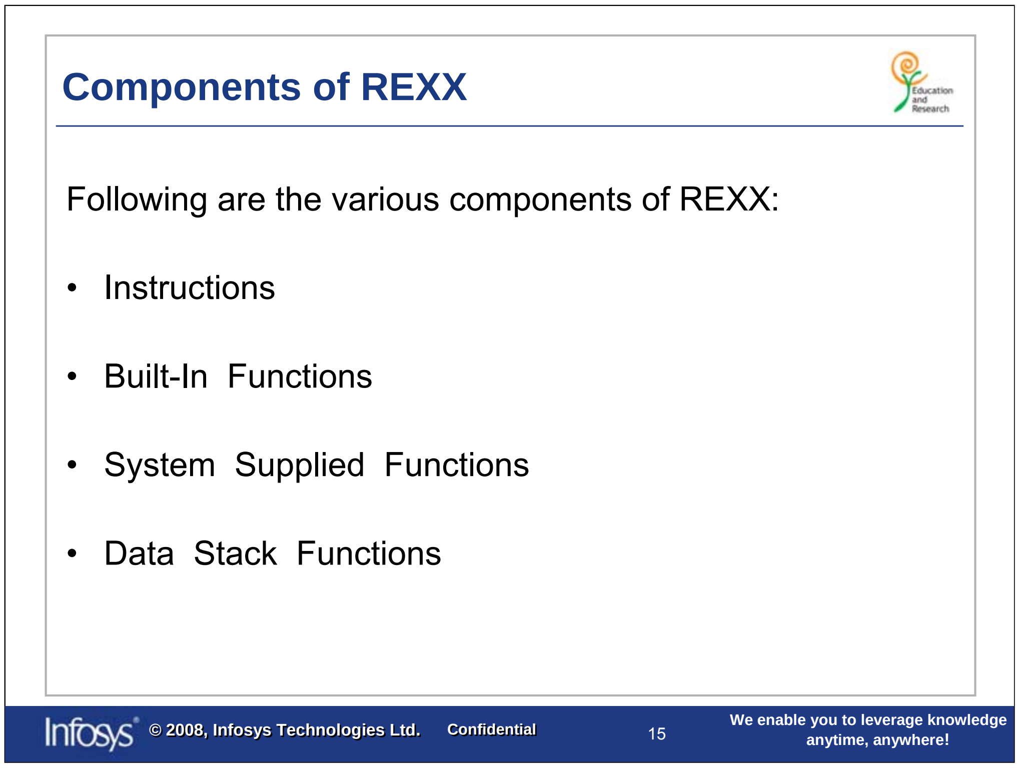


TSO/E REXX

- REXX for TSO/E (Time Sharing Options / Extensions) is available as interpreted language.
- If you run REXX execs in TSO environment, it can contain TSO commands.
- Compiled version of REXX is available as licensed product, IBM Compiler and Library for REXX/370.

TSO/E REXX – Notes

The implementation of REXX language in TSO/E is available as interpreted language. The language processor processes the statements in REXX execs directly. The compiled version of REXX is available as licensed product, IBM Compiler and Library for REXX/370. This course is developed based in TSO/E REXX interpreted version REXX370 3.48.



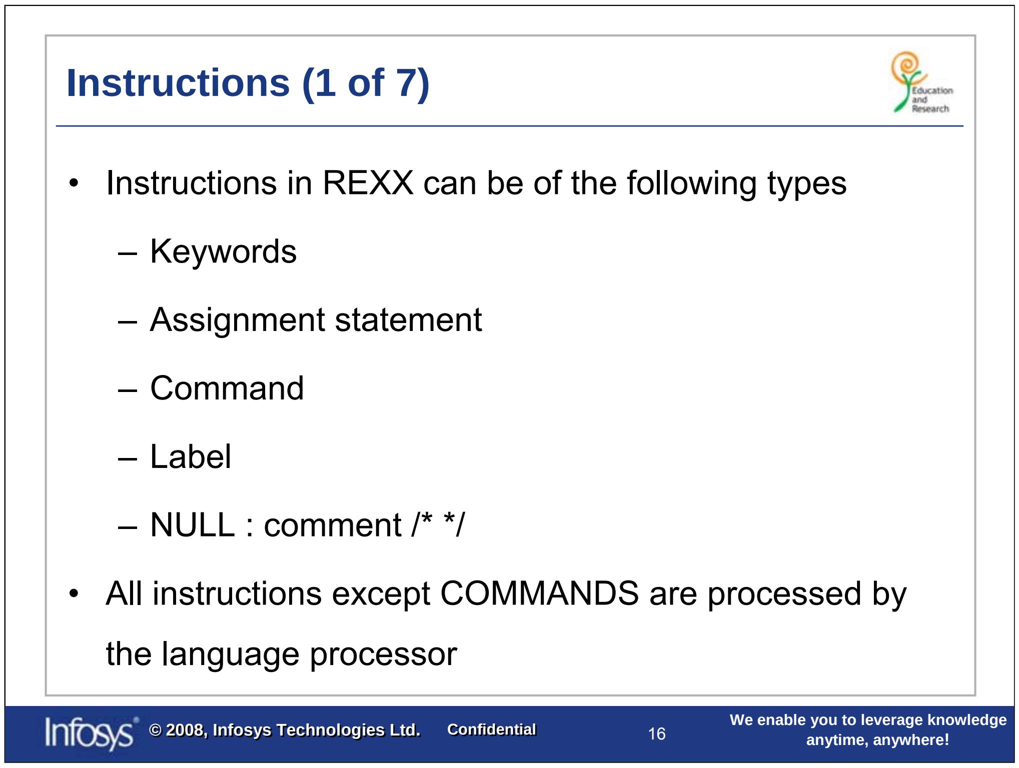
Components of REXX

Following are the various components of REXX:

- Instructions
- Built-In Functions
- System Supplied Functions
- Data Stack Functions

Components of REXX - Notes

The presence of various components in REXX make it a powerful programming language. Broadly speaking there are four components namely instructions, REXX built-in functions, system supplied functions and data stack functions.



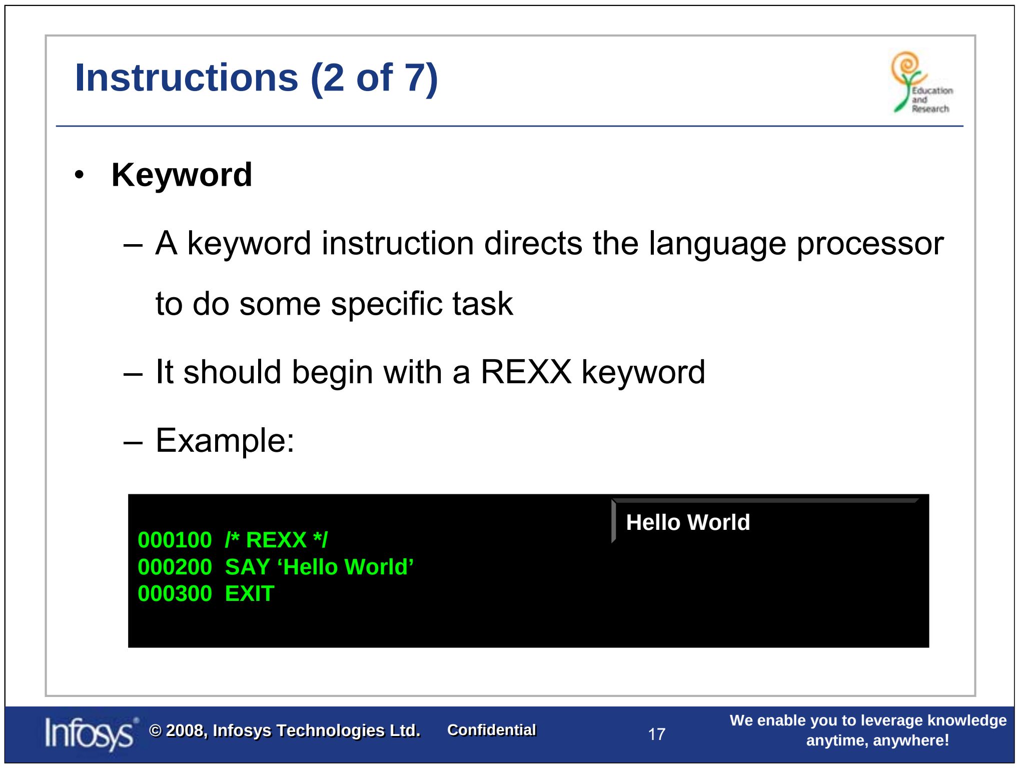
Instructions (1 of 7)

- Instructions in REXX can be of the following types
  - Keywords
  - Assignment statement
  - Command
  - Label
  - NULL : comment /* */
- All instructions except COMMANDS are processed by the language processor

© 2008, Infosys Technologies Ltd. Confidential 16 We enable you to leverage knowledge anytime, anywhere!

Instructions (1 of 7) - Notes

Instructions in REXX can be of the following types : Keyword instructions, assignment statements, null clauses, labels or commands. We shall look at each of these 5 types of instructions as we go ahead.



Instructions (2 of 7)

- **Keyword**
  - A keyword instruction directs the language processor to do some specific task
  - It should begin with a REXX keyword
  - Example:

```
000100  /* REXX */
000200  SAY 'Hello World'
000300  EXIT
```

```
Hello World
```

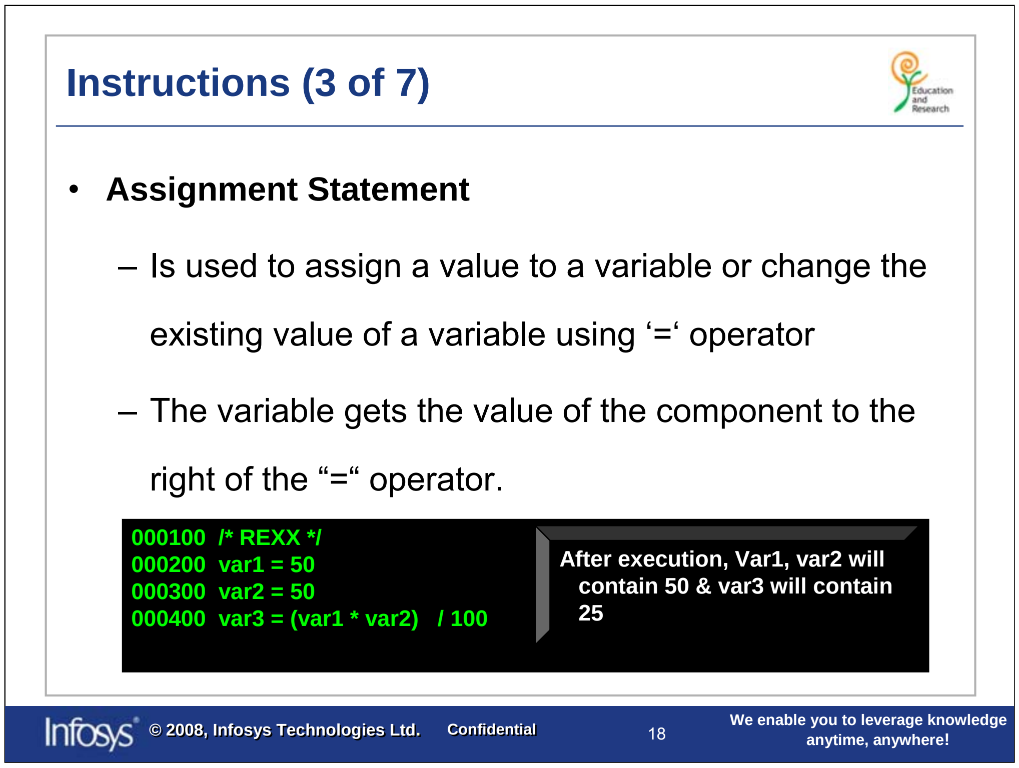
Instructions (2 of 7) - Notes

Keyword instructions consist of one or more clauses beginning with a REXX keyword. They are used to control external interfaces (printers, disks), logic flow and handle some data manipulation tasks.

Some of the keywords are DO, SAY, PARSE.

```
000100  /* REXX */
000200  SAY 'Hello World'
000300  EXIT
```

In the above example, statement 2 is a keyword instruction where 'SAY' is a keyword. This instruction displays the string 'Hello World' on the screen. Statement 3 is also a keyword instruction where EXIT is the keyword. This instruction ends the processing of the program.



Instructions (3 of 7)

- **Assignment Statement**
  - Is used to assign a value to a variable or change the existing value of a variable using ‘=‘ operator
  - The variable gets the value of the component to the right of the “=“ operator.

```
000100  /* REXX */
000200  var1 = 50
000300  var2 = 50
000400  var3 = (var1 * var2)   / 100
```

After execution, Var1, var2 will contain 50 & var3 will contain 25

© 2008, Infosys Technologies Ltd. Confidential 18 We enable you to leverage knowledge anytime, anywhere!

Instructions (3 of 7) - Notes

Assignment instructions are used to give variables new values. They consist of a single clause in the form VARIABLE = EXPRESSION.

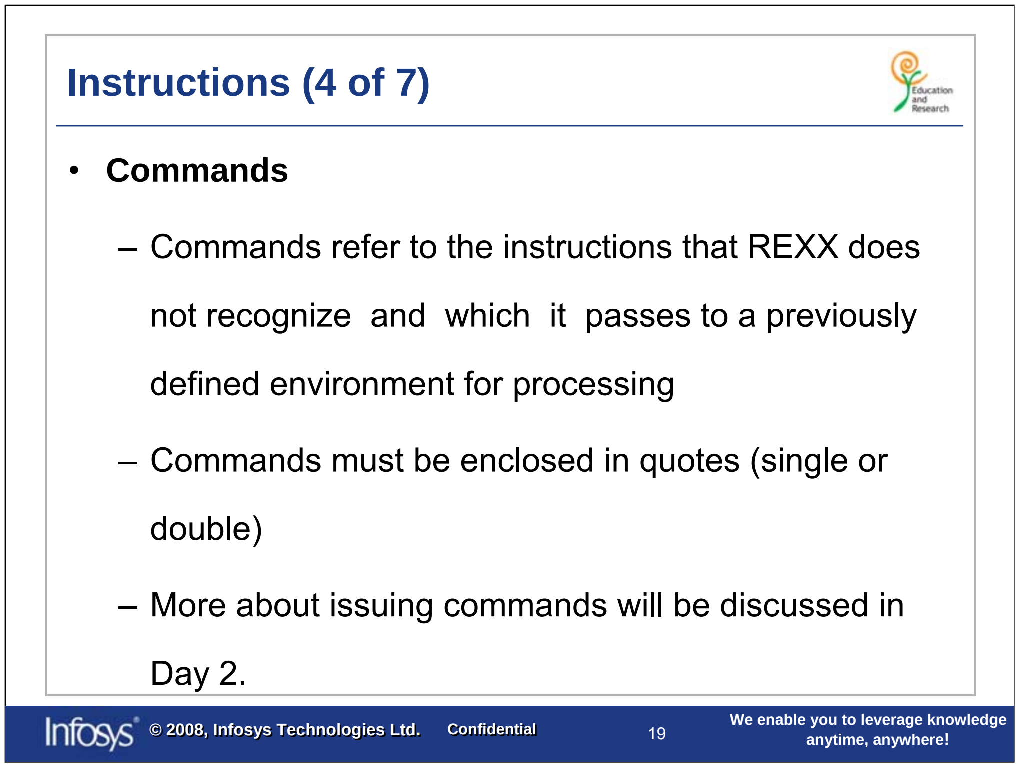
Example:

```
000100  /* REXX */
000200  var1 = 50
000300  var2 = 50
000400  var3 = (var1 * var2)   / 100
```

In the above example, statements 2,3 & 4 are all assignment statements. Statements 2 & 3 are simple assignment statements where a direct value is assigned to the variables. Thus, after the execution of statements 2 & 3, var1 & var2 will both contain 50. In statement 4, instead of a direct value, an expression has been assigned to the variable. During execution of this statement, the expression to the right of ‘=‘ operator is evaluated & it’s value will be assigned to var3. Thus var3 will contain 25.

Some more Examples:

1. VAR1 = VAR2 + (VAR3*FACTOR)
2. TRUE = ((A/B) > C)
3. PRFT = MSLP - (WSP*1.25)



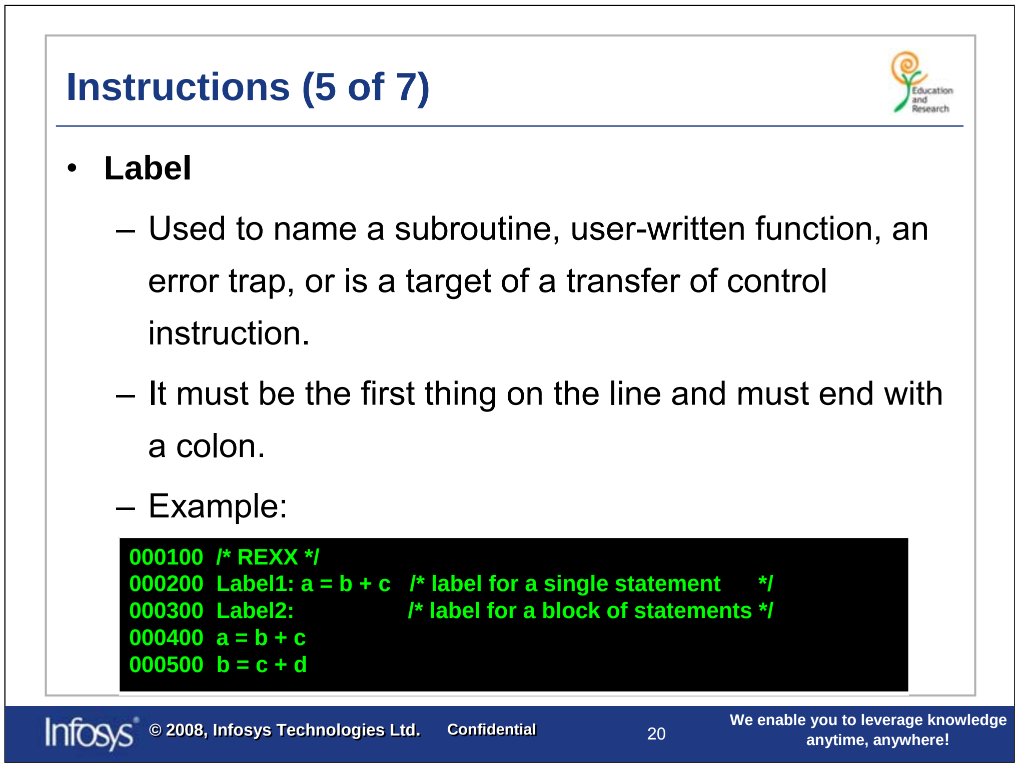
Instructions (4 of 7)

- **Commands**
  - Commands refer to the instructions that REXX does not recognize and which it passes to a previously defined environment for processing
  - Commands must be enclosed in quotes (single or double)
  - More about issuing commands will be discussed in Day 2.

Instructions (4 of 7) - Notes

Host environment commands can also be embedded in the REXX exec. These are referred to as commands. Commands are not processed by the REXX language processor but by the external environment. To issue a command, we must enclose it in quotes (either single or double).

Some examples of the host language commands for TSO are "MAKEBUF" , "DELSTACK" and "EXECIO" .



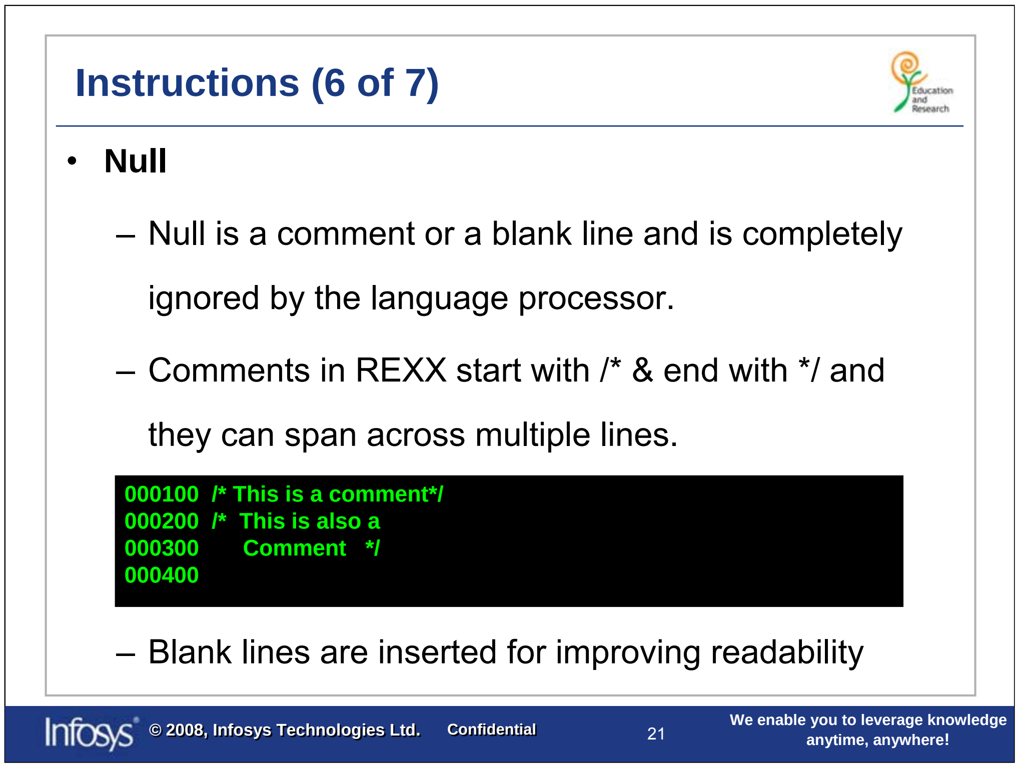
Instructions (5 of 7)

- **Label**
  - Used to name a subroutine, user-written function, an error trap, or is a target of a transfer of control instruction.
  - It must be the first thing on the line and must end with a colon.
  - Example:

```
/* REXX */
Label1: a = b + c   /* label for a single statement      */
Label2:             /* label for a block of statements */
a = b + c
b = c + d
```

Instructions (5 of 7) - Notes

Label instructions consist of a single symbol followed directly by a colon (:). They are used to identify the target of a CALL function , internal function calls, an error trap or a control transfer instruction. Labels must be the first to appear on the line. Examples of some labels are GO: and $$. CALL function and internal function calls will be covered later in the course.



Instructions (6 of 7)

- **Null**
  - Null is a comment or a blank line and is completely ignored by the language processor.
  - Comments in REXX start with /* & end with */ and they can span across multiple lines.

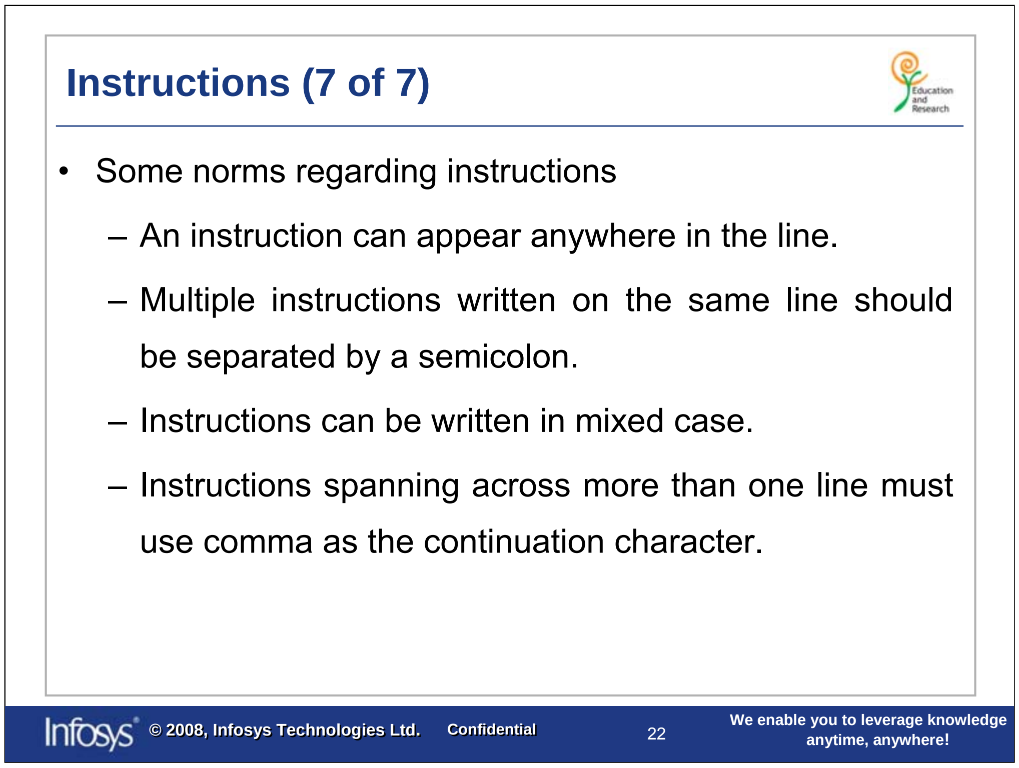
```
/* This is a comment*/
/*  This is also a
    Comment   */

```

  - Blank lines are inserted for improving readability

Instructions (6 of 7) - Notes

Null instructions are used mainly for documentation purposes . They also aid in debugging. These instructions are completely ignored by the processor . Null can either be a comment line or a blank line. A comment in REXX starts with /* and ends with */. It can span across multiple lines. Comments are inserted to provide additional information regarding the exec. Blank lines are inserted in between groups of instructions for improving the readability of the exec.

Example: /* This is a null statement */

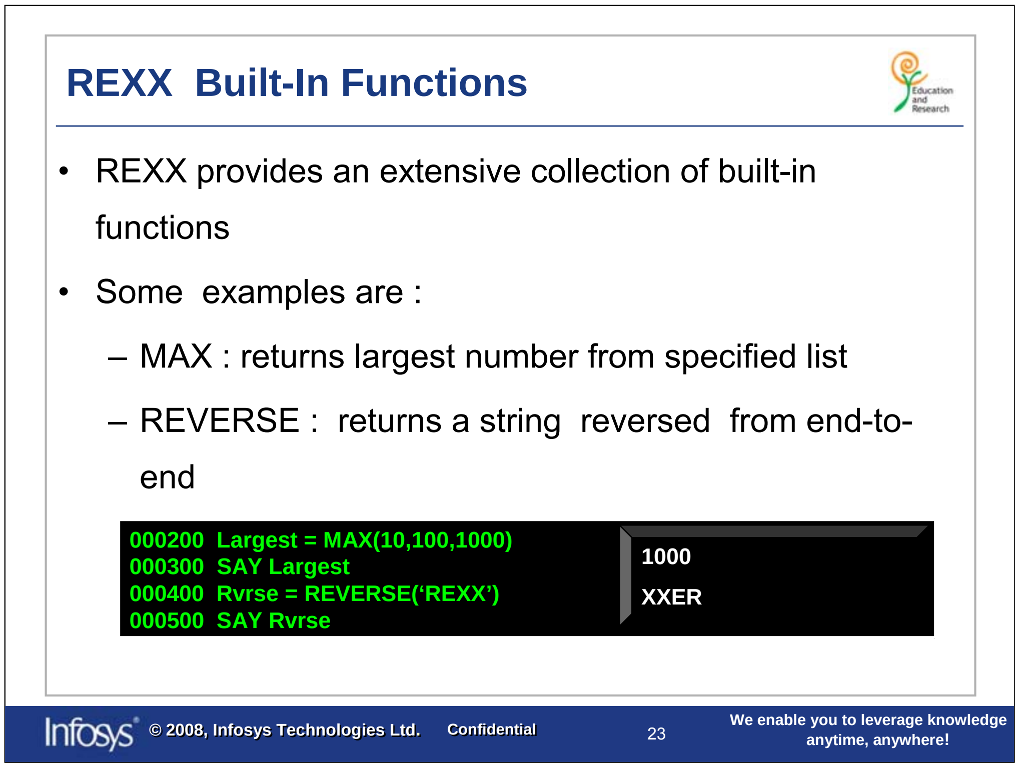


Instructions (7 of 7)

- Some norms regarding instructions
  - An instruction can appear anywhere in the line.
  - Multiple instructions written on the same line should be separated by a semicolon.
  - Instructions can be written in mixed case.
  - Instructions spanning across more than one line must use comma as the continuation character.

Instructions (7 of 7) - Notes

Some norms should be followed when coding the instructions in order to avoid errors. Normally, each instruction occupies one line .It can be anywhere on the line. If you want to put more than one instruction on a line, use a semicolon to separate them. The instruction can be coded in mixed case. If the instruction spans more than one line, put a comma at the end of the line. The comma indicates the continuation of the instruction on the next line.



REXX Built-In Functions

- REXX provides an extensive collection of built-in functions
- Some examples are :
  - MAX : returns largest number from specified list
  - REVERSE : returns a string reversed from end-to-end

```
Largest = MAX(10,100,1000)
SAY Largest
Rvrse = REVERSE('REXX')
SAY Rvrse
```

```
1000
XXER
```

REXX Built-in functions - NOTES

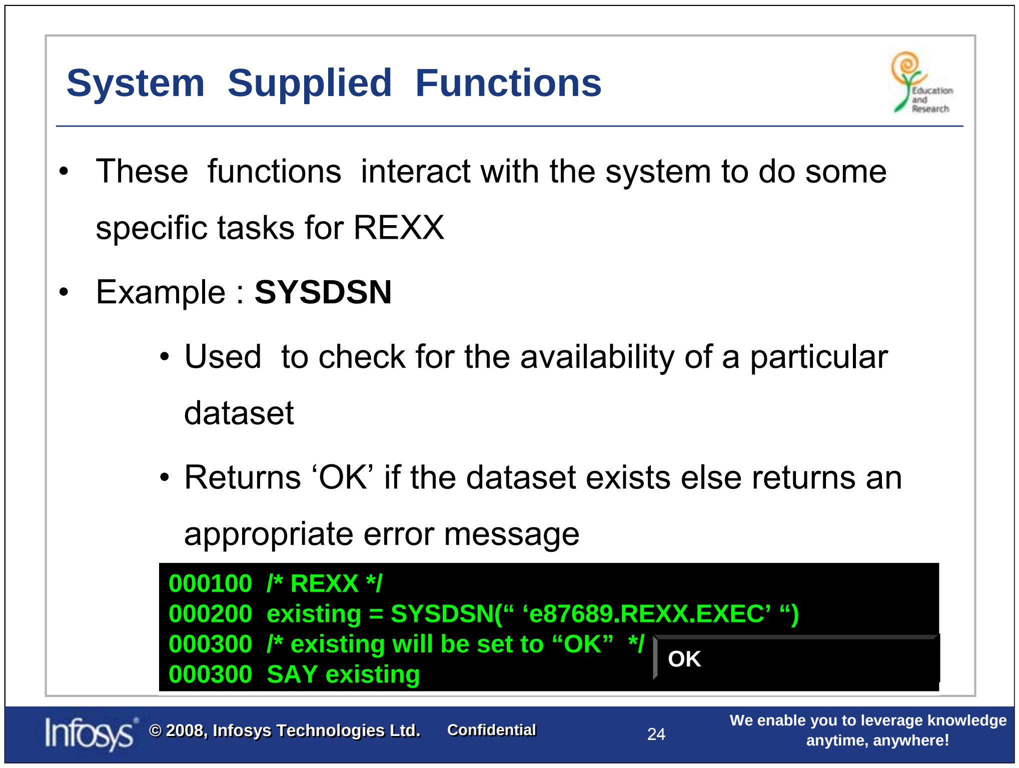
The REXX language comes with an extensive collection of powerful and convenient built-in functions . These functions are part of the language itself and are different from user-defined functions that can be coded by programmers for specific tasks or functionalities . User defined functions will be covered later in the course . Some examples of built-in functions are

•MAX – this function returns the largest number in a list or collection

•REVERSE – this function returns a string (reversed from end-to-end)

The purpose of built-in functions is to provide the programmer with some flexible processing options.

In the example shown in the previous slide, the MAX function will return 1000. And the REVERSE function will reverse the string 'REXX' & return 'XXER'



System Supplied Functions

- These functions interact with the system to do some specific tasks for REXX
- Example : **SYSDSN**
  - Used to check for the availability of a particular dataset
  - Returns 'OK' if the dataset exists else returns an appropriate error message

```
000100  /* REXX */
000200  existing = SYSDSN(" 'e87689.REXX.EXEC' ")
000300  /* existing will be set to "OK"  */
000300  SAY existing
```

```
OK
```

© 2008, Infosys Technologies Ltd. Confidential 24 We enable you to leverage knowledge anytime, anywhere!

System Supplied Functions - Notes

System supplied functions, like REXX built-in functions are also part of the language itself . However instead of providing some flexible processing options to the user , these functions interact with the system and accomplish some system-related tasks . The user can make use of these functions in any REXX program. Some examples are

:SYSDSN – used to determine whether a particular dataset is available for use . In the example shown in the slide, assume that the dataset 'e87689.REXX.EXEC' is existing. Thus the function SYSDSN will return "OK".

While using the SYSDSN function, the following points must be remembered:

1. If a fully qualified dataset name is passed in the function, then use 2 sets of quotes

   Example:

   var = SYSDSN(" 'E87689.REXX.EXEC' ")

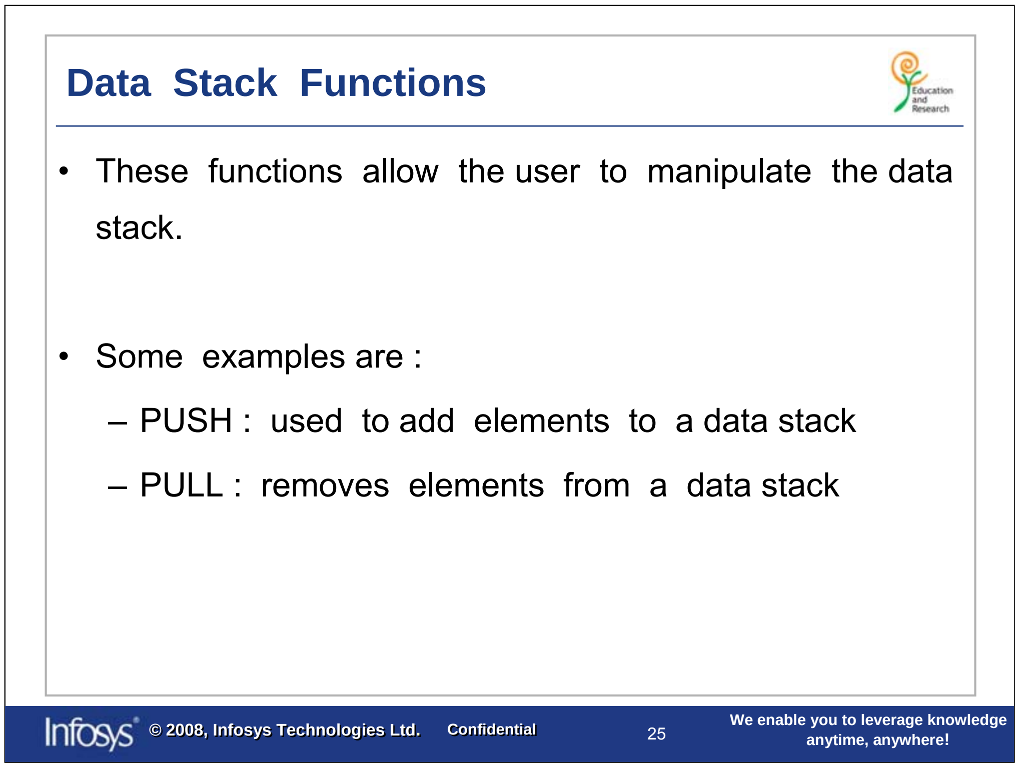
One set of double quotes to indicate a literal string to REXX and another set of single quotes to indicate a fully-qualified dataset name to TSO/E.

2. If the dataset name being passed is not fully-qualified, then use a single set of quotes or no quotes at all.

   Example:

   var = SYSDSN('REXX.EXEC') or

   var = SYSDSN(REXX.EXEC)



Data Stack Functions

- These functions allow the user to manipulate the data stack.
- Some examples are :
  - PUSH : used to add elements to a data stack
  - PULL : removes elements from a data stack

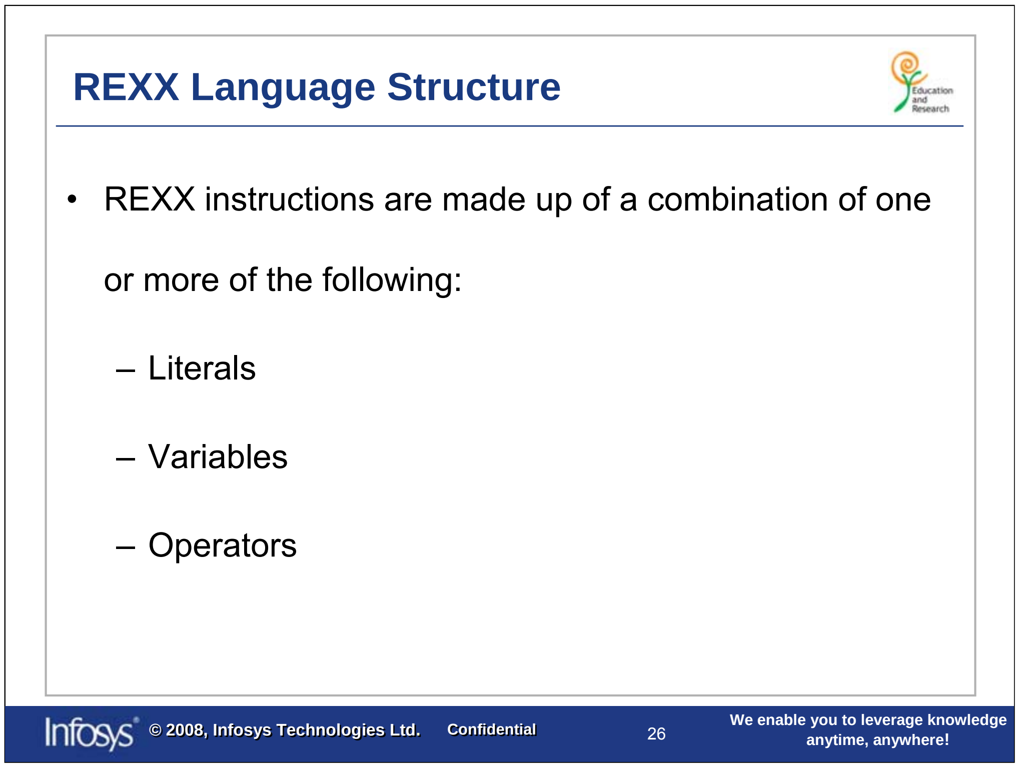
Data Stack Functions - Notes

Data stack functions allows a user to manipulate the data stack structure . The data stack structure is used to store data for I/O and other types of processing . Some examples are :

PUSH : used to add elements to a data stack

PULL : used to remove elements from a data stack

We shall discuss about the data stack structure in detail later in the course .

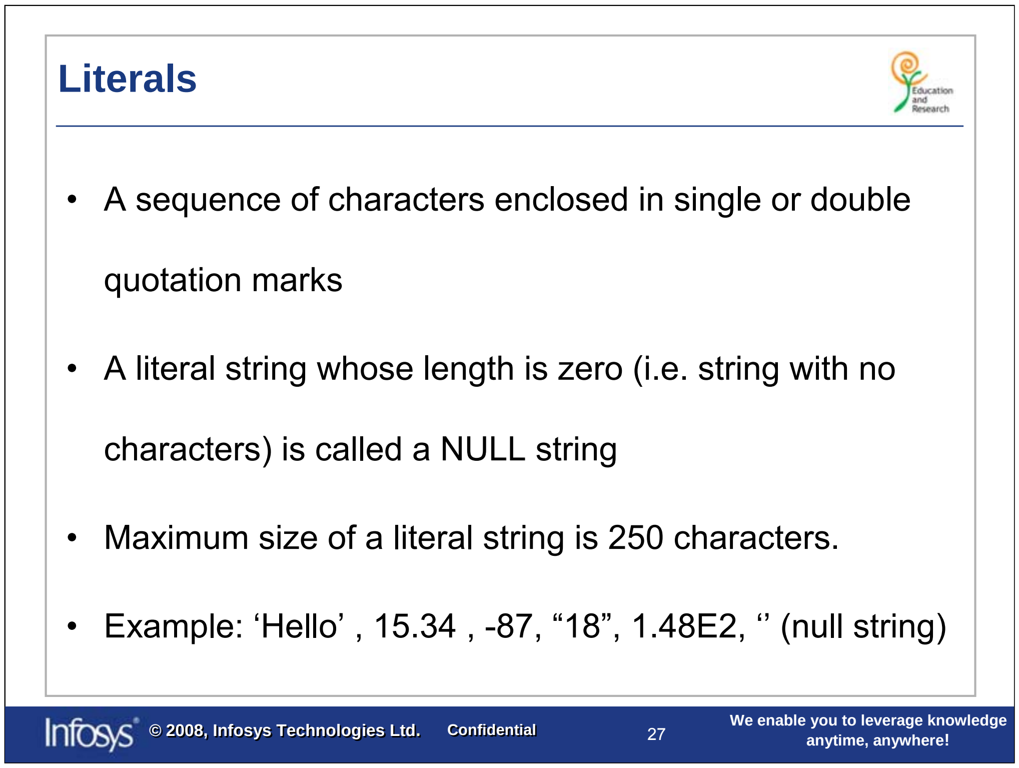


REXX Language Structure

- REXX instructions are made up of a combination of one or more of the following:
  - Literals
  - Variables
  - Operators

REXX Language Structure - Notes

The presence of various components in REXX make it a powerful programming language. Lets take a look into it's components in details. We shall now proceed to talk about literals ,variables and operators used in REXX.

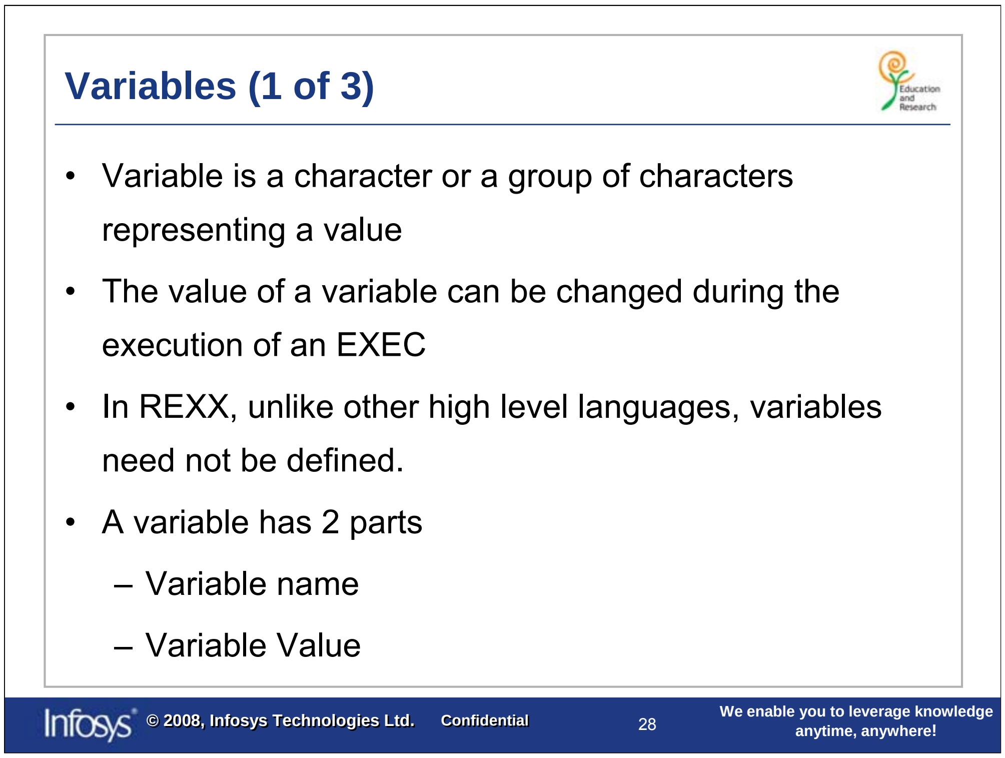


Literals

- A sequence of characters enclosed in single or double quotation marks
- A literal string whose length is zero (i.e. string with no characters) is called a NULL string
- Maximum size of a literal string is 250 characters.
- Example: 'Hello' , 15.34 , -87, "18", 1.48E2, '' (null string)

Literals - Notes

In REXX, literals should be preferably enclosed in quotes or apostrophes. A literal string with no characters in it is called a NULL string. Maximum size of a literal can be 250 characters. The numeric literals in REXX may have quotes or apostrophes, unlike COBOL. If you have an 'E' in a number, the number is supposed to be in scientific notation. Option between quotes and apostrophes provides flexibility of using them suitably like in this example: "7o' clock".



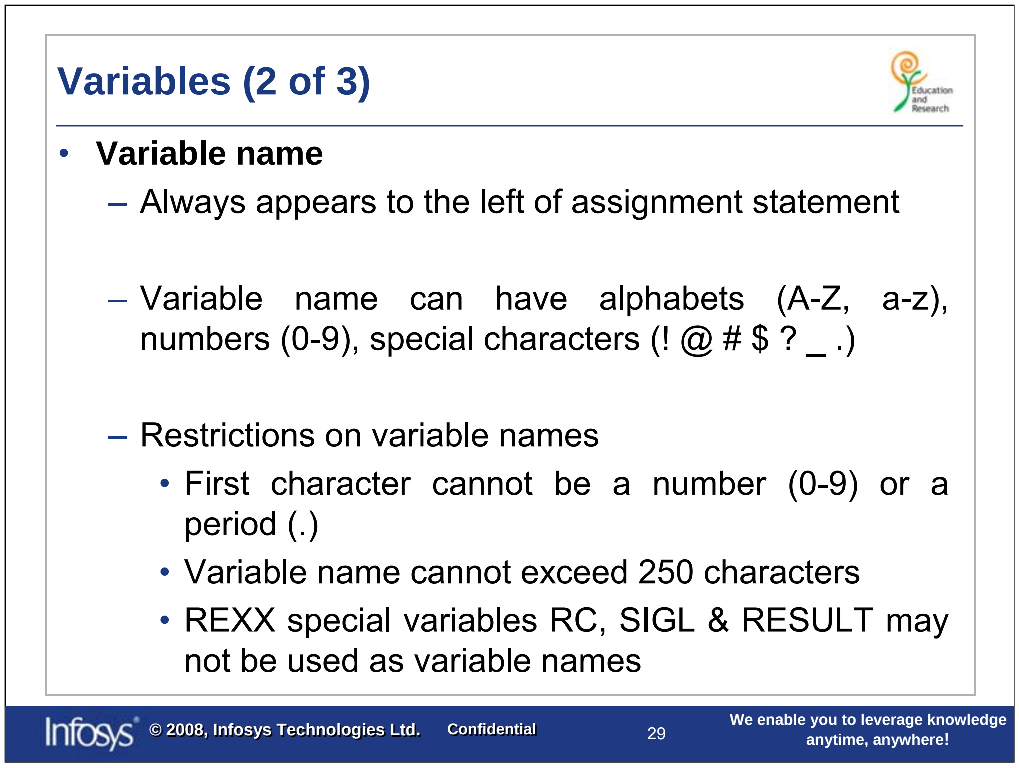
Variables (1 of 3)

- Variable is a character or a group of characters representing a value
- The value of a variable can be changed during the execution of an EXEC
- In REXX, unlike other high level languages, variables need not be defined.
- A variable has 2 parts
  - Variable name
  - Variable Value

Variables (1 of 3) - Notes

In high level languages, it is necessary to define the variables and initialize them. But this is not essential with REXX. REXX variables are declared and initialized any time you decide to use them. Hence REXX variables are called flexible variables.

A variable has two parts – variable name & variable value.



Variables (2 of 3)

- **Variable name**
  - Always appears to the left of assignment statement
  - Variable name can have alphabets (A-Z, a-z), numbers (0-9), special characters (! @ # $? _ .)
  - Restrictions on variable names
    - First character cannot be a number (0-9) or a period (.)
    - Variable name cannot exceed 250 characters
    - REXX special variables RC, SIGL & RESULT may not be used as variable names

Variables (2 of 3) - Notes

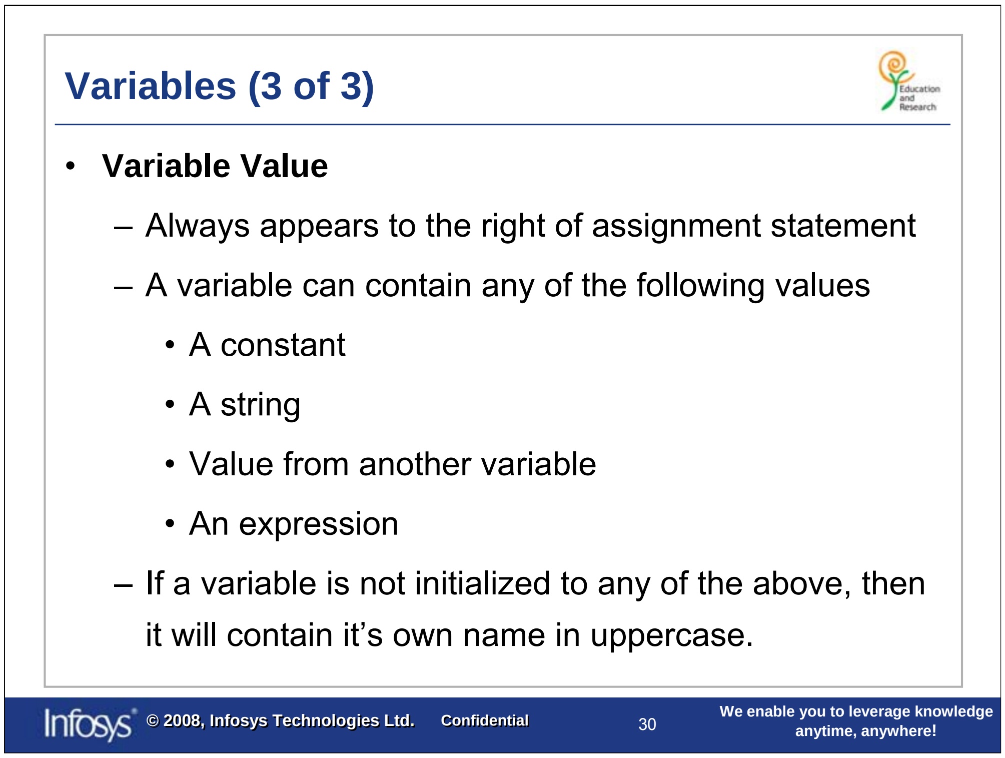
Variable name is always on the left of the assignment statement whereas value appears to the right.

Variables can be given any name up to 250 characters long. The naming can include A-Z, a-z,0-9 and special characters (! @ # $? _ .)

Example: sales_margin, A9, #82X.

However some restrictions do exist. The first character of a variable cannot be a number or a period and the keywords RC, SIGL or RESULT cannot be used as variable names. These are system defined special variables.

Some examples of invalid variable names are: 9A, .point, result



Variables (3 of 3)

- **Variable Value**
  - Always appears to the right of assignment statement
  - A variable can contain any of the following values
    - A constant
    - A string
    - Value from another variable
    - An expression
  - If a variable is not initialized to any of the above, then it will contain it's own name in uppercase.

Infosys © 2008, Infosys Technologies Ltd. Confidential 30 We enable you to leverage knowledge anytime, anywhere!

Variables (3 of 3) - Notes

The variables are initialized to their UPPER case names.

Values in the variable can be any of the following:

1.A constant – is a number that is expressed in any of the following forms

a. An integer – Ex: 5

b. A decimal – Ex: 5.2

c. A floating point number – Ex: 5.2E2, which is the same as 5.2 * 10^2 (520)

d. A signed number – Ex: -5.2

e. A string constant – Ex: ' 5'

2. A string – is one which contains one or more words, which may or may not be enclosed within quotes

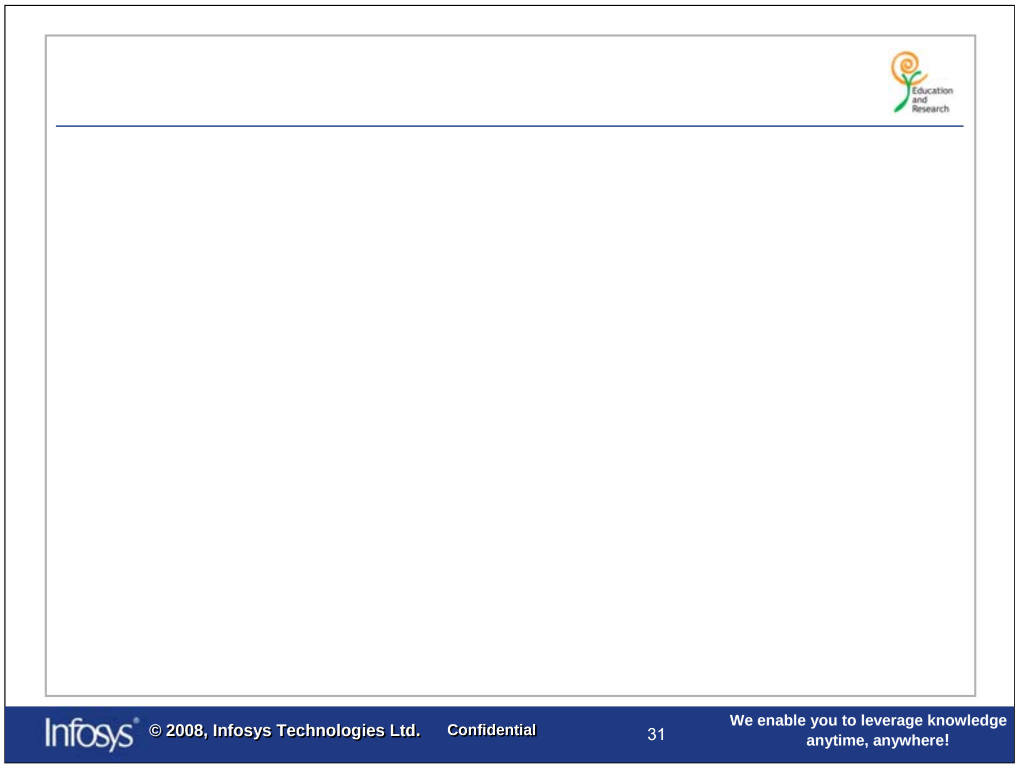
Ex: 'This is a string'.

This is also a string.

3. Value from another variable

Ex: Var1 = Var2

In the above example, the value of Var2 gets assigned to Var1. Contents of var2 remain unchanged.



© 2008, Infosys Technologies Ltd. Confidential 31 We enable you to leverage knowledge anytime, anywhere!

4. An expression – is something which has to be evaluated

Ex: Var1 = 5 + 5 – 1

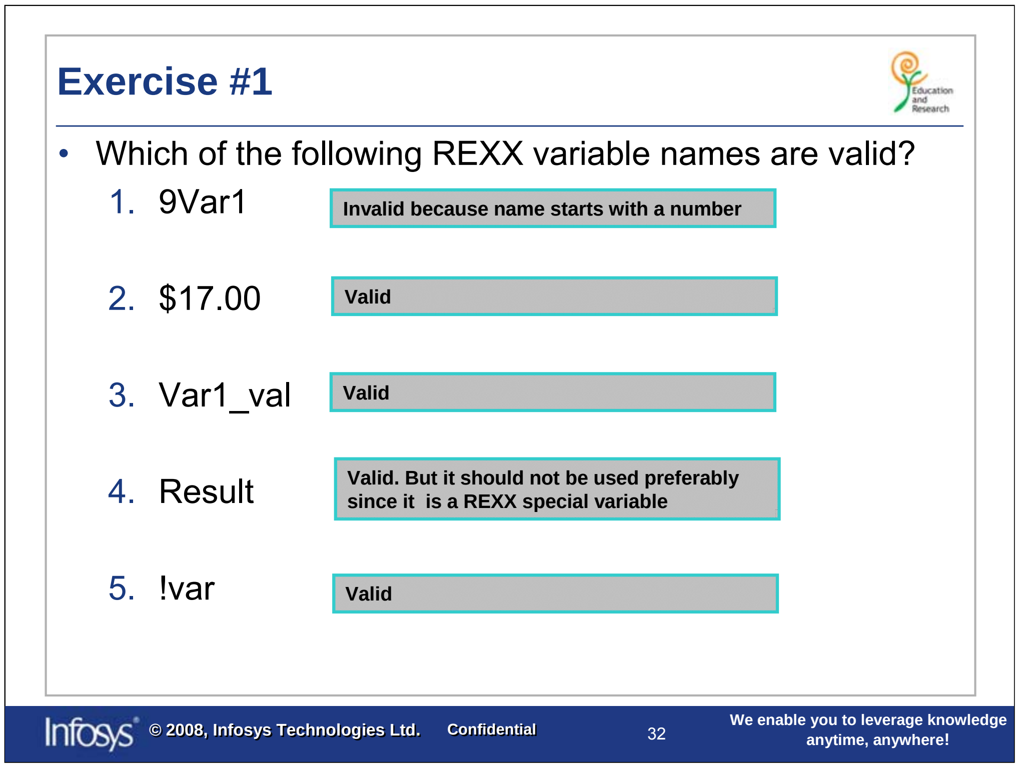
The expression ‘5 + 5 – 1’ is evaluated to 9 and the result ‘9’ is assigned to Var1.

One more important thing to note here is that, if a variable is not assigned any of the above values, then the variable will contain it’s own name translated to upper case.

For Ex: Assume ‘item1’ is not assigned any value & the following statement is issued within the EXEC.

```
/* REXX */
SAY item1
```

The output would be ITEM1.



Exercise #1

- Which of the following REXX variable names are valid?
  1. 9Var1 — Invalid because name starts with a number
  2. $17.00 — Valid
  3. Var1_val — Valid
  4. Result — Valid. But it should not be used preferably since it is a REXX special variable
  5. !var — Valid

Infosys © 2008, Infosys Technologies Ltd. Confidential 32 We enable you to leverage knowledge anytime, anywhere!

Exercise #1 – NOTES

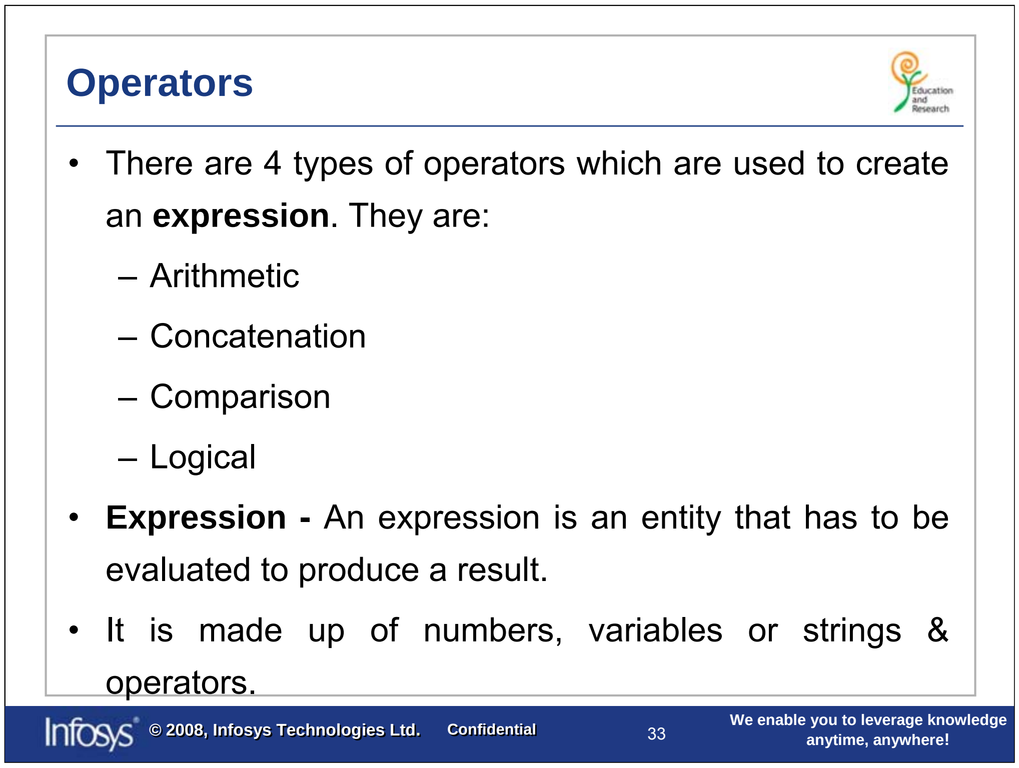
1. 9Var1 – is invalid because the variable name starts with a number

2. $17.00 – is valid since $ is a valid special character for REXX variable name

3. Var1_val – is also valid since '_' is also a valid special character for REXX variable name

4. Result – is valid but it's use as a variable is not recommended since it is a special system variable in REXX

5. !var – is also valid since it contains ! & letters of the alphabet, all of which are valid characters for REXX variable names.



Operators

- There are 4 types of operators which are used to create an **expression**. They are:
  – Arithmetic
  – Concatenation
  – Comparison
  – Logical
- **Expression -** An expression is an entity that has to be evaluated to produce a result.
- It is made up of numbers, variables or strings & operators.

Infosys © 2008, Infosys Technologies Ltd. Confidential 33 We enable you to leverage knowledge anytime, anywhere!

Operators - Notes

There are 4 types of operators which are used to create an expression. They are:

Arithmetic

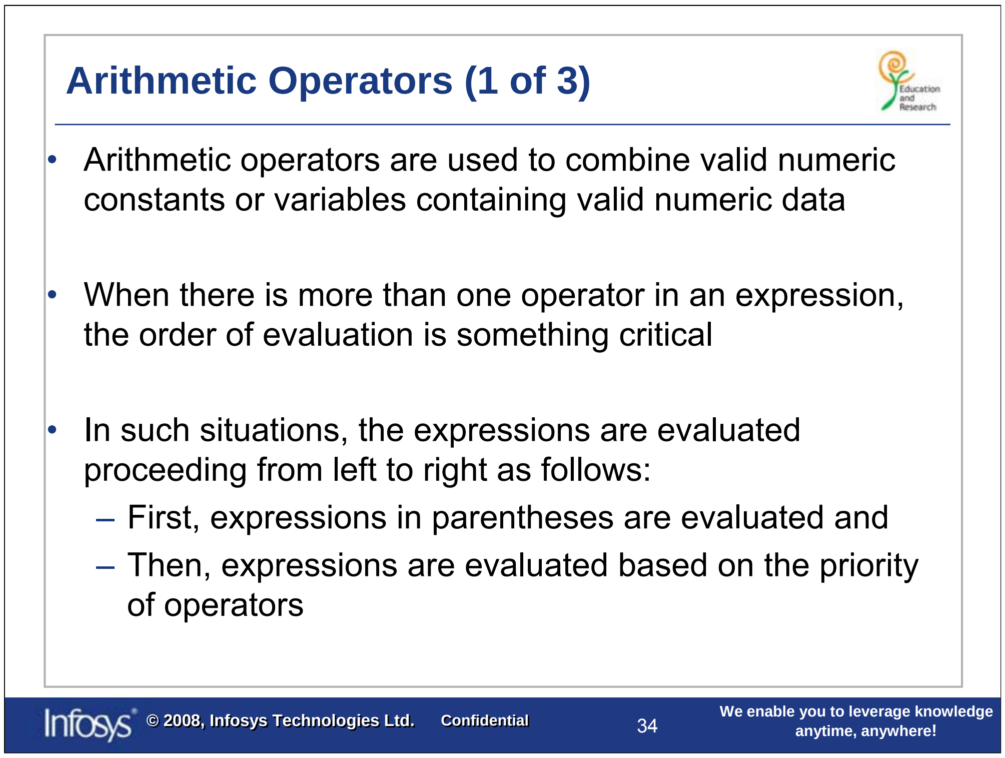
Concatenation

Comparison

Logical

Expression - An expression is an entity which needs to be evaluated. It is made up of numbers , variables or strings and zero or more operators.

The operation to be performed on the numbers, variables & strings in the expression is decided by operators which can be arithmetic, comparison, logical, or concatenation operators. Lets take a look at each if these operators in detail.



Arithmetic Operators (1 of 3)

- Arithmetic operators are used to combine valid numeric constants or variables containing valid numeric data
- When there is more than one operator in an expression, the order of evaluation is something critical
- In such situations, the expressions are evaluated proceeding from left to right as follows:
  - First, expressions in parentheses are evaluated and
  - Then, expressions are evaluated based on the priority of operators

Infosys © 2008, Infosys Technologies Ltd. Confidential 34 We enable you to leverage knowledge anytime, anywhere!

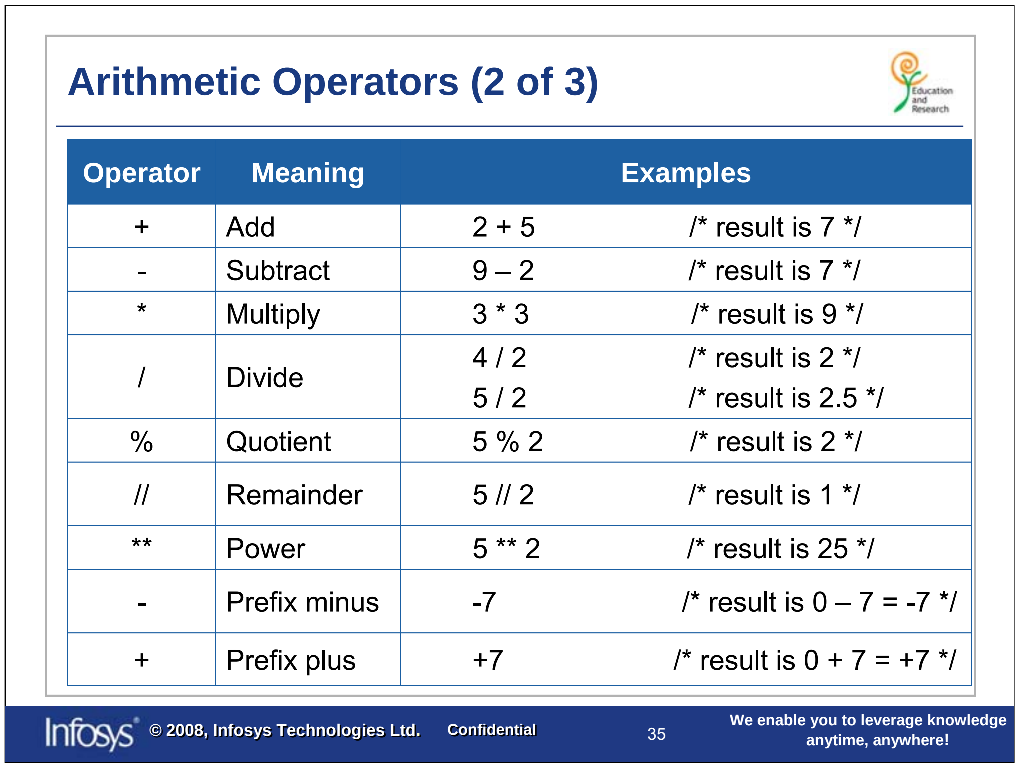
Arithmetic Operators (1 of 3) - Notes

If there are more than one operators in an expression, then the order in which the expression is evaluated depends upon priority. Highest priority is given to expressions within parentheses. This means that the expressions within parentheses are evaluated first. Next, the expressions are evaluated based on the operator priority. This means that the operators with high priority are evaluated before operators with low priority.

For example, which operation is performed first in this example? 9 - 2 * (9 / 3) – 1.

Arithmetic operator priority is as follows, with the highest first:

– +	Prefix operators
**	Power (exponential)
/ % //	Multiplication and division
+ -	Addition and subtraction

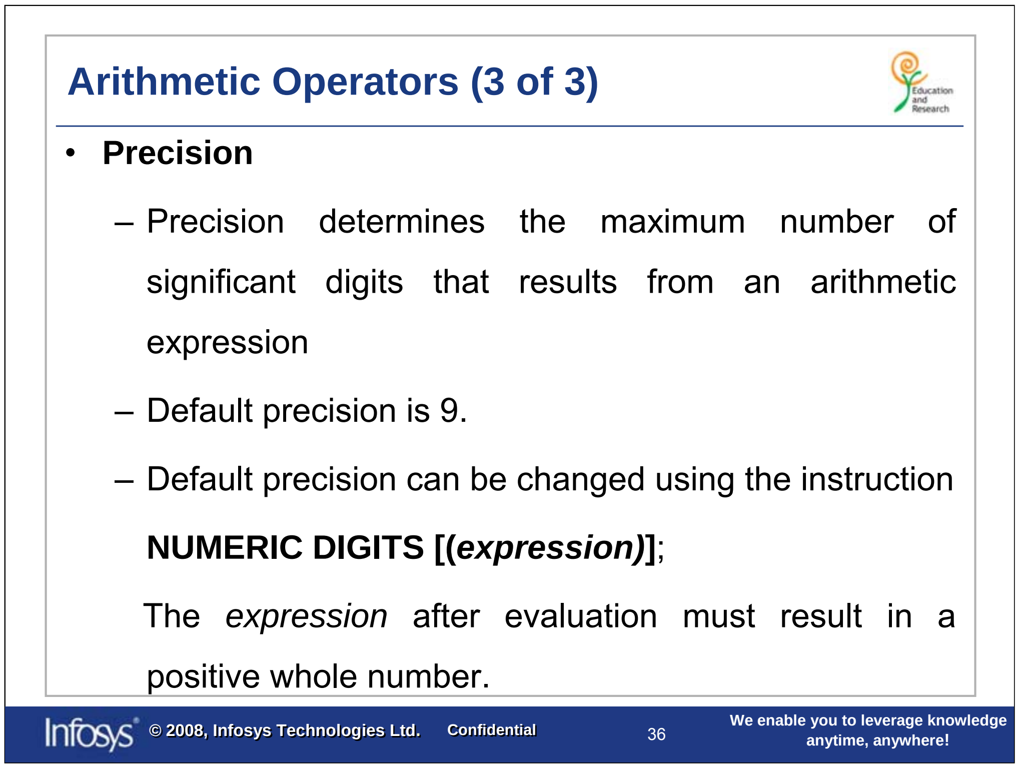


Arithmetic Operators (2 of 3)

Operator	Meaning	Examples	
+	Add	2 + 5	/* result is 7 */
-	Subtract	9 – 2	/* result is 7 */
*	Multiply	3 * 3	/* result is 9 */
/	Divide	4 / 2 5 / 2	/* result is 2 */ /* result is 2.5 */
%	Quotient	5 % 2	/* result is 2 */
//	Remainder	5 // 2	/* result is 1 */
**	Power	5 ** 2	/* result is 25 */
-	Prefix minus	-7	/* result is 0 – 7 = -7 */
+	Prefix plus	+7	/* result is 0 + 7 = +7 */

Arithmetic Operators (2 of 3) - Notes

Arithmetic operators are used to operate on valid numeric constants or on variables which contain valid numeric data. The adjoining table summaries the list of arithmetic operators and their meaning along with examples.



Arithmetic Operators (3 of 3)

- **Precision**
  - Precision determines the maximum number of significant digits that results from an arithmetic expression
  - Default precision is 9.
  - Default precision can be changed using the instruction

    NUMERIC DIGITS [(*expression*)];

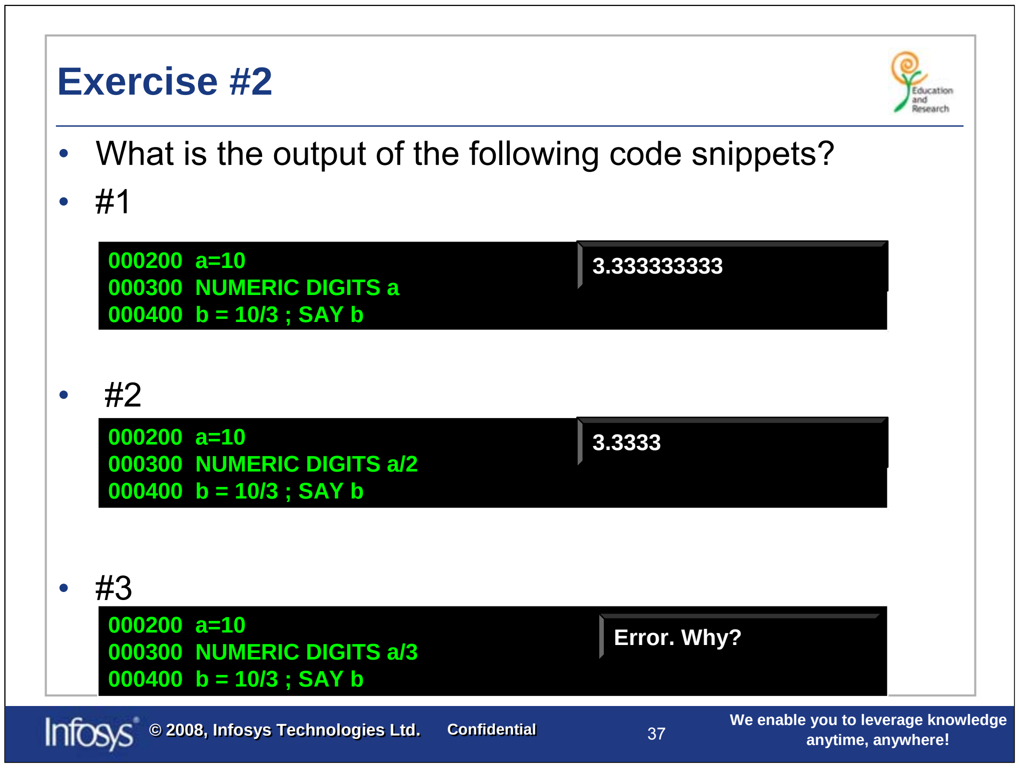
    The *expression* after evaluation must result in a positive whole number.

Arithmetic Operators (3 of 3) -Notes

The instruction which determines the maximum number of significant digits that appear in the result after an arithmetic expression is evaluated is

NUMERIC DIGITS [expression] is The expression after evaluation must result in a positive whole number . This defines the precision (number of significant digits) to which arithmetic calculations will be carried out . If expression is not specified, then the default precision (9) is used.

Note: In the syntax, elements in square brackets [(expression)] is optional. Square brackets are used just to indicate that whatever comes within them is optional; square brackets are not a part of the actual syntax.



Exercise #2

- What is the output of the following code snippets?
- #1

```
000200  a=10
000300  NUMERIC DIGITS a
000400  b = 10/3 ; SAY b
```

```
3.333333333
```

- #2

```
000200  a=10
000300  NUMERIC DIGITS a/2
000400  b = 10/3 ; SAY b
```

```
3.3333
```

- #3

```
000200  a=10
000300  NUMERIC DIGITS a/3
000400  b = 10/3 ; SAY b
```

Error. Why?

Exercise #2 - Notes

Example #1

```
000200  a=5
000300  NUMERIC DIGITS a
000400  b = 10/3 ; SAY b
```

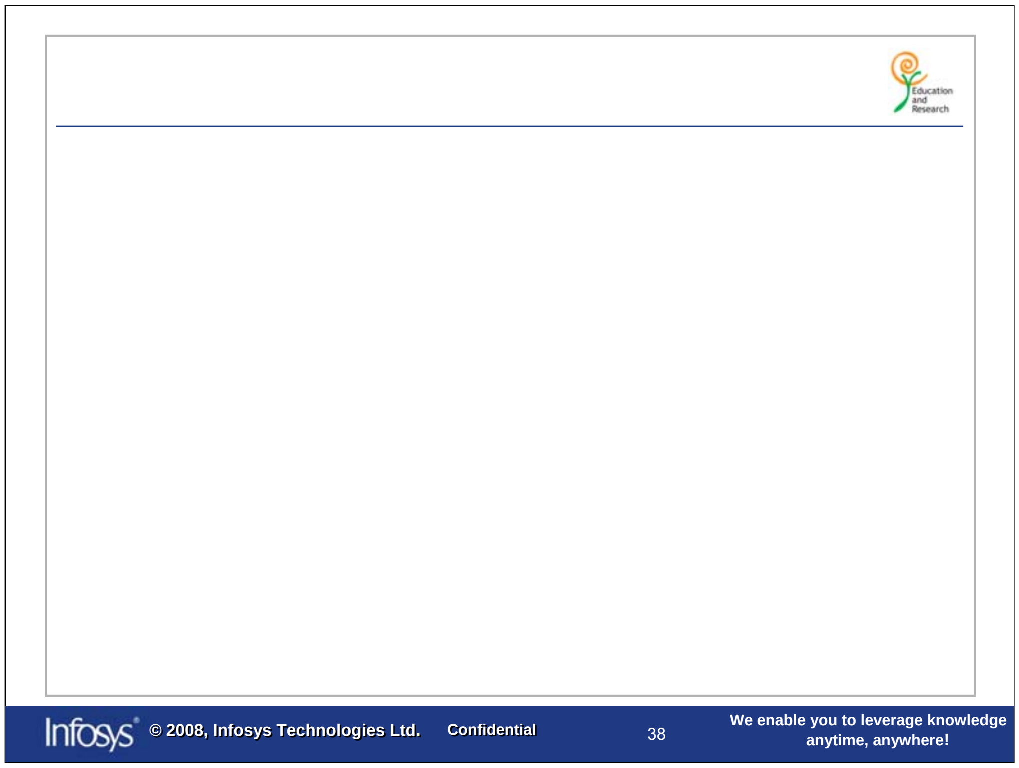
O/P: **3.333333333**

In the above example, the default precision of 9 is changed to 10 by the NUMERIC DIGITS instruction. Hence the
the division of 10 by 3 results in **3.333333333.**

Example #2

```
000200  a=10
000300  NUMERIC DIGITS a/2
000400  b = 10/3 ; SAY b
```

O/P: **3.3333**
In the above example, default precision of 9 is changed to 5 (result from the expression a/2). Hence the output
contains 5 significant digits.

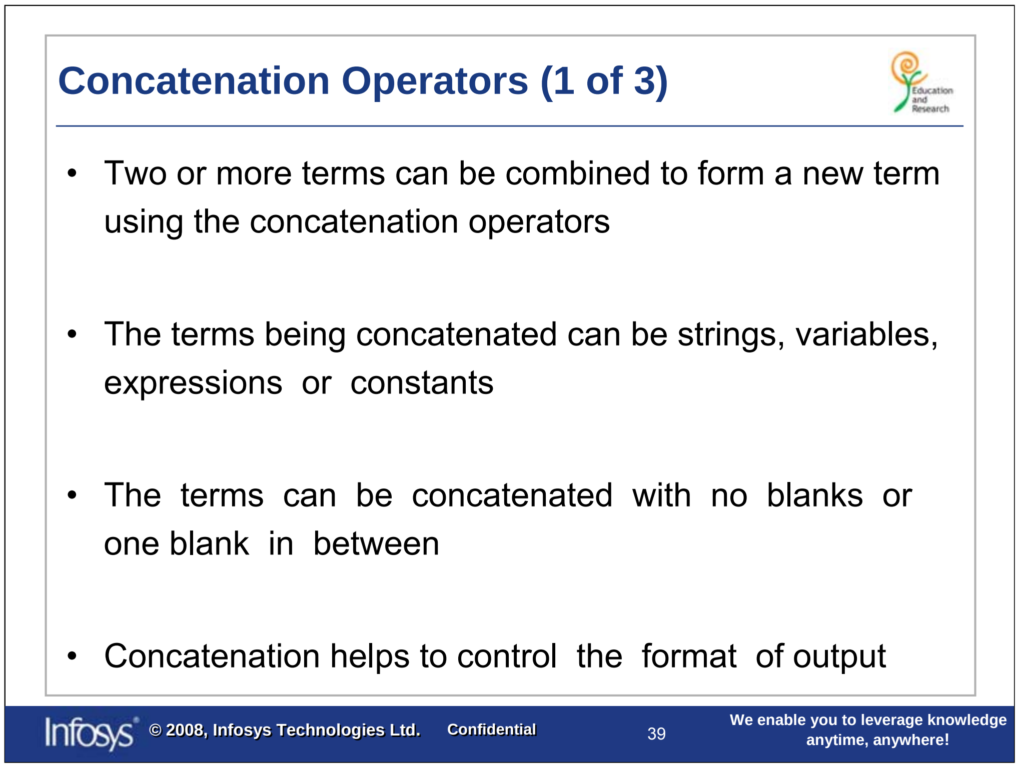


Example #3

```
000200  a=10
000300  NUMERIC DIGITS a/3
000400  b = 10/3 ; SAY b
```

O/P: Error.

In the above example, a/3 will not result in a whole number. Hence, the above code snippet will cause a run time error.



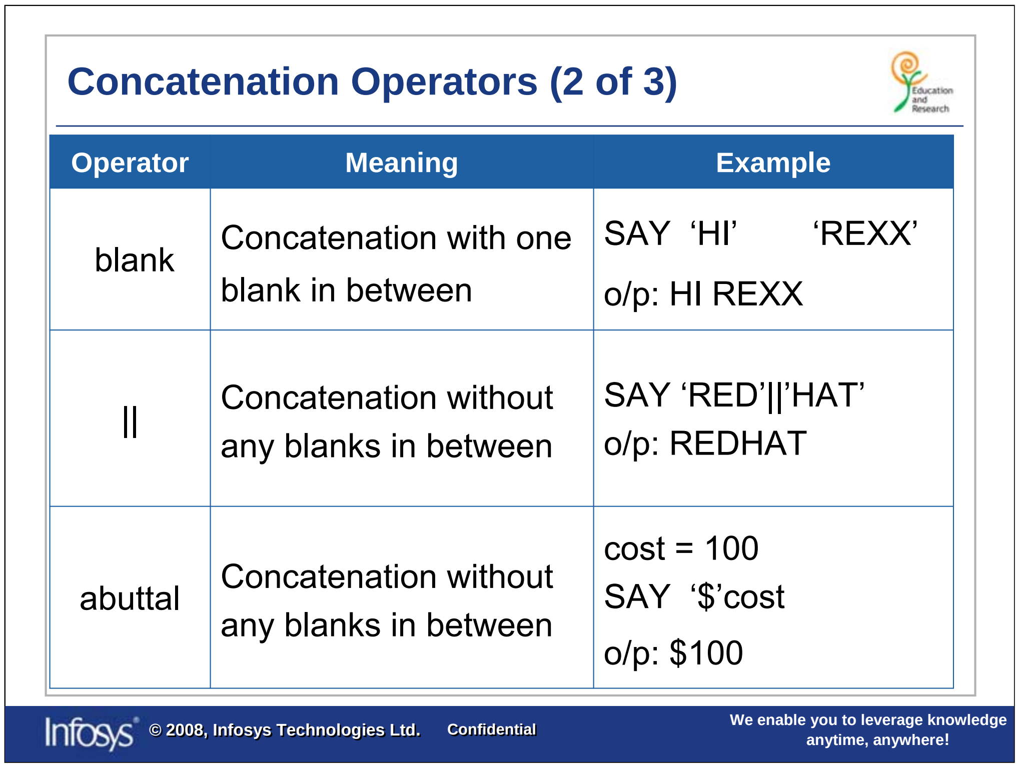
Concatenation Operators (1 of 3)

- Two or more terms can be combined to form a new term using the concatenation operators
- The terms being concatenated can be strings, variables, expressions or constants
- The terms can be concatenated with no blanks or one blank in between
- Concatenation helps to control the format of output

Infosys © 2008, Infosys Technologies Ltd. Confidential 39 We enable you to leverage knowledge anytime, anywhere!

Concatenation Operators (1 of 3) - Notes

Two or more terms can be combined to form a new term using the concatenation operators. The terms being concatenated can be strings, variables, expressions or constants. Concatenation is of very much use while formatting the output. We shall look at different concatenation operators and some examples in the subsequent slides.



Concatenation Operators (2 of 3)

Operator	Meaning	Example
blank	Concatenation with one blank in between	SAY ‘HI’ ‘REXX’ o/p: HI REXX
||	Concatenation without any blanks in between	SAY ‘RED’||’HAT’ o/p: REDHAT
abuttal	Concatenation without any blanks in between	cost = 100 SAY ‘$’cost o/p: $100

Infosys © 2008, Infosys Technologies Ltd. Confidential

We enable you to leverage knowledge anytime, anywhere!

Concatenation Operators (2 of 3) - Notes

REXX provides a blank operator which is used to concatenate terms with one blank in between. Even though multiple blank operators (or simply blanks) are used in the expression, the final output will contain only one blank between any two terms.

E.g. SAY ‘HELLO’ ‘REXX’

The output displayed would be

HELLO REXX

There would be only one blank between the terms HELLO and REXX during display.

The next operator || concatenates terms without any blanks in between

e.g. SAY ‘HI’||’REXX’

The output would be

HIREXX

The abuttal operator concatenates terms without any blanks in between.

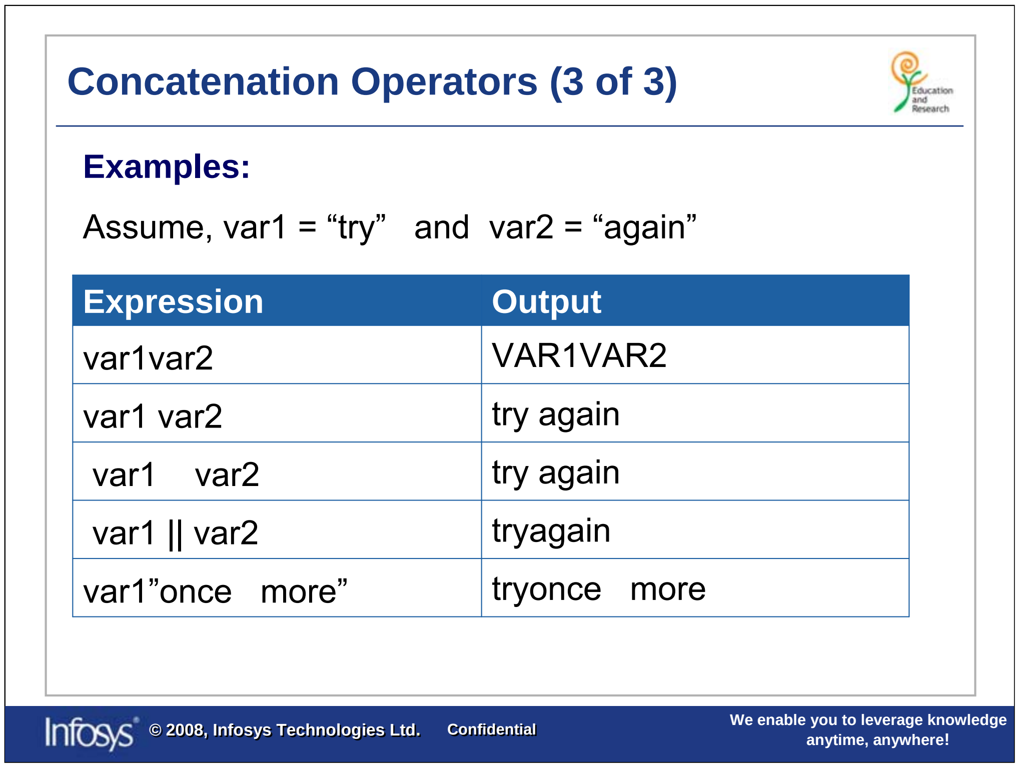
E.g. Money = 200

SAY ‘RS.’Money

Output :

RS.200

We shall look at some more examples in the next slide.



Concatenation Operators (3 of 3)

Examples:

Assume, var1 = “try”   and var2 = “again”

Expression	Output
var1var2	VAR1VAR2
var1 var2	try again
var1 var2	try again
var1 || var2	tryagain
var1”once more”	tryonce more

Concatenation Operators (3 of 3) - Notes

Here var1 = “try” and var2 = “again”

In the first example , var1var2 is taken as one variable . So as we have not assigned any value to it , it is translated to upper case and displayed as VAR1VAR2

In the second example, the output would be

try again

There would be one blank between the two terms.

In the third example , though the two terms are separated by more than one blank , it defaults to one blank on display. Hence the output is again

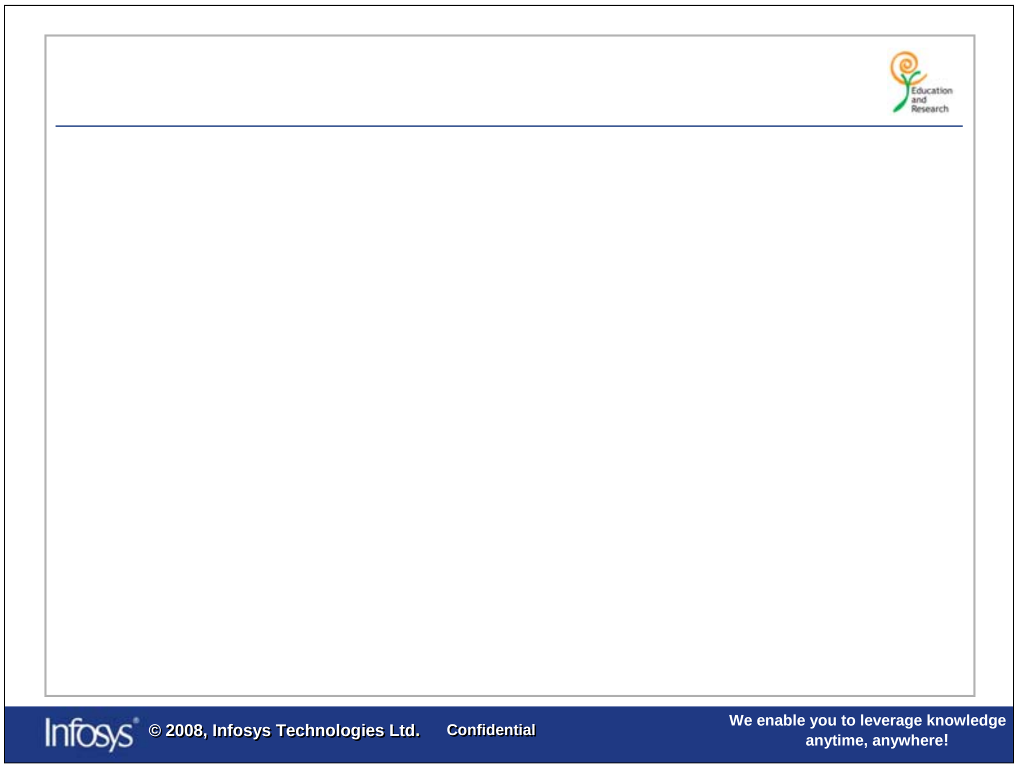
try again

Here again , there is one blank between the two terms .

In the fourth example , the || operator is used and so the terms are concatenated without any blanks between them .

Hence the output is

tryagain



Education and Research

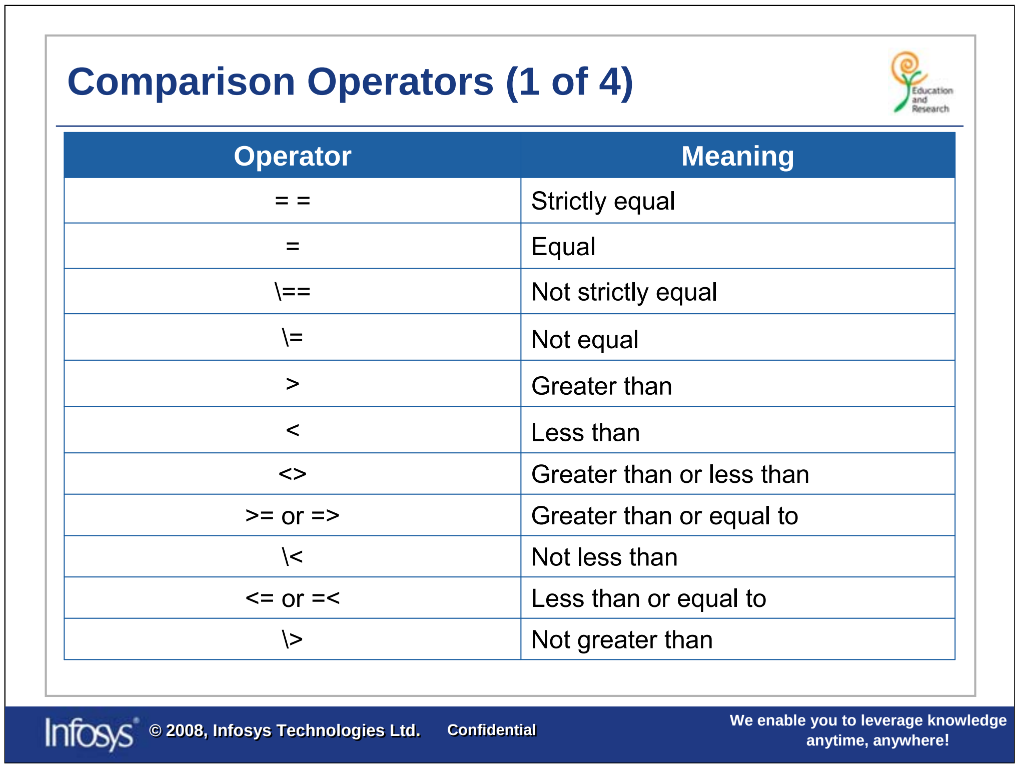
© 2008, Infosys Technologies Ltd. Confidential

We enable you to leverage knowledge anytime, anywhere!

In the fifth example in the previous slide, the two terms are abutted and so there is no blank between “try” and “once   more”

Hence the output is

```
tryonce   more
```

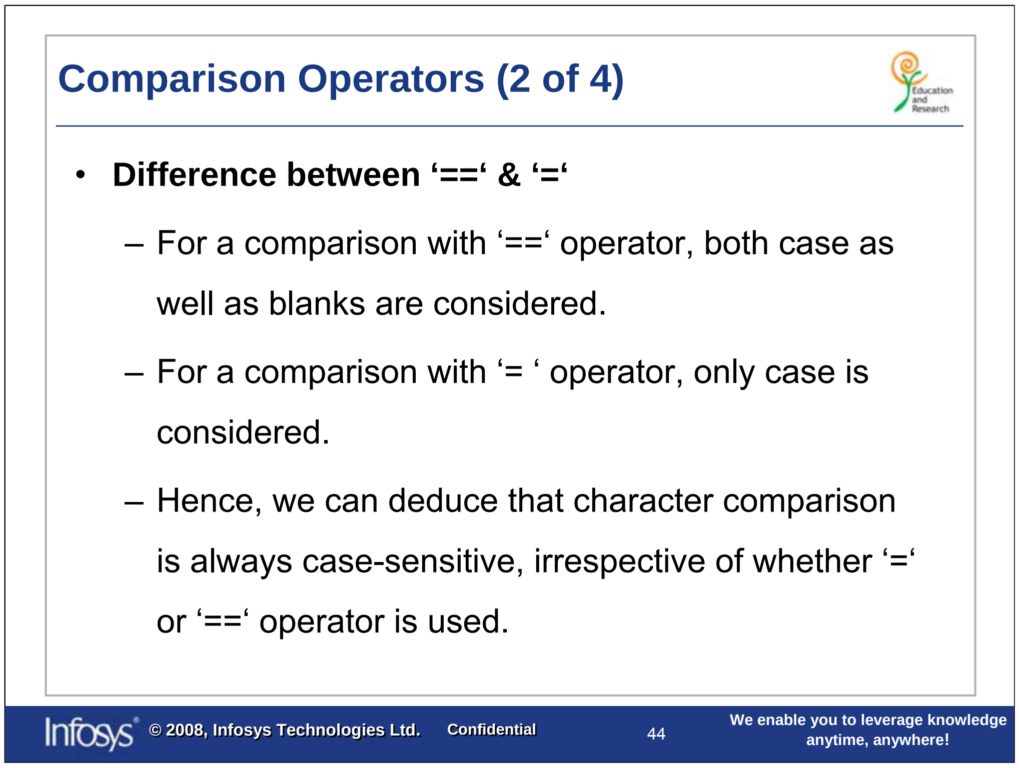


Comparison Operators (1 of 4)

Operator	Meaning
= =	Strictly equal
=	Equal
\==	Not strictly equal
\=	Not equal
>	Greater than
<	Less than
<>	Greater than or less than
>= or =>	Greater than or equal to
\<	Not less than
<= or =<	Less than or equal to
\>	Not greater than

Comparison Operators (1 of 4) - Notes

Expressions that use comparison operators return either 1, which represents true, or 0, which represents false. The adjoining table summarizes the list of comparison operators and their meanings.



Comparison Operators (2 of 4)

- **Difference between '==' & '='**
  - For a comparison with '==' operator, both case as well as blanks are considered.
  - For a comparison with '= ' operator, only case is considered.
  - Hence, we can deduce that character comparison is always case-sensitive, irrespective of whether '=' or '==' operator is used.

Comparison Operators (2 of 4) - Notes

The operator == means two items are exactly equal in all the aspects including blanks and case.

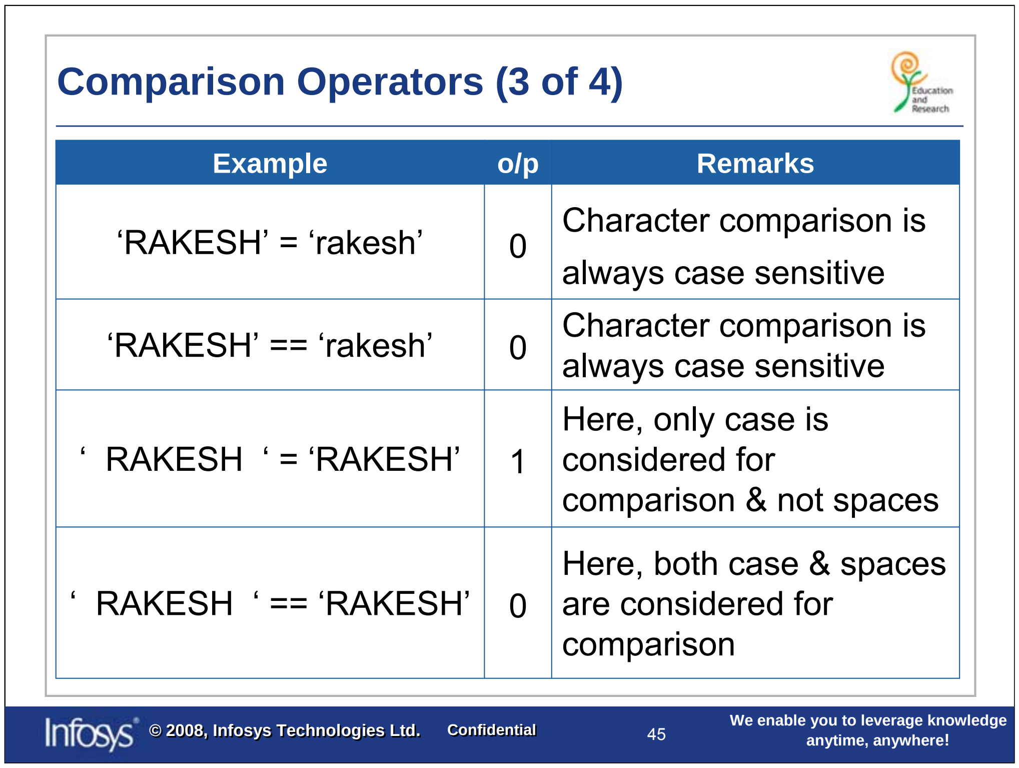
For example "   Rakesh" is not == "Rakesh" or "rakesh" or "RAKESH"

Similarly 000.56 is not == .56.

However 000.56 = .56

But please note that though "   RAJ" = "RAJ", "RAJ" is not = "raj" .

This is because character comparison is case sensitive.



Comparison Operators (3 of 4)

Example	o/p	Remarks
‘RAKESH’ = ‘rakesh’	0	Character comparison is always case sensitive
‘RAKESH’ == ‘rakesh’	0	Character comparison is always case sensitive
‘ RAKESH ‘ = ‘RAKESH’	1	Here, only case is considered for comparison & not spaces
‘ RAKESH ‘ == ‘RAKESH’	0	Here, both case & spaces are considered for comparison

Comparison Operators (3 of 4) – Notes

‘RAKESH’ = ‘rakesh’ & ‘RAKESH’ == ‘rakesh’

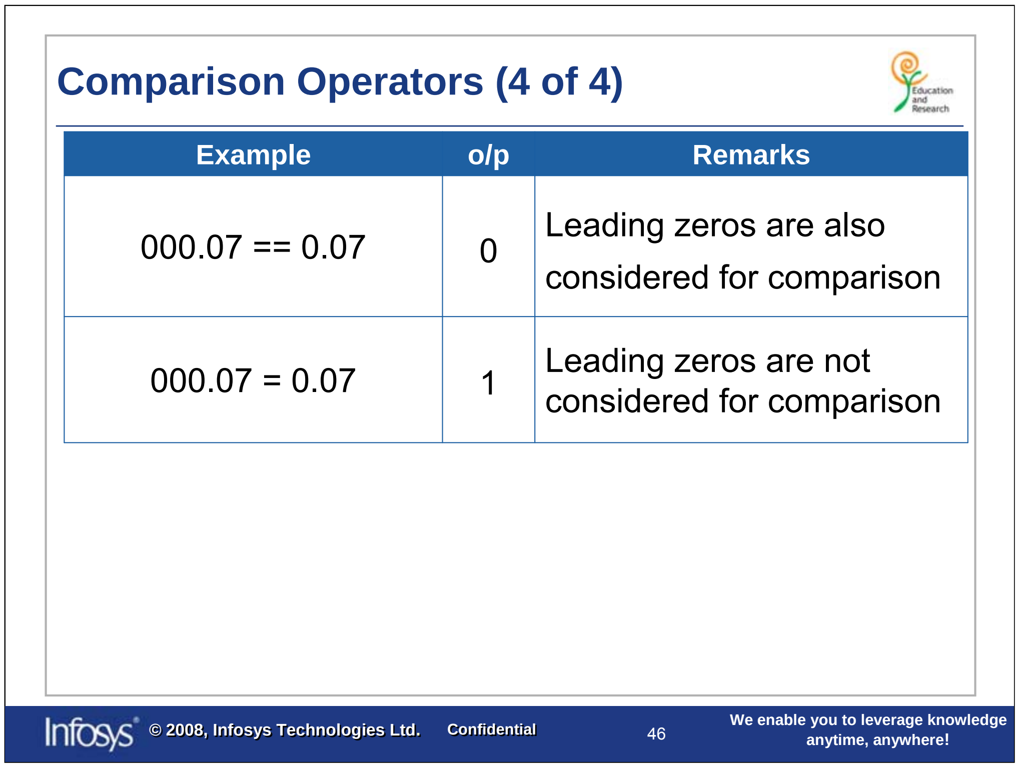
In both the examples, boolean value will be set to 0. This is because, character comparison is always case-sensitive.

‘ RAKESH ‘ = ‘RAKESH’

In this example, boolean value is set to 1. This is because, only case is considered for comparison & not spaces

‘ RAKESH ‘ == ‘RAKESH’

In this example, boolean value is set to 0. This is because, both case & spaces are considered for comparison



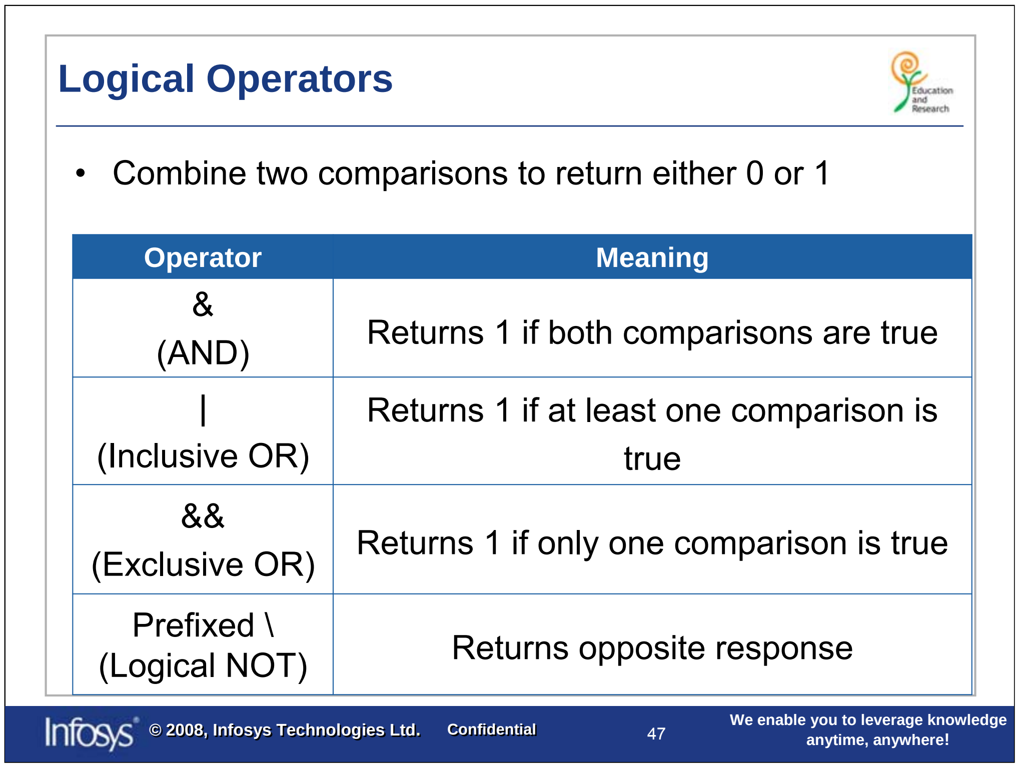
Comparison Operators (4 of 4)

Example	o/p	Remarks
000.07 == 0.07	0	Leading zeros are also considered for comparison
000.07 = 0.07	1	Leading zeros are not considered for comparison

Comparison Operators (4 of 4) - Notes

000.07 == 0.07
Since leading zeros are also considered for comparison, the boolean value will be set to 0.

000.07 =0.07
Since leading zeros are not considered for comparison, the boolean value will be set to 1.



Logical Operators

- Combine two comparisons to return either 0 or 1

Operator	Meaning
& (AND)	Returns 1 if both comparisons are true
| (Inclusive OR)	Returns 1 if at least one comparison is true
&& (Exclusive OR)	Returns 1 if only one comparison is true
Prefixed \ (Logical NOT)	Returns opposite response

Logical Operators - Notes

Logical expressions return 1 (true) or 0 (false) when processed. They are used to combine two comparisons and they return 1 or 0. 0 is returned when the comparison is FALSE. 1 is returned when the comparison is TRUE.

The logical operators and their meaning are listed here.

Lets take a look at some examples.

In the first example, both expressions (A=a) and (5<8) are true so the output is 1

In the second example, the expression (A==a) is false so the output is 0

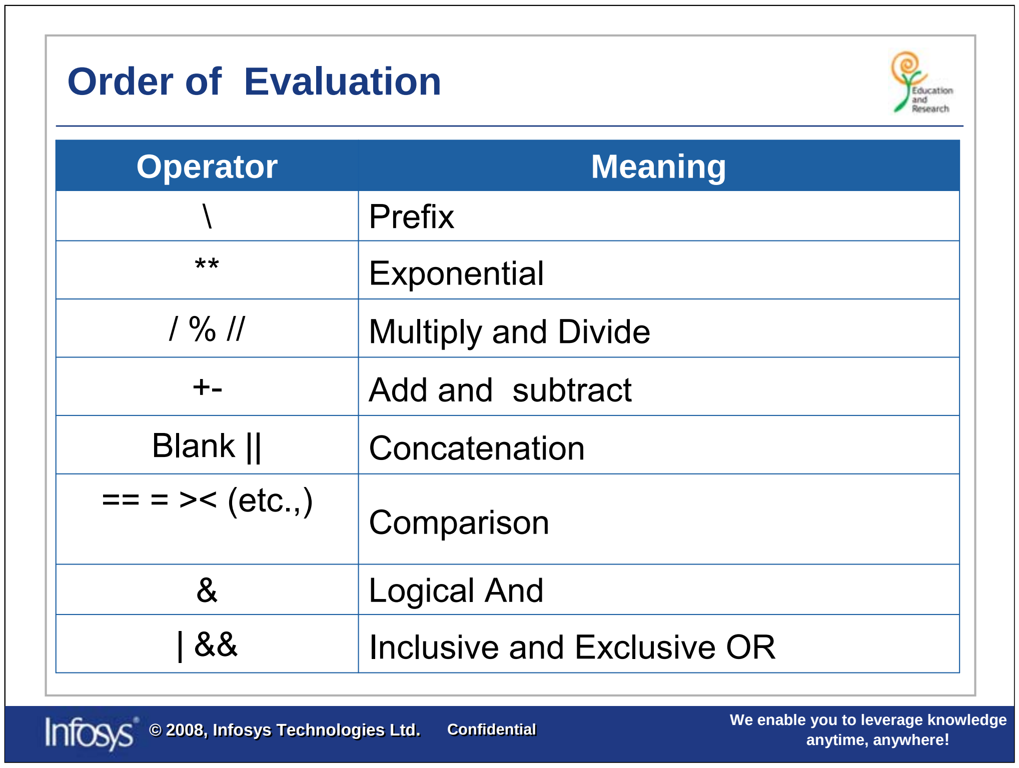
In the third example , (5 <8) returns 1 so the output is 1

In the fourth example, (“one”==“ONE”) and (a= b) both return 0 so the output is 0

In the fifth example , (“one”=“one”) returns 1 and (a=b) returns 0. Since at most one expression is true i.e. returns 1, so the output is 1.

In the sixth example (“one”=“one”) gives 1 so (“one”=“one”)|(45>54) returns 0 . So \((“one”=“one”)|(45>54)) will return 1.

In the seventh example , both (“one”=“one”) and (45<54) are true . So the output is 0. (The output would have been 1 only if one of the expressions had been true).

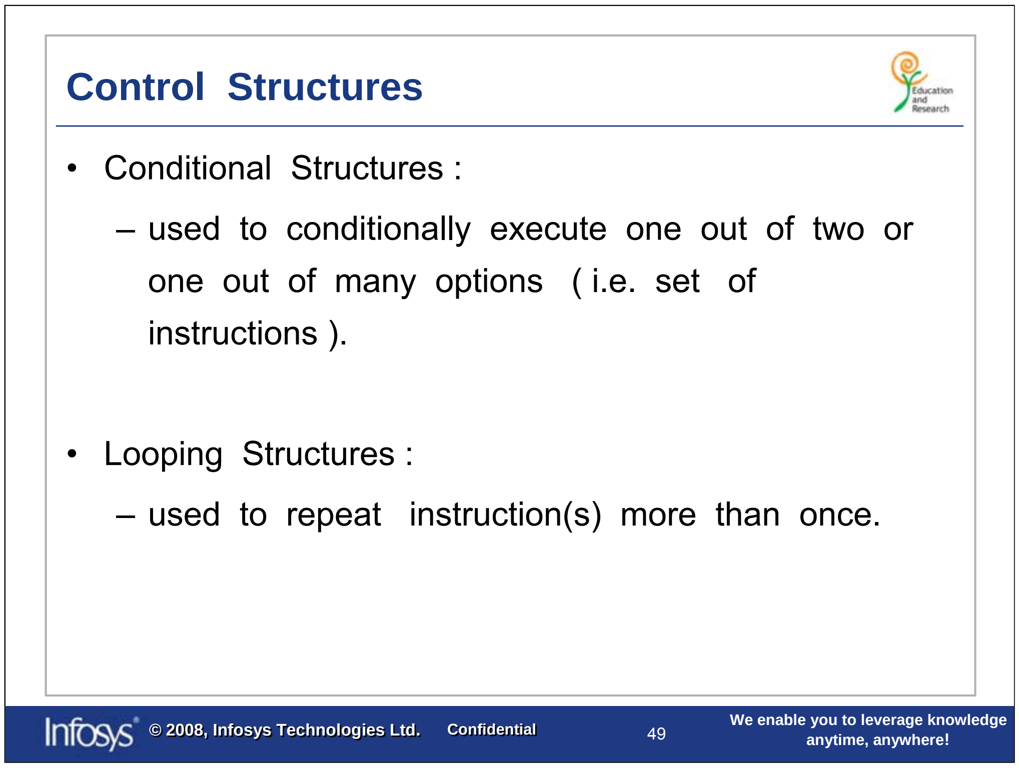


Order of Evaluation

Operator	Meaning
\	Prefix
**	Exponential
/ % //	Multiply and Divide
+-	Add and subtract
Blank ||	Concatenation
== = >< (etc.,)	Comparison
&	Logical And
| &&	Inclusive and Exclusive OR

Order of Evaluation - Notes

The order of evaluation is as shown in the table. The expression in the parentheses would be the first ones to be evaluated. Followed by all the arithmetic, comparison and logical operators.



Control Structures

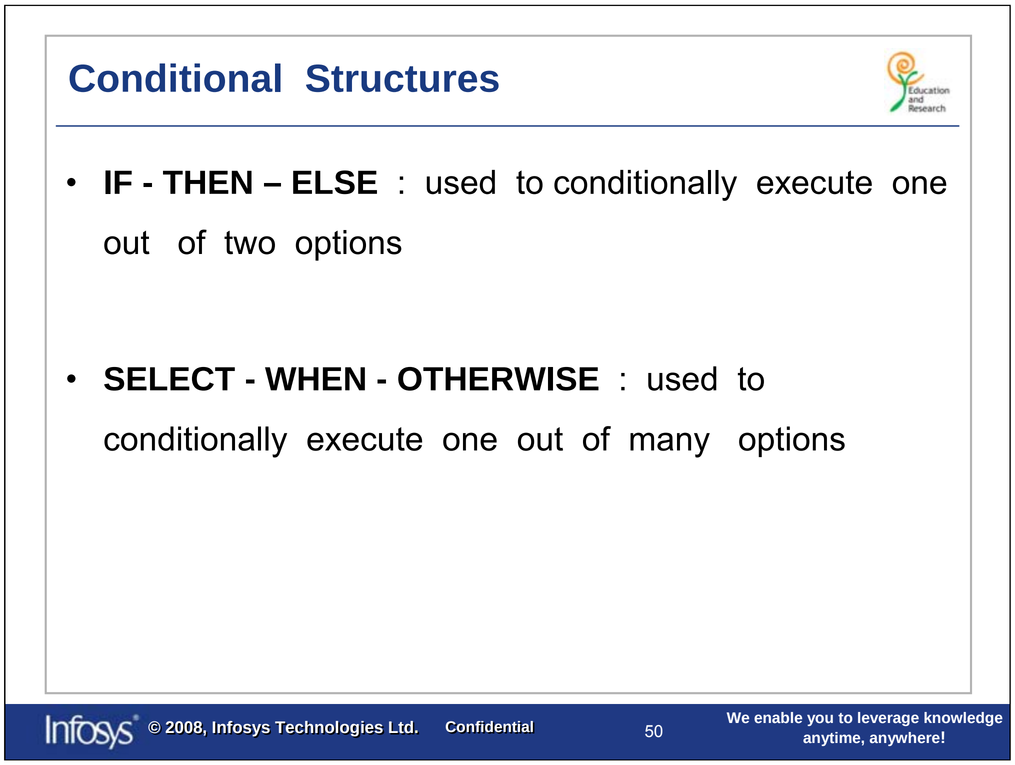
- Conditional Structures :
  - used to conditionally execute one out of two or one out of many options (i.e. set of instructions).
- Looping Structures :
  - used to repeat instruction(s) more than once.

Control Structures - Notes

We shall now proceed with the discussion on some control structures provided in REXX . There are mainly two categories of control structures --- Conditional Structures and Looping Structures .

Conditional structures are used to conditionally execute one out of two or one out of many instruction sets.

Looping structures are used to repeat instructions more than once.



Conditional Structures

- **IF - THEN – ELSE** : used to conditionally execute one out of two options

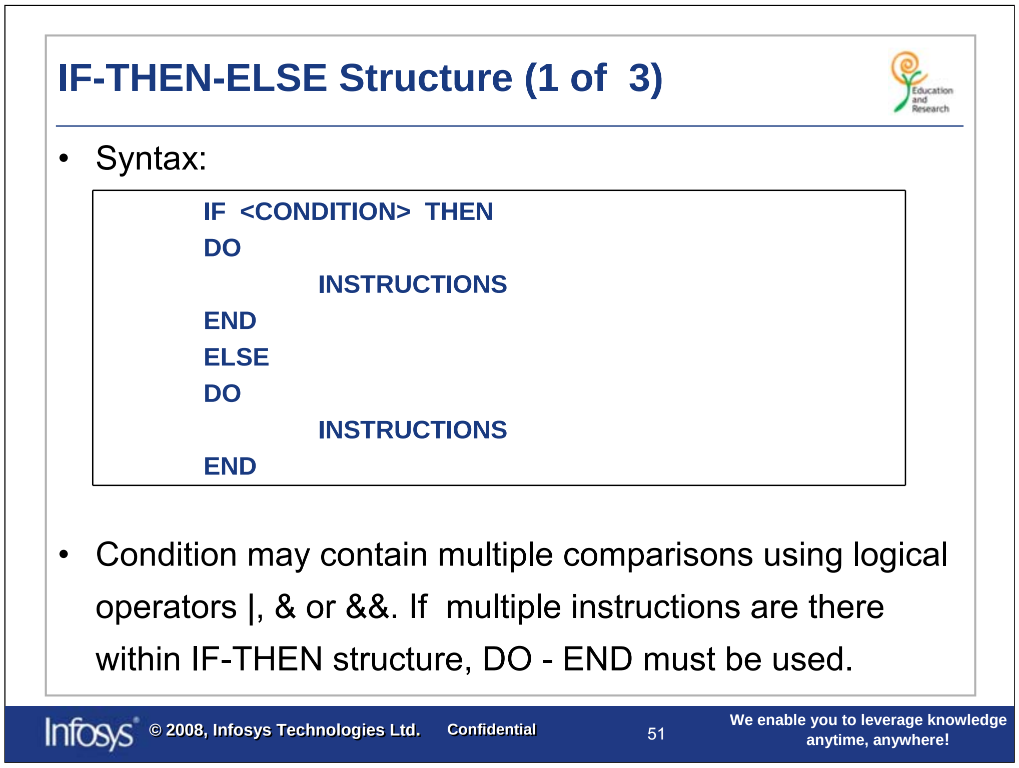
- **SELECT - WHEN - OTHERWISE** : used to conditionally execute one out of many options

Conditional Structures - Notes

REXX has two conditional structures. They are IF THEN ELSE and SELECT WHEN OTHERWISE.

IF-THEN-ELSE is used to conditionally execute one out of two options.

SELECT-WHEN-OTHERWISE is used to conditionally execute one out of many options.



IF-THEN-ELSE Structure (1 of 3)

- Syntax:

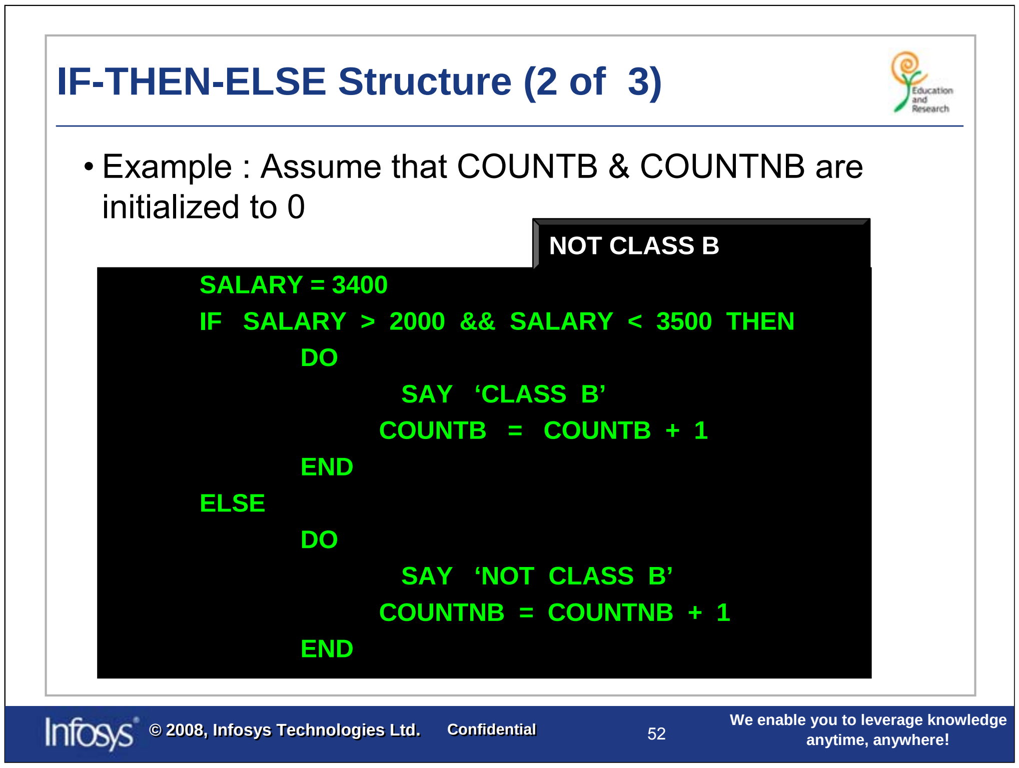
```
IF  <CONDITION>  THEN
DO
          INSTRUCTIONS
END
ELSE
DO
          INSTRUCTIONS
END
```

- Condition may contain multiple comparisons using logical operators |, & or &&. If multiple instructions are there within IF-THEN structure, DO - END must be used.

IF-THEN-ELSE Structure (1 of 3) - Notes

IF structure is used to conditionally execute one out of the two sets of instructions. If the expression mentioned in the IF statement evaluates to TRUE, then the set of statements which are a part of IF block are executed. If the expression evaluates to FALSE, then the set of statements or instructions which are a part of the ELSE block are executed. However, it is to be noted that the expression has to be evaluated to either 0 or 1.

It is to be kept in mind that if there are multiple statements following the THEN clause or ELSE clause, they must be enclosed within a pair of DO – END statements as shown in the syntax above. Also a condition (expression) can contain multiple comparisons using logical operators |,& or &&.



IF-THEN-ELSE Structure (2 of 3)

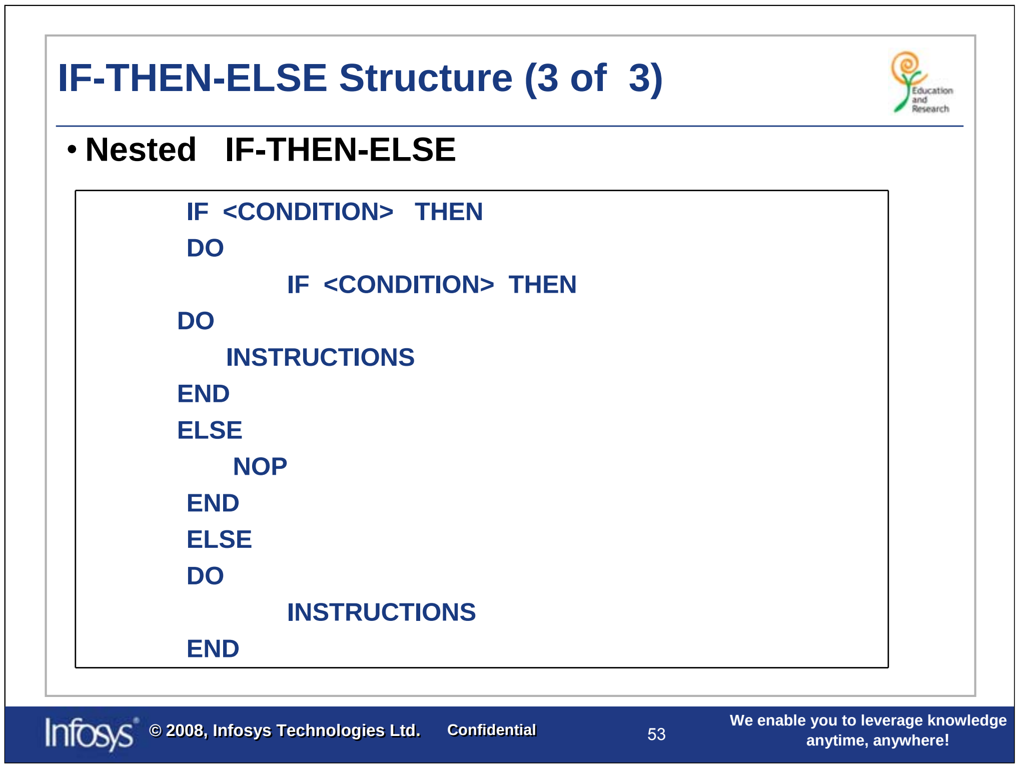
- Example : Assume that COUNTB & COUNTNB are initialized to 0

NOT CLASS B

```
SALARY = 3400
IF  SALARY  >  2000  &&  SALARY  <  3500  THEN
        DO
                SAY  'CLASS  B'
              COUNTB   =   COUNTB  +  1
        END
ELSE
        DO
                SAY  'NOT  CLASS  B'
              COUNTNB  =  COUNTNB  +  1
        END
```

IF-THEN-ELSE Structure (2 of 3) - Notes

In the above example, if the value of the variable SALARY is greater than 2000 and less than 3500 , NOT CLASS B will be displayed on the terminal and the value of a counter variable COUNTNB is incremented by 1 . Otherwise the message CLASS B is displayed on the terminal and a different counter variable COUNTB is incremented by 1 . Note that the statements in the IF branch and as well as the ELSE branch are enclosed with DO- END.



IF-THEN-ELSE Structure (3 of 3)

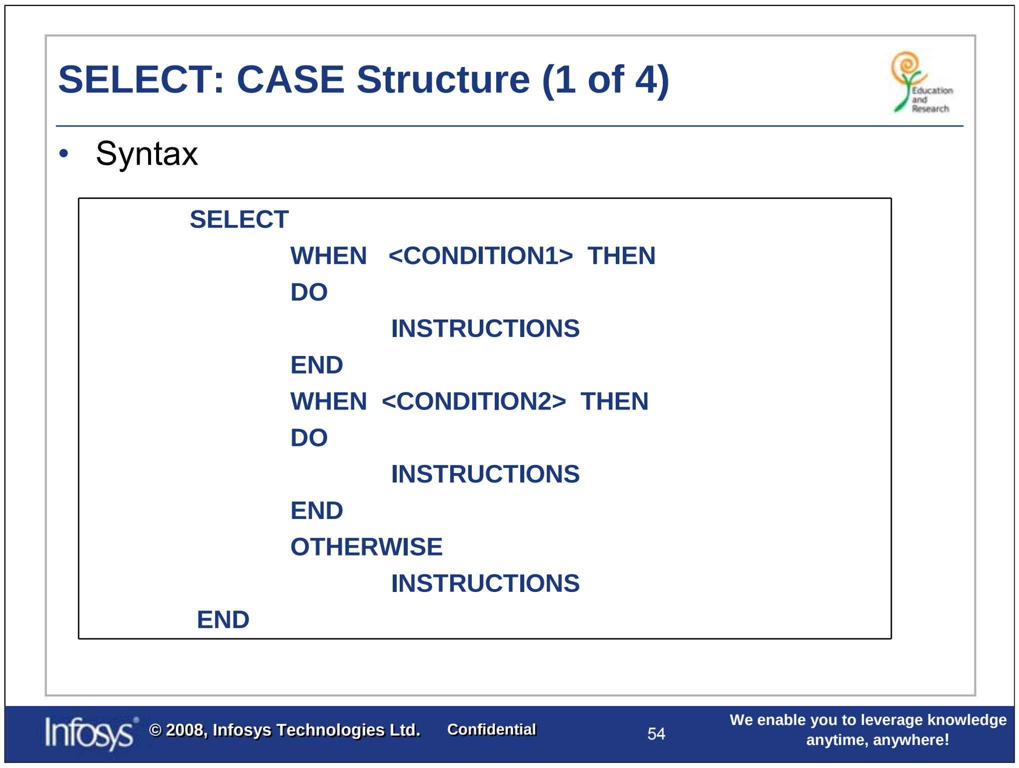
• Nested IF-THEN-ELSE

```
IF <CONDITION> THEN
DO
        IF <CONDITION> THEN
DO
    INSTRUCTIONS
END
ELSE
    NOP
END
ELSE
DO
        INSTRUCTIONS
END
```

Infosys © 2008, Infosys Technologies Ltd. Confidential 53 We enable you to leverage knowledge anytime, anywhere!

IF-THEN-ELSE Structure (3 of 3) - Notes

We can have one or more IF-THEN-ELSE structures within another IF-THEN-ELSE structure as shown in the slide . Such a structure is known as nested IF-THEN-ELSE . It is to be noted that if there are more than one statements in either branch (IF or ELSE) , they have to be enclosed within a pair of DO – END clauses . Also if there is no operation to be performed in either branch (IF or ELSE) , then we can write NOP in that branch instead as shown above in the slide .



SELECT: CASE Structure (1 of 4)

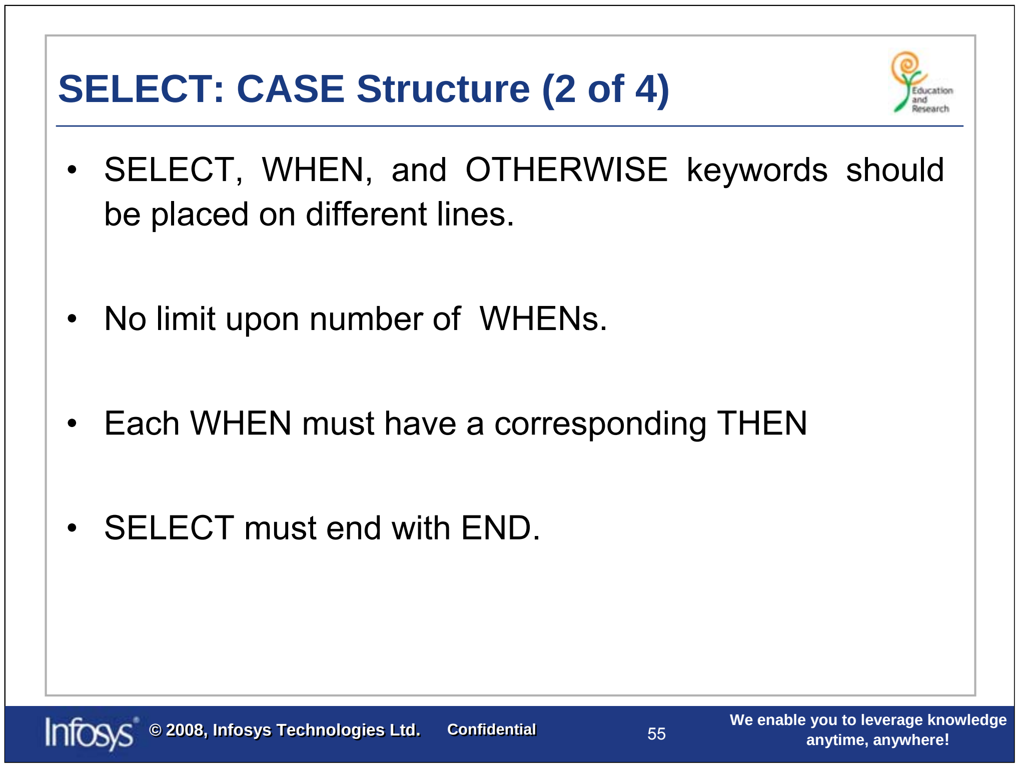
- Syntax

```
SELECT
        WHEN  <CONDITION1>  THEN
        DO
                INSTRUCTIONS
        END
        WHEN  <CONDITION2>  THEN
        DO
                INSTRUCTIONS
        END
        OTHERWISE
                INSTRUCTIONS
END
```

Infosys © 2008, Infosys Technologies Ltd. Confidential 54 We enable you to leverage knowledge anytime, anywhere!

SELECT : CASE Structure (1 of 4) - Notes

SELECT executes any one set out of many sets of instructions. Have a look at the SELECT statement given here. The condition specified after WHEN is evaluated and if it results in 1, the set of instructions following the associated WHEN are executed. The control is then passed to END. But, if the condition results in 0, the control is passed to the next WHEN clause. If none of the conditions specified after WHEN is satisfied, control is shifted to the instructions following the OTHERWISE clause .



SELECT: CASE Structure (2 of 4)

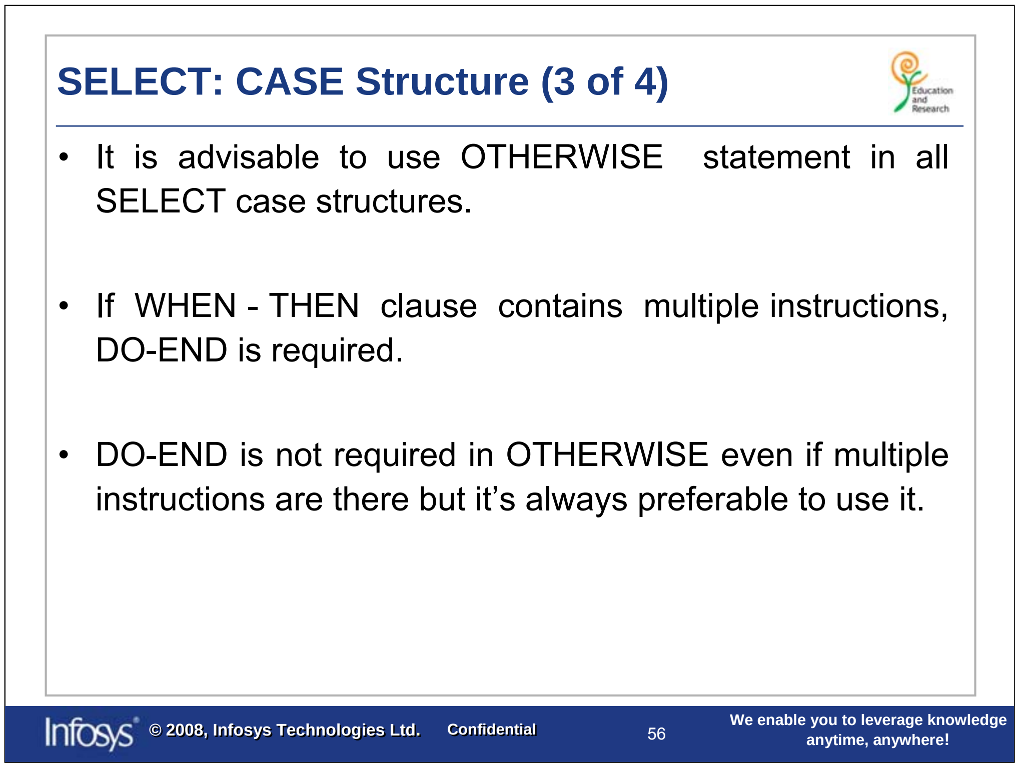
- SELECT, WHEN, and OTHERWISE keywords should be placed on different lines.
- No limit upon number of WHENs.
- Each WHEN must have a corresponding THEN
- SELECT must end with END.

© 2008, Infosys Technologies Ltd. Confidential 55 We enable you to leverage knowledge anytime, anywhere!

SELECT : CASE Structure (2 of 4) - Notes

These points must always be kept in mind when using SELECT: CASE structure to avoid syntax errors.

1. SELECT, WHEN, and OTHERWISE should be placed on different lines
2. No limit upon number of WHENs
3. Each WHEN must have a corresponding THEN associated with it.
4. SELECT must end with END


SELECT: CASE Structure (3 of 4)

- It is advisable to use OTHERWISE statement in all SELECT case structures.
- If WHEN - THEN clause contains multiple instructions, DO-END is required.
- DO-END is not required in OTHERWISE even if multiple instructions are there but it's always preferable to use it.

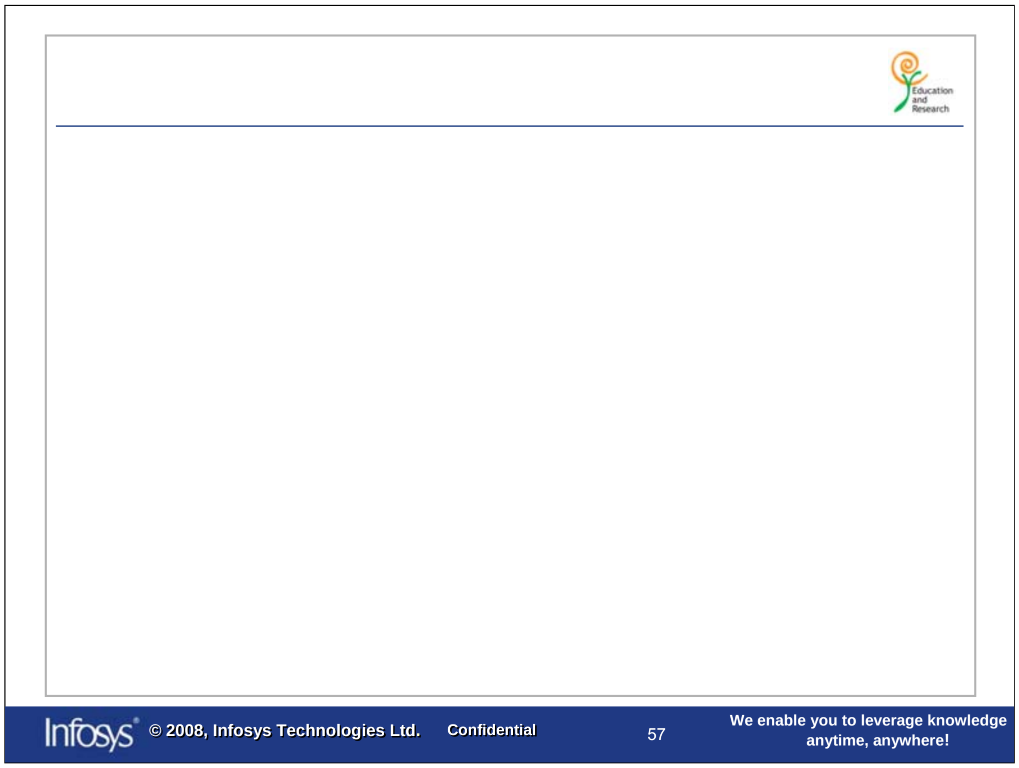
SELECT : CASE Structure (3 of 4) - Notes

1. OTHERWISE is optional so it may not contain any useful instructions. If no instructions are to be executed within OTHERWISE, then NOP is used to indicate the same. It is advisable to use OTHERWISE statement in all SELECT case structures.

Consider the following example:

```
/* REXX */
SALARY = 2500
SELECT
  WHEN SALARY > 2000 THEN
    SAY 'Condition satisfied'
END
```

In the above example, OTHERWISE is not specified. But even then it works fine & displays **'Condition satisfied'.** This is because, once the condition in the WHEN statement is satisfied & the instructions corresponding to that WHEN are executed, control will be passed out of the SELECT structure.



SELECT : CASE Structure (3 of 4) - Notes (Contd.)

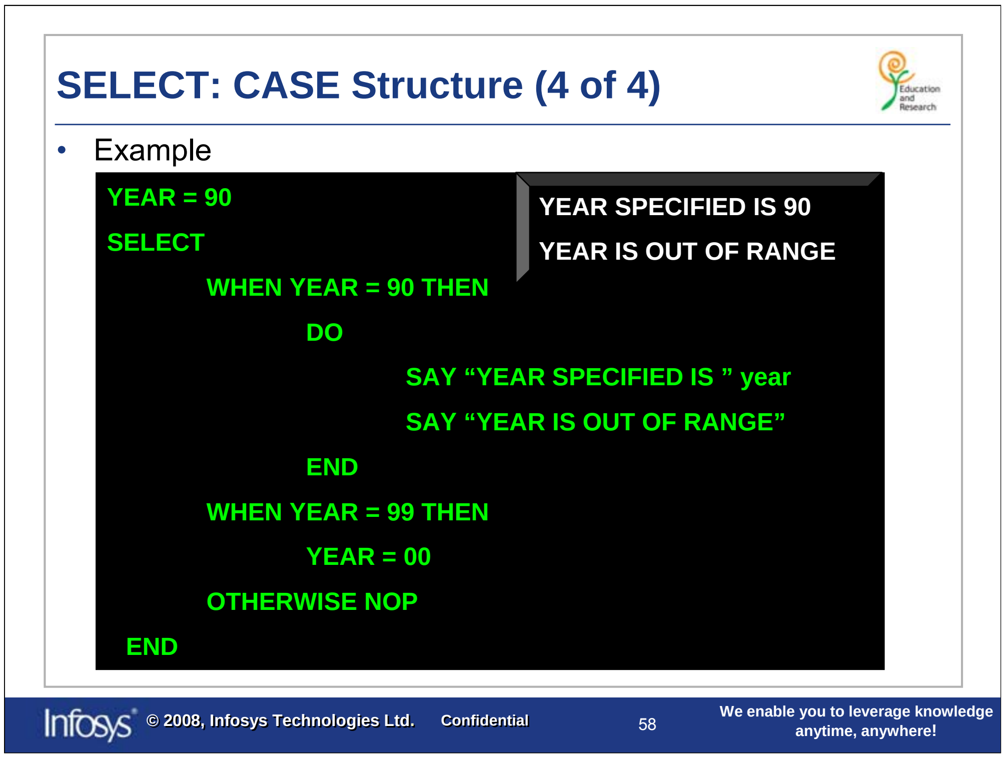
Consider the same example without OTHERWISE:

```
/* REXX */
SALARY = 1400
SELECT
  WHEN SALARY > 2000 THEN
    SAY SALARY
END
```

In this case, the condition is not satisfied. Hence, the processor looks out for OTHERWISE clause & since OTHERWISE is not specified, an error will occur.

If WHEN - THEN clause contains multiple instructions, DO-END is required

DO-END is not required in OTHERWISE even if multiple instructions are there but it’s always preferable to use it.



SELECT: CASE Structure (4 of 4)

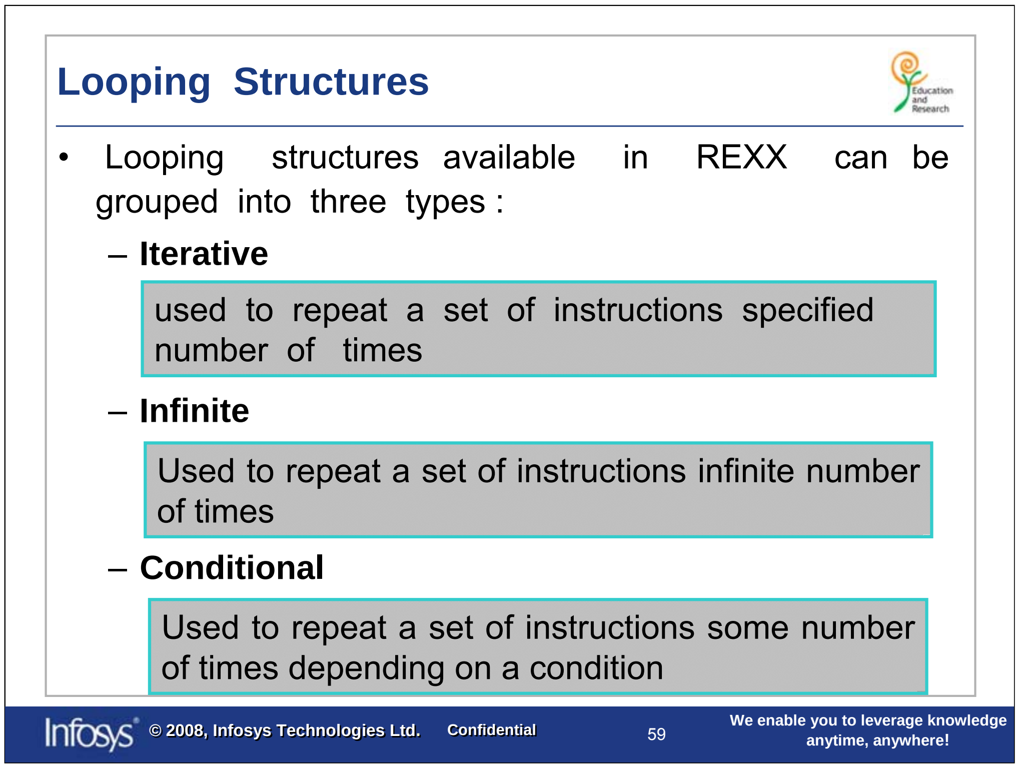
- Example

```
YEAR = 90
SELECT
          WHEN YEAR = 90 THEN
                    DO
                              SAY "YEAR SPECIFIED IS " year
                              SAY "YEAR IS OUT OF RANGE"
                    END
          WHEN YEAR = 99 THEN
                    YEAR = 00
          OTHERWISE NOP
  END
```

```
YEAR SPECIFIED IS 90
YEAR IS OUT OF RANGE
```

SELECT : CASE Structure (4 of 4) - Notes

Here is an example for the SELECT CASE structure. Based on the value of the variable YEAR, two different sets of instructions can be processed . If the value of YEAR is 90 then "**YEAR SPECIFIED IS 90"** and "**YEAR IS OUT OF RANGE**" is printed . The control then goes to the statement following the END statement of the SELECT structure. If instead, the value of YEAR is 99, then the value of the variable YEAR is reset to 00 . If the value of YEAR is neither 90 nor 99, then nothing is done (as there are no instructions to be executed in the OTHERWISE clause).



Looping Structures

- Looping structures available in REXX can be grouped into three types :
  - **Iterative**

    used to repeat a set of instructions specified number of times
  - **Infinite**

    Used to repeat a set of instructions infinite number of times
  - **Conditional**

    Used to repeat a set of instructions some number of times depending on a condition

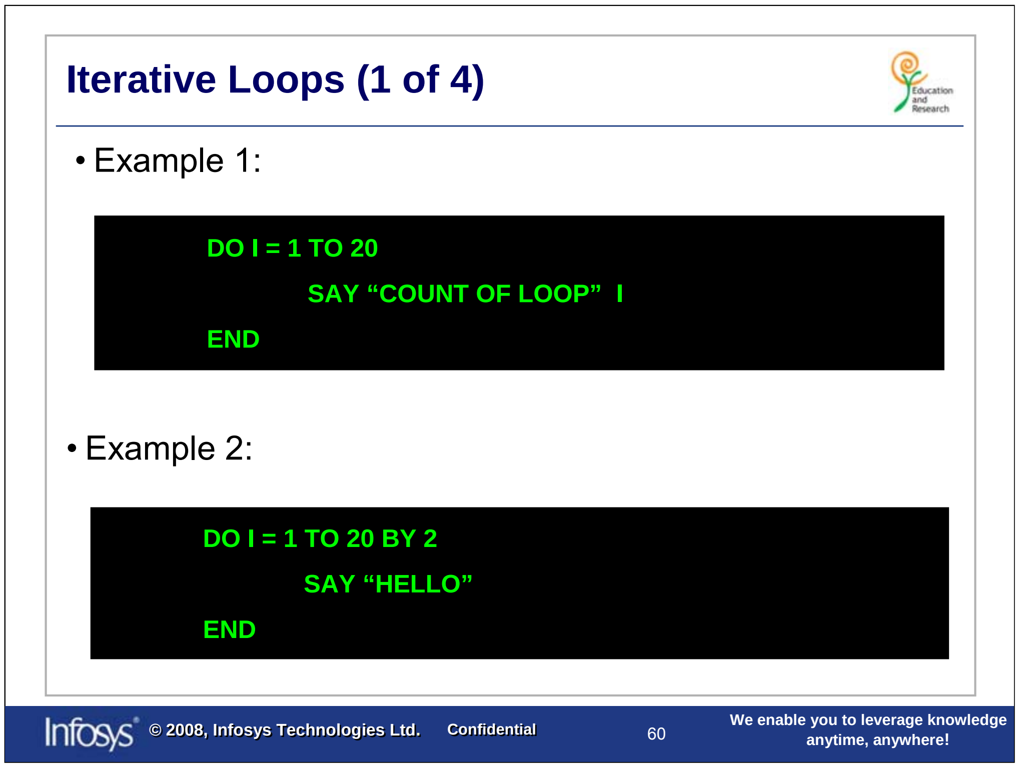
Looping Structures - Notes

We now discuss about the looping structures available in REXX . Looping structures are used to repeat a set of instructions. There are three types of looping structures :

Iterative : These looping structures can be used to repeat a set of instructions a specified number of times

Infinite : REXX has a looping structure that can be used to set up an infinite loop (i.e. A loop that never terminates)

Conditional : These looping structures can be used to repeat a set of instructions some number of times depending on a condition (or expression)



Iterative Loops (1 of 4)

- Example 1:

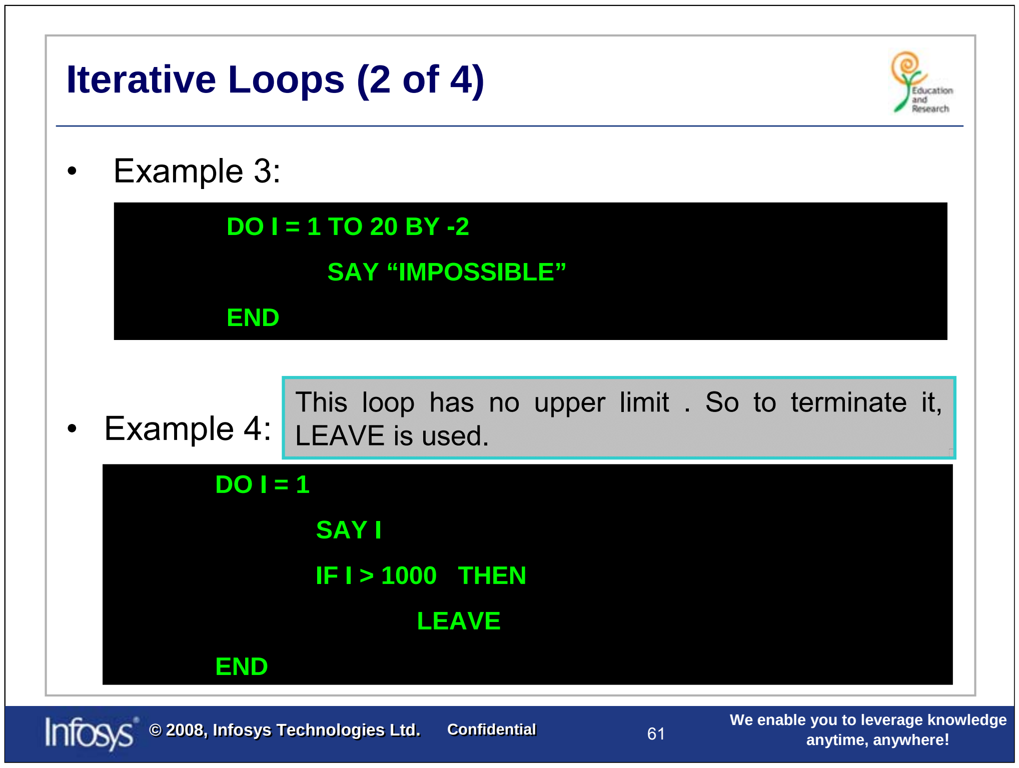
```
DO I = 1 TO 20
         SAY “COUNT OF LOOP”  I
END
```

- Example 2:

```
DO I = 1 TO 20 BY 2
         SAY “HELLO”
END
```

Iterative Loops (1 of 4) - Notes

The iterative loops are executed for a specified number of times. Let us understand this loop with examples. In example 1, I is initialized to 1. The loop is executed 20 times incrementing the value of I by one during each execution of the loop. In example 2, I is initialized to 1. The loop is executed 10 times incrementing the value of I by two during each execution of the loop.



Iterative Loops (2 of 4)

- Example 3:

```
DO I = 1 TO 20 BY -2
        SAY “IMPOSSIBLE”
END
```

- Example 4: This loop has no upper limit . So to terminate it, LEAVE is used.

```
DO I = 1
        SAY I
        IF I > 1000   THEN
                LEAVE
END
```

Infosys © 2008, Infosys Technologies Ltd. Confidential 61 We enable you to leverage knowledge anytime, anywhere!

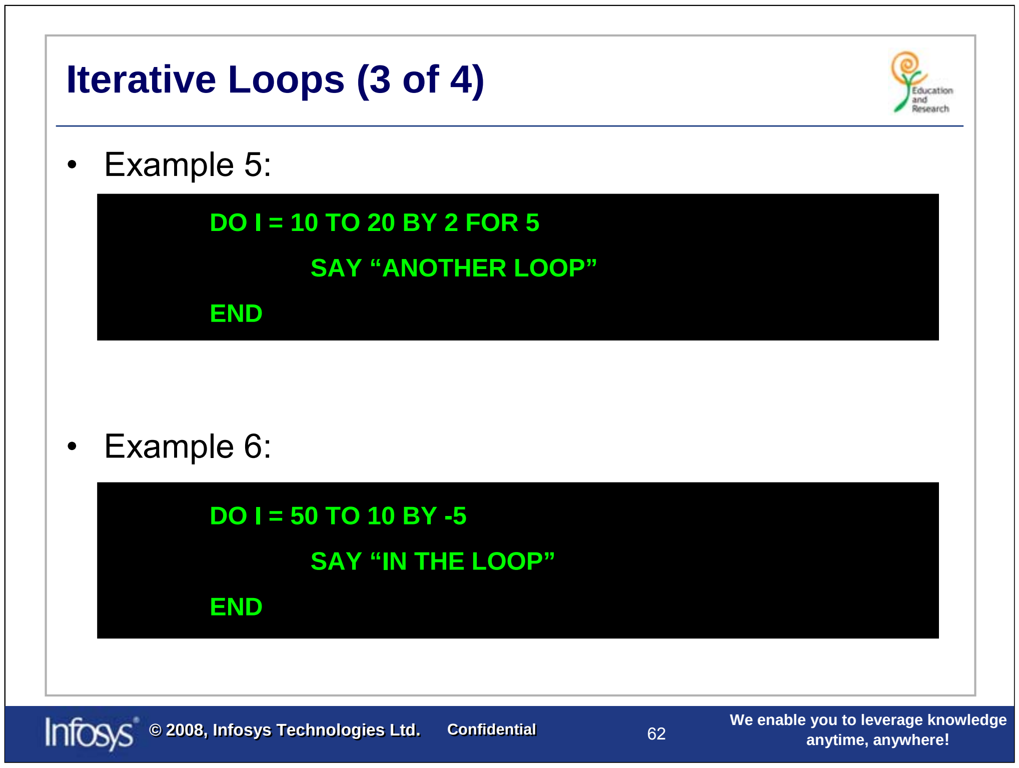
Iterative Loops (2 of 4) - Notes

Since Example 3 is an impossible condition, REXX will ignore it. This is because it is not possible to reach the value of 20 from 1 by decrementing it by 2 at each step . As a result , the loop will not get executed even once.

Example4:

The loop in example 4 has no upper limit . So to terminate it, LEAVE is used. We shall revisit the LEAVE instruction later.

In the above example, numbers from 1 to 1001 will be displayed & at 1001 the loop terminates as the IF condition is satisfied.



Iterative Loops (3 of 4)

- Example 5:

```
DO I = 10 TO 20 BY 2 FOR 5
        SAY “ANOTHER LOOP”
END
```

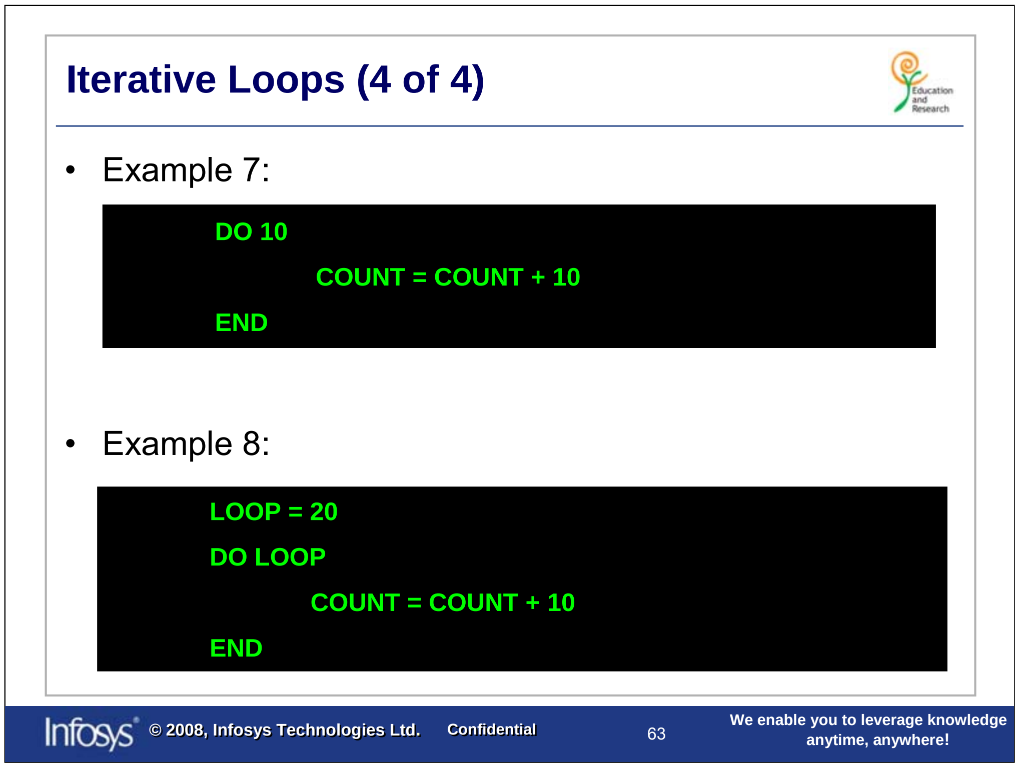
- Example 6:

```
DO I = 50 TO 10 BY -5
        SAY “IN THE LOOP”
END
```

Infosys © 2008, Infosys Technologies Ltd. Confidential 62 We enable you to leverage knowledge anytime, anywhere!

Iterative Loops (3 of 4) - Notes

In case of Example: 5, The set of instructions will be executed until either the upper limit 20 is reached or for 5 (as specified by FOR) times, whichever condition is met earlier. In example 6, the loop counter I is initialized to 50 . The counter value for this loop gets decremented by 5 in each iteration of the loop starting from 50 till lower limit 10 is reached .



Iterative Loops (4 of 4)

- Example 7:

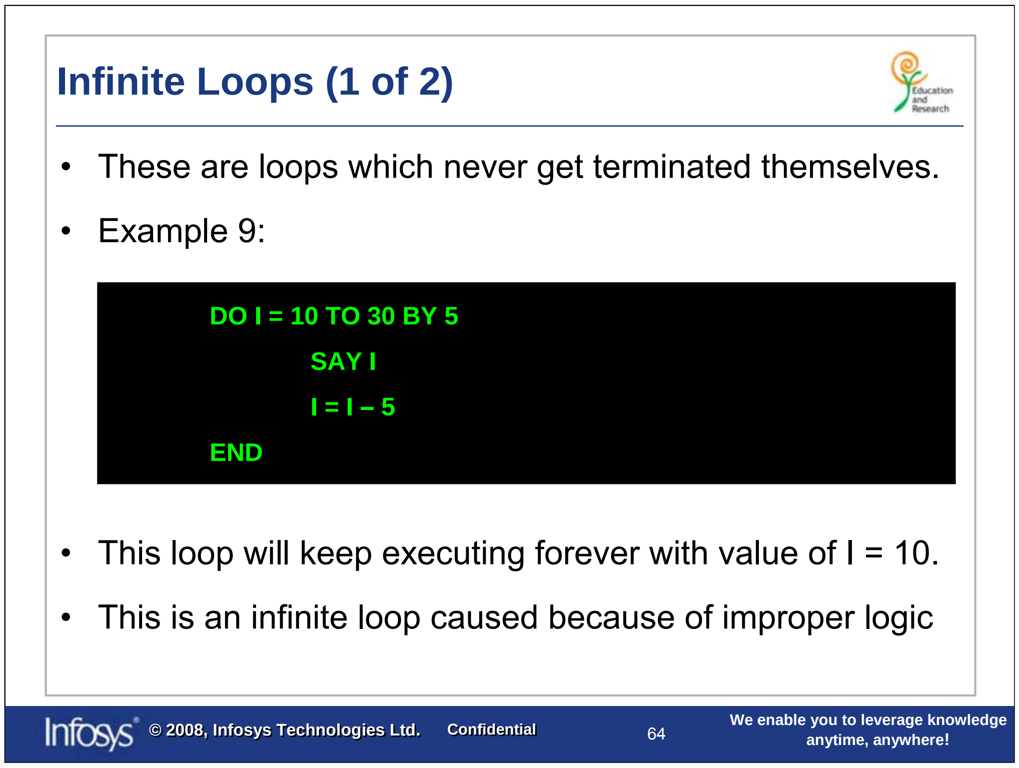
```
        DO 10
                COUNT = COUNT + 10
        END
```

- Example 8:

```
        LOOP = 20
        DO LOOP
                COUNT = COUNT + 10
        END
```

Iterative Loops (4 of 4) - Notes

In example 7 , the loop will get executed 10 times (as specified after DO) . In each iteration , the value of the variable COUNT will be incremented by 10 . In case of example 8, the value of the variable LOOP (here it is 20) will get substituted. So the loop will get executed 20 times.



Infinite Loops (1 of 2)

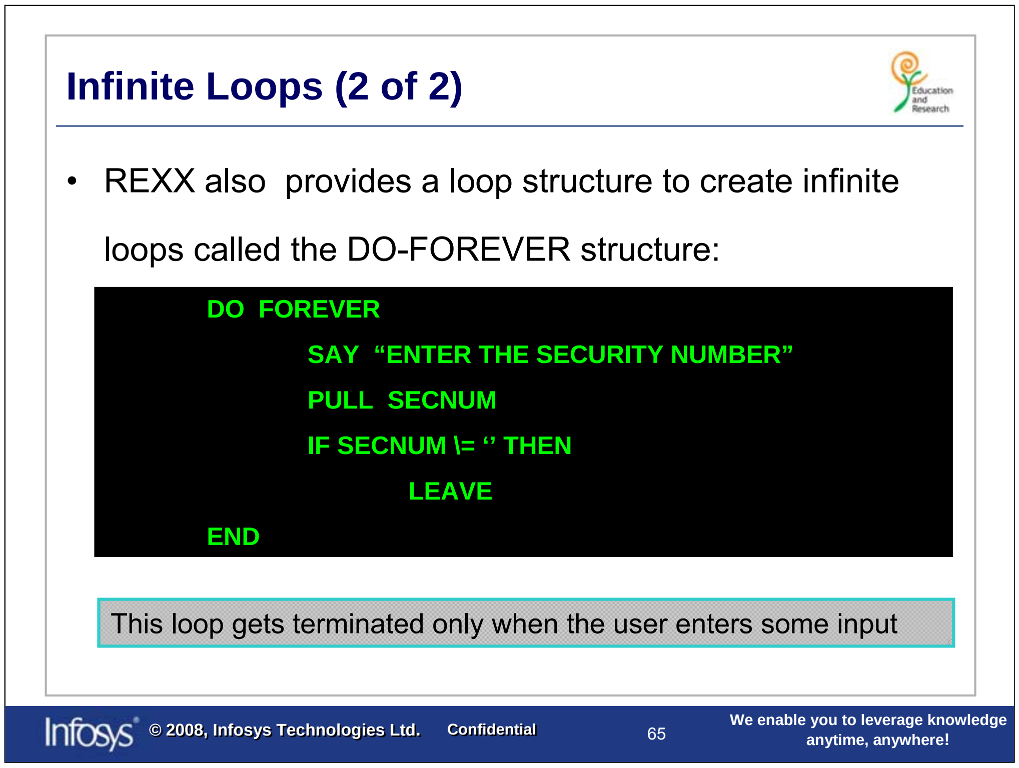
- These are loops which never get terminated themselves.
- Example 9:

```
        DO I = 10 TO 30 BY 5
                SAY I
                I = I – 5
        END
```

- This loop will keep executing forever with value of I = 10.
- This is an infinite loop caused because of improper logic

Infinite Loops (1 of 2) - Notes

We can create infinite loops as shown in example 9 here. In the first iteration, the value of I is 10. This value is then printed on the terminal (by virtue of the SAY statement in the loop). The next instruction I = I – 5 decrements the value of I by 5 and so the value of I now becomes 5 (from 10). In the next iteration, the value of I is incremented by 5 and becomes 10. This value is printed on the terminal . The next statement again reduces the value of I from 10 to 5 . In the next iteration , the value of I is incremented by 5 and becomes 10 again . This cycle continues and so the value of I never reaches the upper limit 30. Thus the loop will keep executing forever with value of I = 10.



Infinite Loops (2 of 2)

- REXX also provides a loop structure to create infinite loops called the DO-FOREVER structure:

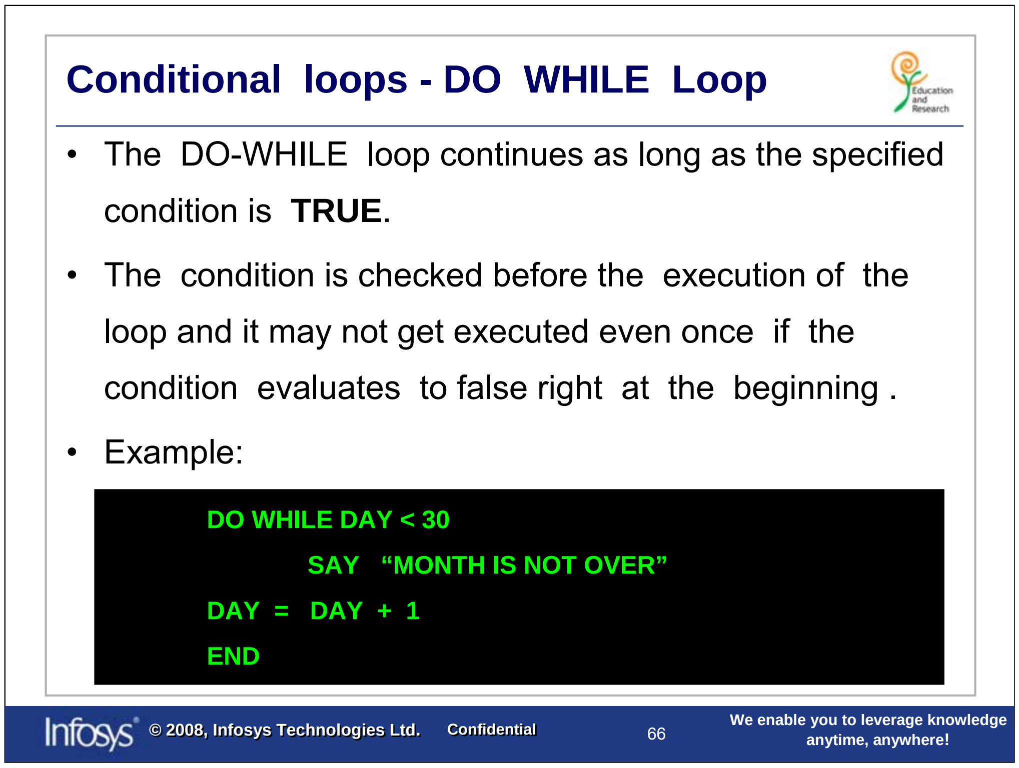
```
DO  FOREVER
        SAY  “ENTER THE SECURITY NUMBER”
        PULL  SECNUM
        IF SECNUM \= ‘’ THEN
                LEAVE
END
```

This loop gets terminated only when the user enters some input

Infosys © 2008, Infosys Technologies Ltd. Confidential 65 We enable you to leverage knowledge anytime, anywhere!

Infinite Loops (2 of 2) - Notes

Instead of using a numeric expression to create an infinite loop as shown in the previous slide, we can place the keyword FOREVER after DO. Then the loop will continue to repeat indefinitely, unless the user manually aborts the script, or the ITERATE or LEAVE keywords are used within the loop . We shall revisit the ITERATE and LEAVE keywords later in this course.



Conditional loops - DO WHILE Loop

- The DO-WHILE loop continues as long as the specified condition is **TRUE**.
- The condition is checked before the execution of the loop and it may not get executed even once if the condition evaluates to false right at the beginning .
- Example:

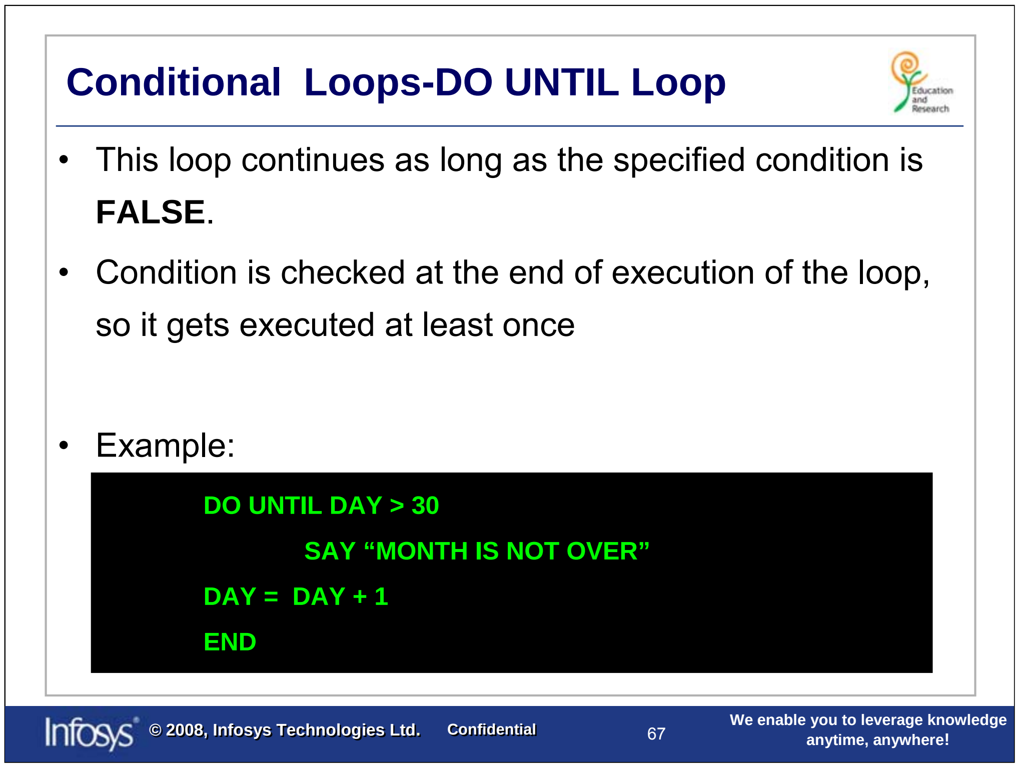
```
DO WHILE DAY < 30
        SAY   “MONTH IS NOT OVER”
DAY  =  DAY  +  1
END
```

© 2008, Infosys Technologies Ltd. Confidential 66 We enable you to leverage knowledge anytime, anywhere!

Conditional loops – DO WHILE loop

We now discuss conditional loops that execute a set of instructions some number of times depending on the value of an expression (condition) . In case of DO WHILE , the condition is checked before the execution of the loop and it continues to execute only if the condition is true. Note that there maybe cases where the loop does not get executed even once.

In the above example, the loop will continue to print the line “MONTH IS NOT OVER” as long as the value of the variable DAY is less than 30. Note that within the body of the loop, the value of the variable DAY is incremented by 1 in each iteration of the loop.



Conditional Loops-DO UNTIL Loop

- This loop continues as long as the specified condition is **FALSE**.
- Condition is checked at the end of execution of the loop, so it gets executed at least once
- Example:

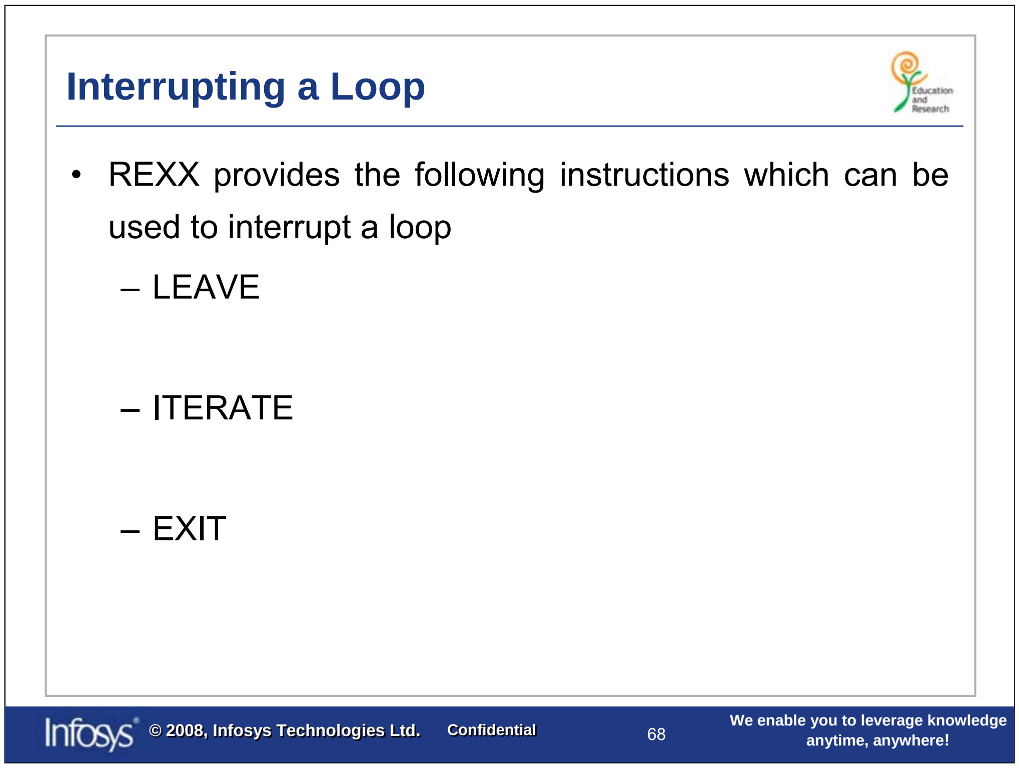
```
          DO UNTIL DAY > 30
                    SAY “MONTH IS NOT OVER”
          DAY =  DAY + 1
          END
```

Conditional loops – DO UNTIL loop

In the DO UNTIL loop, the condition is checked at the end of execution of the loop. So the loop gets executed at least once . The execution of the loop stops as soon as the condition specified becomes true.

In the above example, the loop will continue to print the line “MONTH IS NOT OVER” as long as the value of the variable DAY is not greater than 30. i.e. The loop will terminate as soon as the value of the variable DAY exceeds 30.

Please note that the line “MONTH IS NOT OVER” gets printed at least once since the condition (DAY > 30) is checked at the end of execution of the loop.

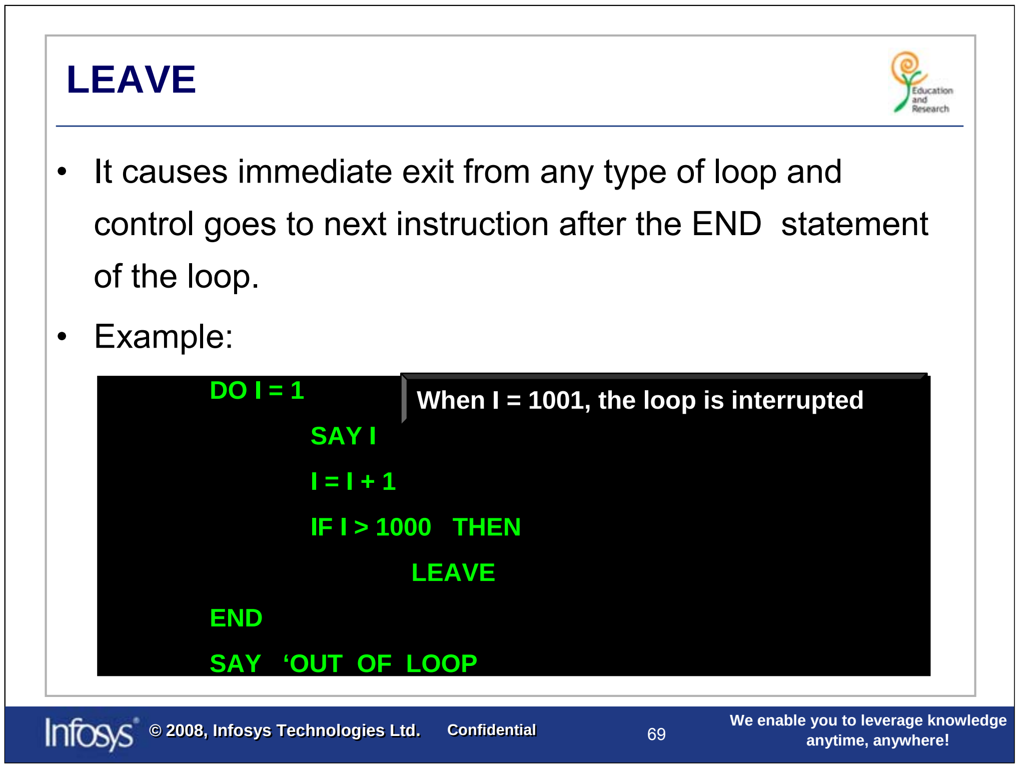


Interrupting a Loop

- REXX provides the following instructions which can be used to interrupt a loop
  - LEAVE
  - ITERATE
  - EXIT

Skipping the loop - Notes

REXX provides three instructions to interrupt a loop. They are EXIT, LEAVE & ITERATE.



LEAVE

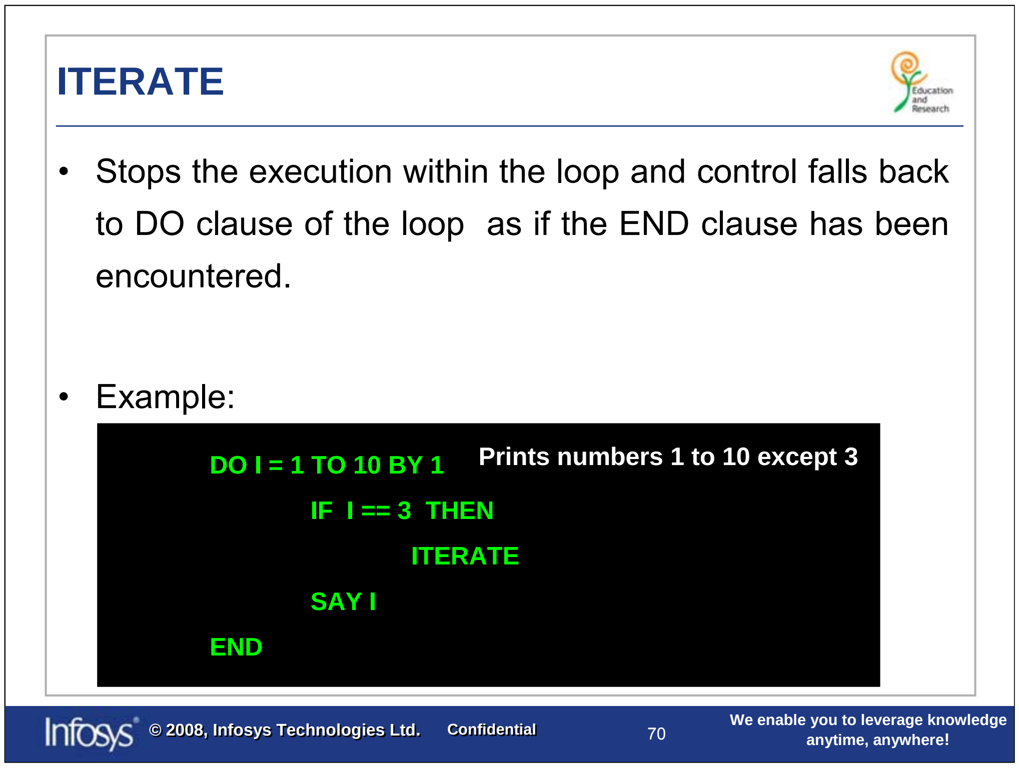
- It causes immediate exit from any type of loop and control goes to next instruction after the END statement of the loop.
- Example:

```
        DO I = 1
                SAY I
                I = I + 1
                IF I > 1000   THEN
                        LEAVE
        END
        SAY   'OUT  OF  LOOP
```

When I = 1001, the loop is interrupted

LEAVE - Notes

Here, the value of I is initialized to 1. In each iteration, the value of I is printed out and then the value of I is incremented by 1 . When the value of I becomes greater than 1000 , the LEAVE instruction causes the control to jump out of the loop . Control then passes to the statement following the END viz. OUT OF LOOP message is printed on the terminal .



ITERATE

- Stops the execution within the loop and control falls back to DO clause of the loop as if the END clause has been encountered.

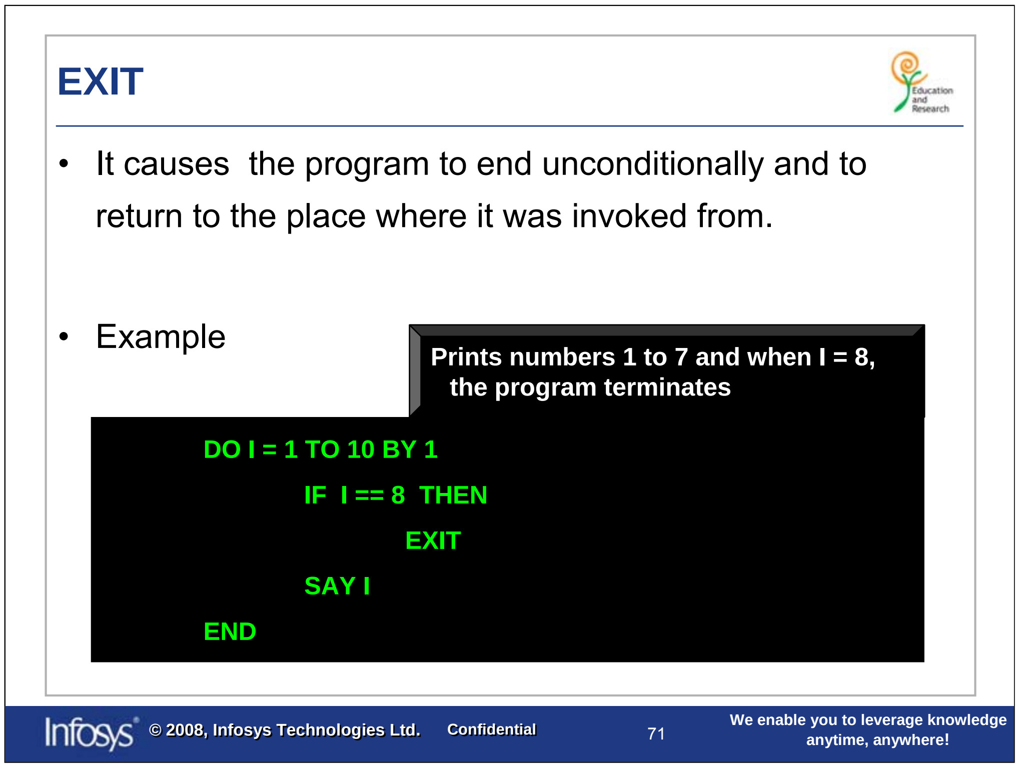
- Example:

```
DO I = 1 TO 10 BY 1        Prints numbers 1 to 10 except 3
        IF  I == 3  THEN
                ITERATE
        SAY I
END
```

ITERATE – Notes

Stops the execution within the loop and control falls back to DO clause of the loop as if the END clause has been encountered. The control variable (if any) is then iterated as normal .

The above code prints out values from 1 to 10 except for 3 . When the value of I becomes equal to 3, then the ITERATE instruction causes the control to fall back to the DO clause of the loop and the next iteration starts and the control variable I is incremented as usual by 1 .



EXIT

- It causes the program to end unconditionally and to return to the place where it was invoked from.

- Example

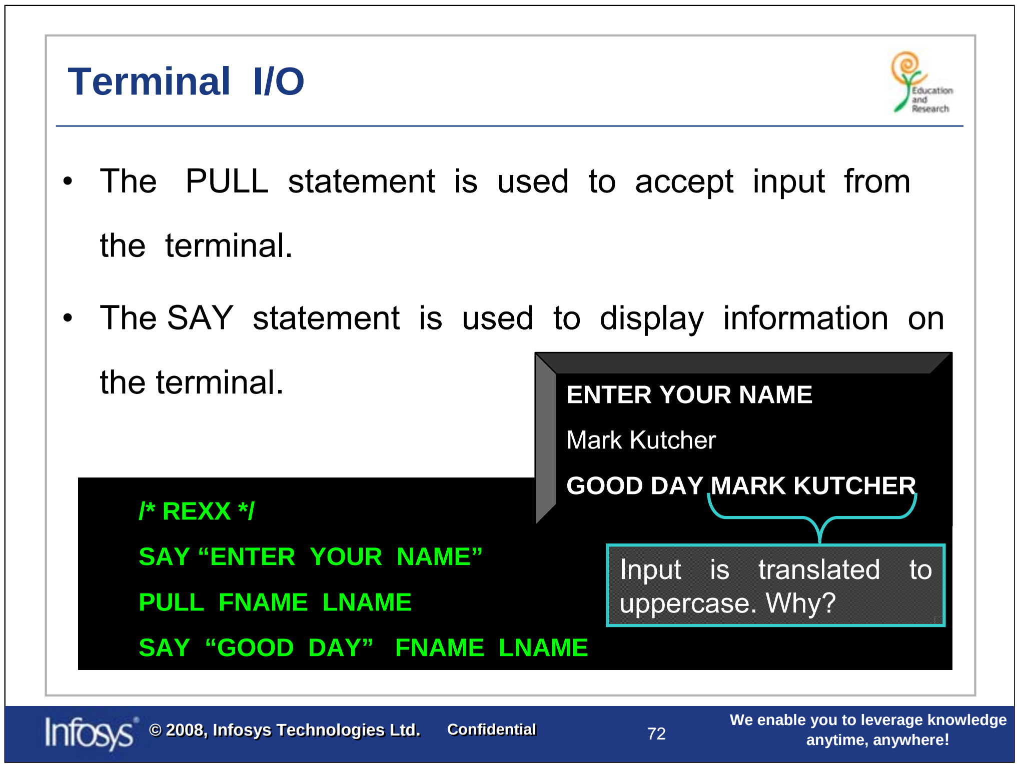
Prints numbers 1 to 7 and when I = 8, the program terminates

```
DO I = 1 TO 10 BY 1
        IF  I == 8  THEN
                EXIT
        SAY I
END
```

© 2008, Infosys Technologies Ltd.   Confidential   71   We enable you to leverage knowledge anytime, anywhere!

EXIT - Notes

In the given example , the values 1 to 7 are printed out on the terminal . When the value of I becomes equal to 8 , then the program terminates unconditionally .



Terminal I/O

- The PULL statement is used to accept input from the terminal.
- The SAY statement is used to display information on the terminal.

```
/* REXX */
SAY "ENTER YOUR NAME"
PULL FNAME LNAME
SAY "GOOD DAY" FNAME LNAME
```

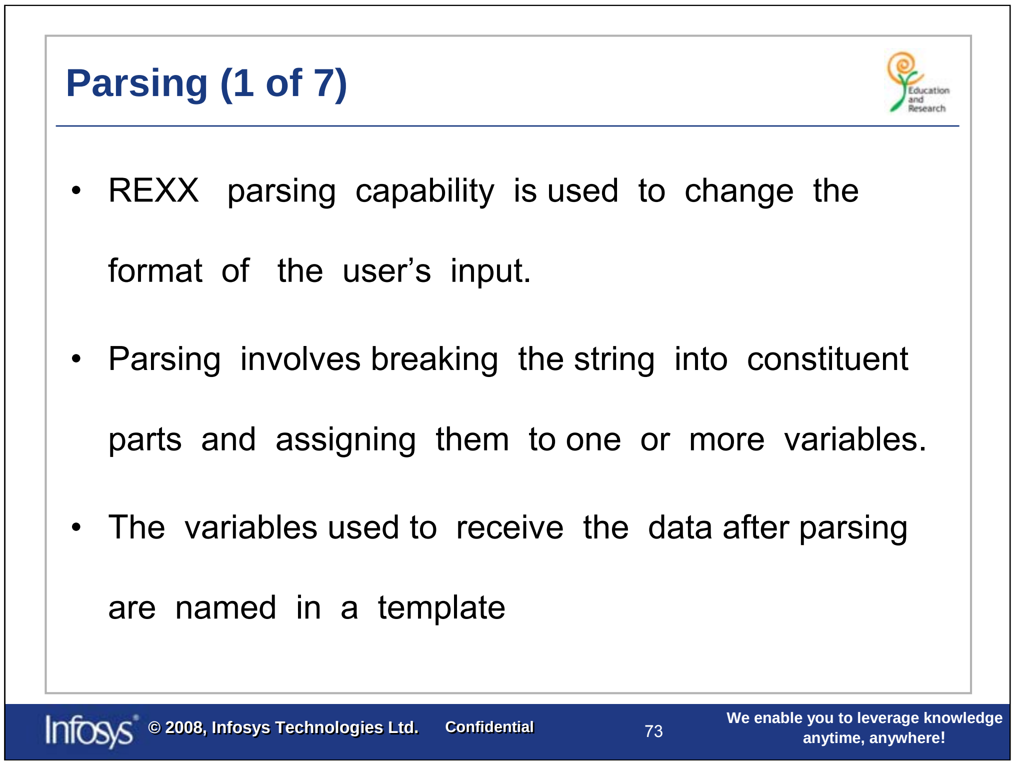
```
ENTER YOUR NAME
Mark Kutcher
GOOD DAY MARK KUTCHER
```

Input is translated to uppercase. Why?

Terminal I/O - Notes

The REXX instruction used to accept input from the user is PULL. The PULL statement automatically translates all input to UPPER CASE . For example, suppose 'Mark' is the input value given for the variable name FNAME and 'Kutcher' is the input value given for the variable name 'LNAME, then the output of the above program would be GOOD DAY MARK KUTCHER. We will see why this translation to UPPER CASE is done when we talk about parsing a little later. Also note that the PULL instruction can be used to accept more than one data item from the terminal .

The REXX instruction used to display on the terminal is SAY .



Parsing (1 of 7)

- REXX parsing capability is used to change the format of the user's input.
- Parsing involves breaking the string into constituent parts and assigning them to one or more variables.
- The variables used to receive the data after parsing are named in a template

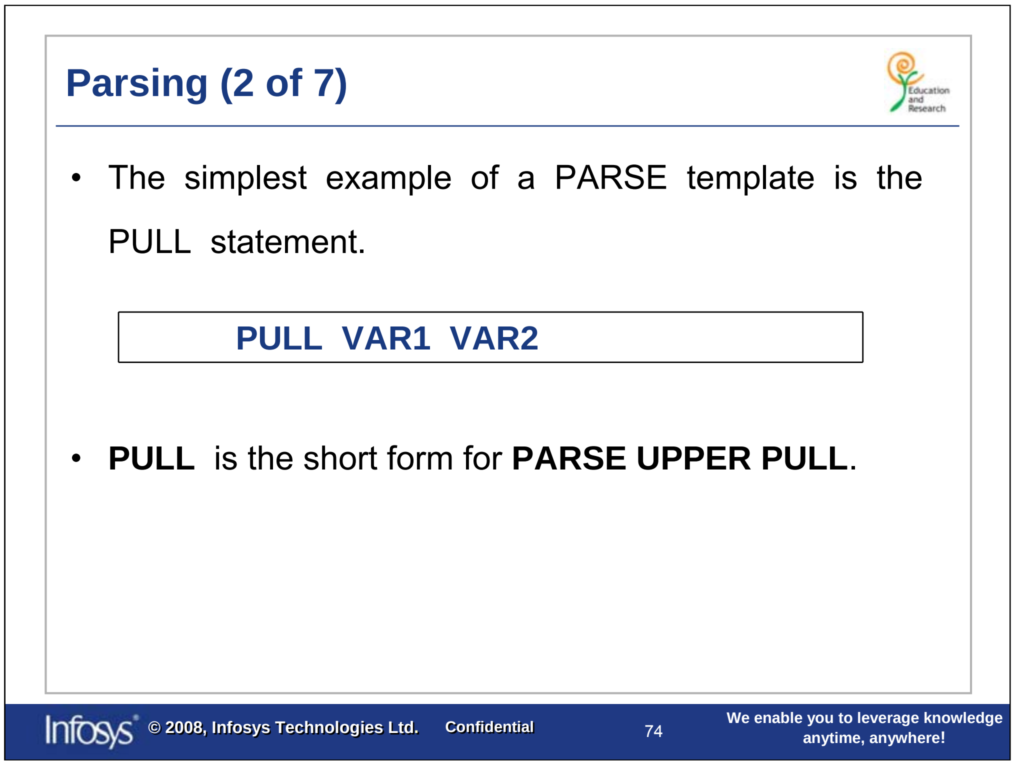
© 2008, Infosys Technologies Ltd. Confidential 73 We enable you to leverage knowledge anytime, anywhere!

Parsing (1 of 7) - Notes

The REXX language has a powerful parsing capability that can be used to change the format of the user's input. Parsing means breaking input string (s) into constituent parts and assigning them to one or more variables. The variables used to receive data after parsing are named in a template. A template is a model telling how to split the data.

A template can be as simple as a list of variable names to receive the data. The string that is being parsed is split into words . We define a word as a group of characters delimited by blanks. These constituent words of the string being parsed are assigned to the variables named in the template . More complex templates can contain patterns . These patterns specify how the input string is to be split . We would take a detailed look at such complex templates in the upcoming slides.

Please note that all the variables listed in the template will be assigned a new value regardless of the previous contents.



Parsing (2 of 7)

- The simplest example of a PARSE template is the PULL statement.

```
PULL  VAR1  VAR2
```

- **PULL** is the short form for **PARSE UPPER PULL**.

© 2008, Infosys Technologies Ltd. Confidential 74 We enable you to leverage knowledge anytime, anywhere!

Parsing (2 of 7) - Notes

The simplest form of parsing (i.e. a parse template) is the PULL statement. It accepts values either from the terminal or from the datastack.

e.g. PULL VAR1 VAR2.

The above statement will parse the input value into VAR1 and VAR2 (delimited by spaces by default). For example, if input value from the terminal is “first second”, the parsed values in the variables would be: VAR1 = FIRST & VAR2 = SECOND.

Please note that the PULL statement converts all the lower case letters to upper case unless specified.

The instruction PULL fname lname is actually the short form for the instruction PARSE UPPER PULL fname lname.

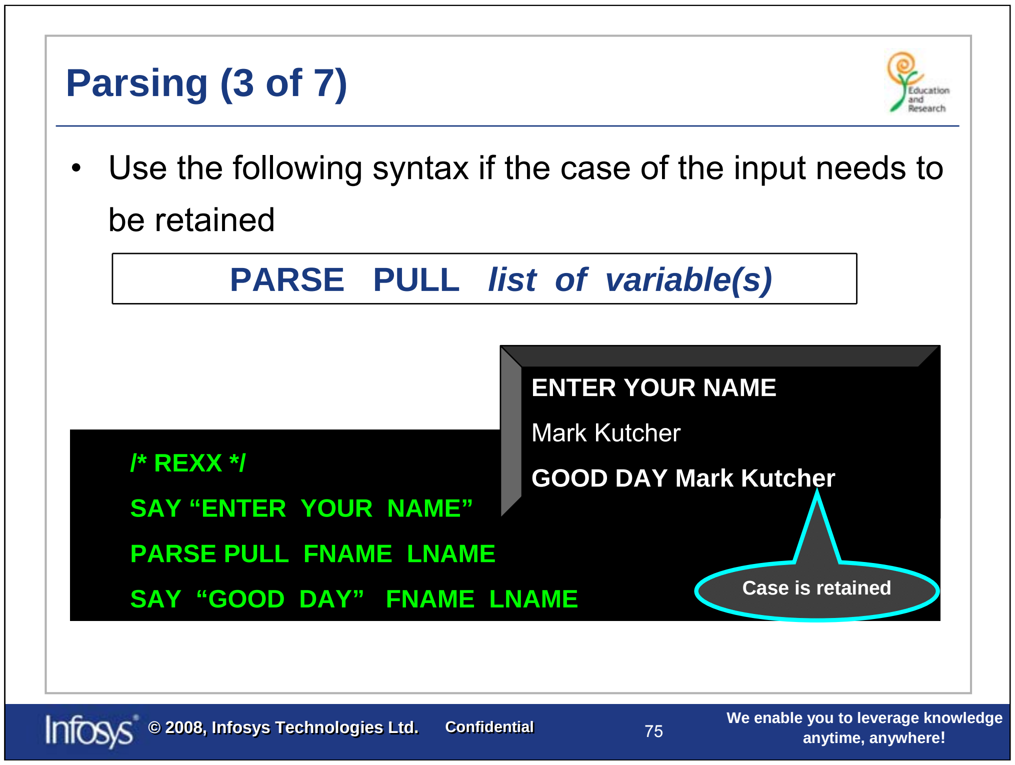
This is why the PULL statement automatically translates all input to upper case (unless otherwise specified as discussed in the next slide).

If we assume the input value from the terminal to be “first second”, the parsed values in the variables would be:

fname= FIRST

lname = SECOND.

Please note that the PULL statement converts all the lower case letters to upper case.



Parsing (3 of 7)

- Use the following syntax if the case of the input needs to be retained

PARSE PULL *list of variable(s)*

```
/* REXX */
SAY "ENTER  YOUR  NAME"
PARSE PULL  FNAME  LNAME
SAY  "GOOD  DAY"   FNAME  LNAME
```

```
ENTER YOUR NAME
Mark Kutcher
GOOD DAY Mark Kutcher
```

Case is retained

© 2008, Infosys Technologies Ltd. Confidential 75 We enable you to leverage knowledge anytime, anywhere!

Parsing (3 of 7) - Notes

If we want the case of the input accepted through the PULL statement (either from the datastack or from the terminal) to be retained , then we can do so by using the full form of the PULL statement but just dropping the UPPER clause .

i.e. Instead of using

PULL list of variable(s)

use

PARSE PULL list of variable(s)

Please note that we have dropped the UPPER clause from the full form of the PULL statement.

e.g. PARSE PULL fname lname

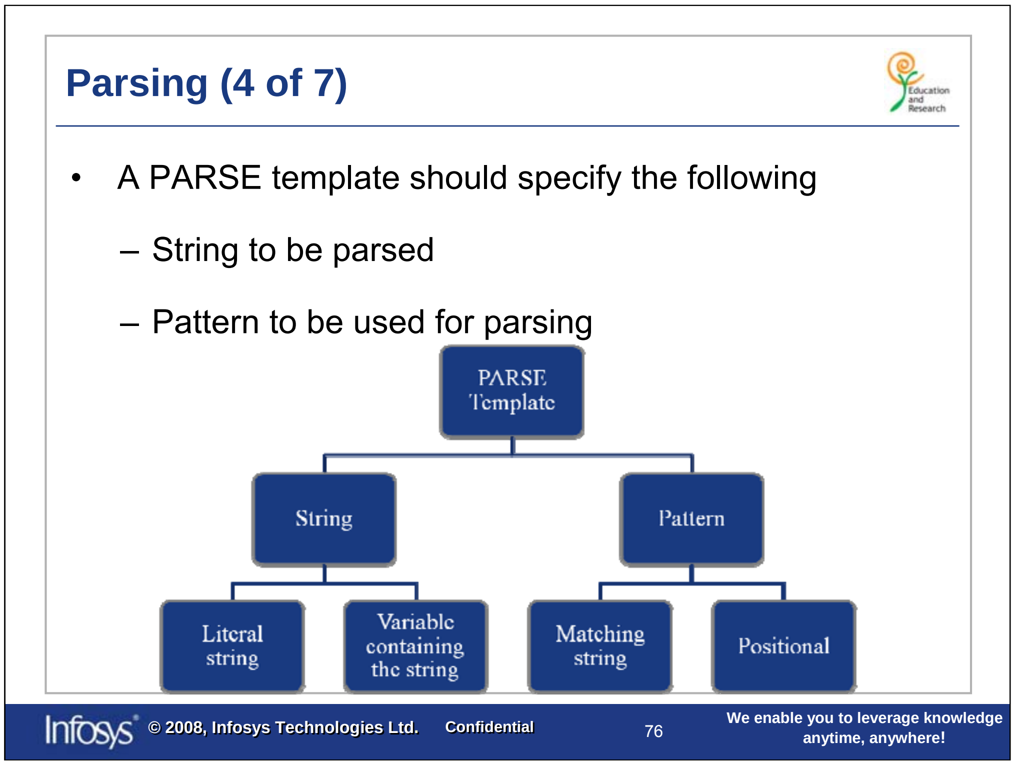
If the input from the terminal is "Mark Kutcher" then the values in the variables named in the template are

fname = Mark

lname = Kutcher

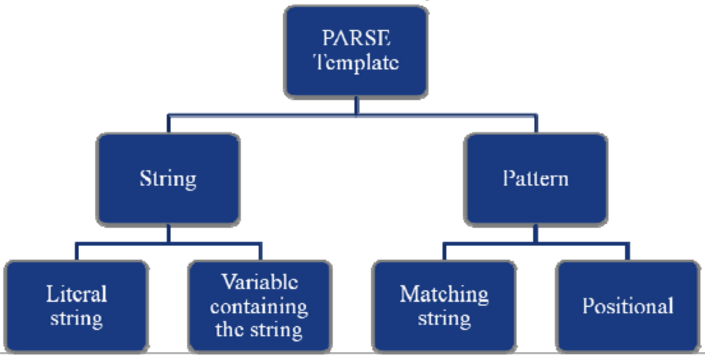
Please note that in this case , the input is not translated to upper case . Instead, the case of the input is retained .

Also note that there is no keyword called LOWER in the syntax to convert input to lower case.



Parsing (4 of 7)

- A PARSE template should specify the following
  - String to be parsed
  - Pattern to be used for parsing



Parsing (4 of 7) - Notes

A PARSE template should specify the following

•String to be parsed

•Pattern used for parsing

To be able to use the powerful PARSE statement in REXX , we need to first of all specify some string to be parsed in the PARSE statement .

As represented in the above figure, a parse template requires information about :

a.The string to be parsed

b.Pattern in which the string should be parsed

The string can be specified in the PARSE statement in one of the following ways:

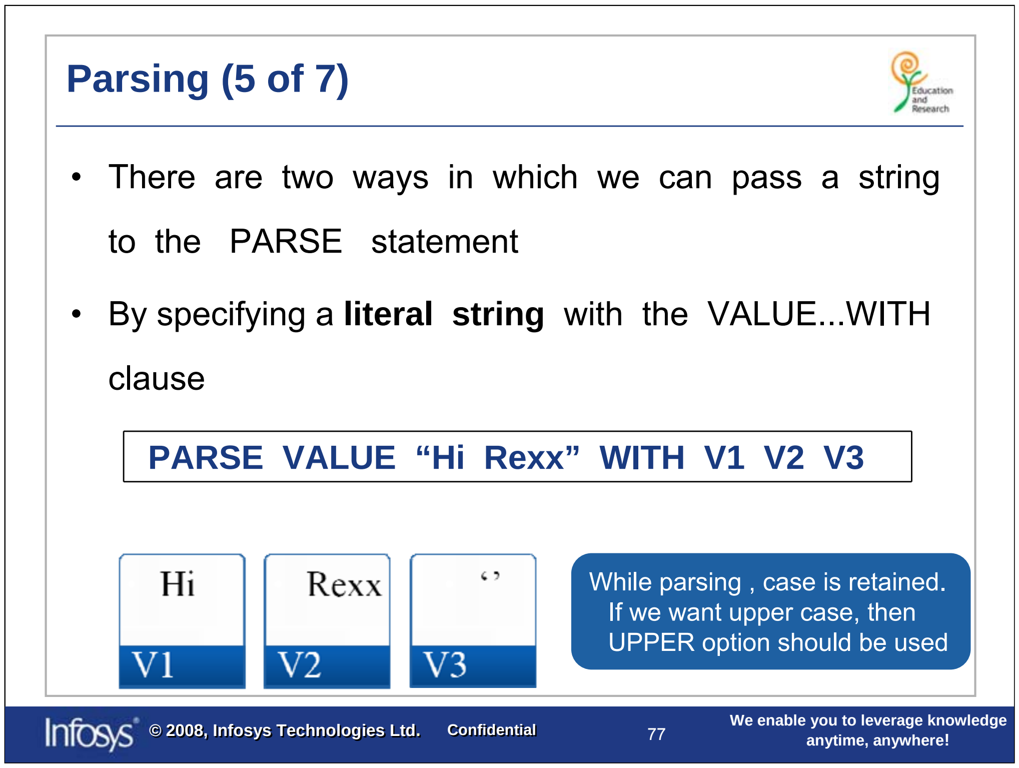
a.Specifying the literal string with the VALUE .....WITH clause

b.Specifying a variable containing the string using the VAR clause

Parsing patterns are of 2 types:

a.Matching string

b.Positional pattern



Parsing (5 of 7)

- There are two ways in which we can pass a string to the PARSE statement
- By specifying a **literal string** with the VALUE...WITH clause

PARSE VALUE "Hi Rexx" WITH V1 V2 V3

Hi
Rexx
' '
V1
V2
V3

While parsing , case is retained. If we want upper case, then UPPER option should be used

Parsing (5 of 7) - Notes

The PARSE VALUE...WITH instruction parses a specified literal string into one or more variables whose names follow the WITH keyword.

PARSE VALUE does not convert the parsed data to upper case before assigning it into variables. If you want uppercase translation, use PARSE UPPER VALUE.

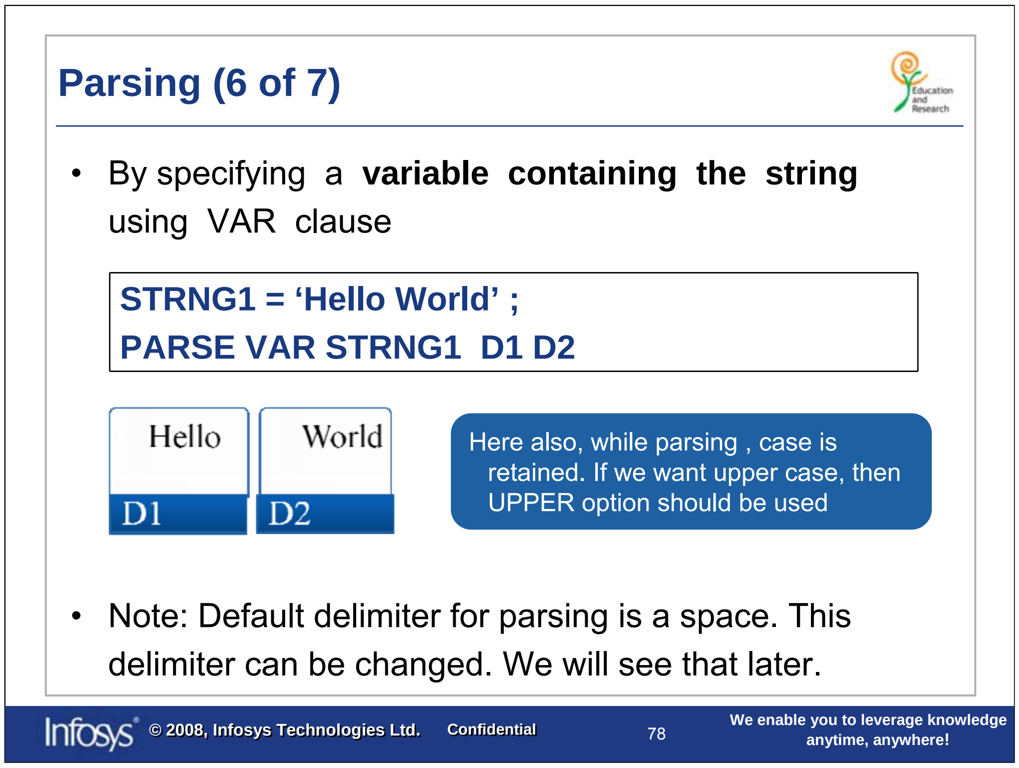
In the example shown in the slide, the value "Hi Rexx" is to be parsed into the variables named in the template viz. V1 ,V2 and V3 .

V1 will contain "Hi" , V2 will contain "Rexx" and V3 will be assigned NULL . If V3 had contained some other value (other than NULL) prior to the execution of the PARSE statement , then that value will be lost (overwritten).

Also please note that the case of the input string has been retained . If we want upper case translation to occur , we are to use

PARSE UPPER VALUE "Hi Rexx" WITH V1 V2 V3

Now V1 will contain "HI" , V2 will contain "REXX" and V3 will contain NULL



Parsing (6 of 7)

- By specifying a **variable containing the string** using VAR clause

```
STRNG1 = 'Hello World' ;
PARSE VAR STRNG1  D1 D2
```

Hello
World
D1
D2

Here also, while parsing , case is retained. If we want upper case, then UPPER option should be used

- Note: Default delimiter for parsing is a space. This delimiter can be changed. We will see that later.

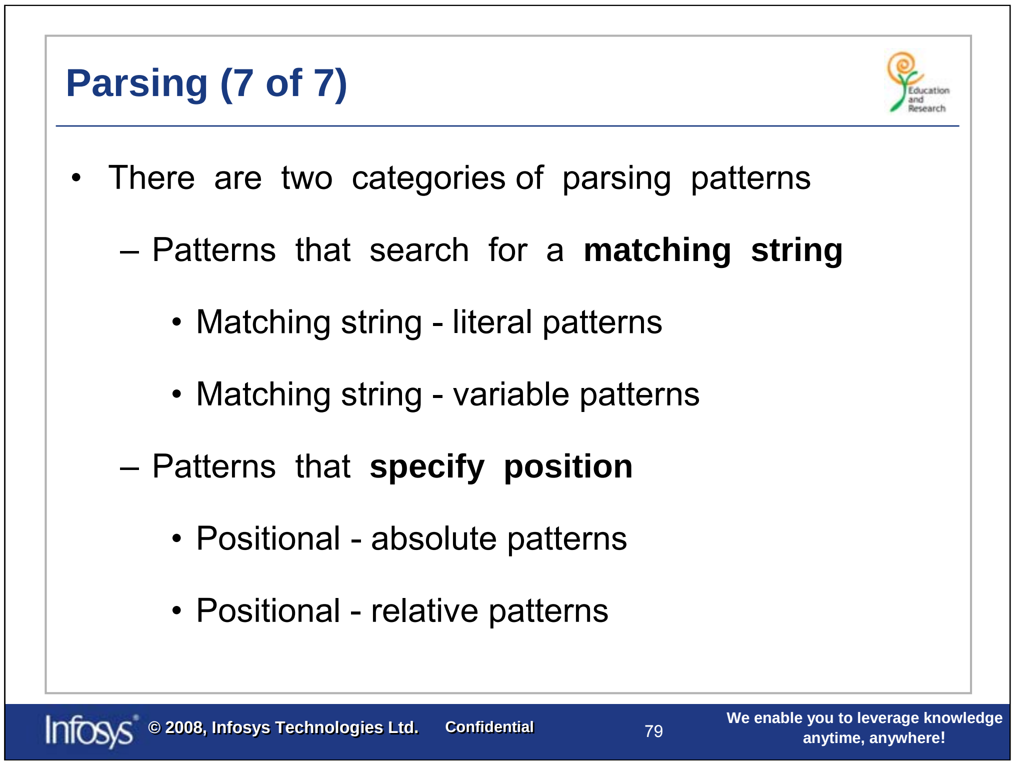
Parsing (6 of 7) - Notes

The PARSE VAR instruction parses a string contained in a specified variable into one or more variables(as named in the template). PARSE VAR does not convert the parsed data to upper case before assigning it into variables. If you want uppercase translation, use PARSE UPPER VAR.

In the example shown in the slide, the string "Hello World" is to be parsed into the variables named in the template viz. D1 and D2. D1 will contain "Hello" and D2 will contain "World" . Please note that the case of the input string has been retained . If we want upper case translation to occur , we are to use

PARSE UPPER VAR STRNG1 D1 D2

Now D1 will contain "HELLO" and D2 will contain "WORLD"



Parsing (7 of 7)

- There are two categories of parsing patterns
  - Patterns that search for a **matching string**
    - Matching string - literal patterns
    - Matching string - variable patterns
  - Patterns that **specify position**
    - Positional - absolute patterns
    - Positional - relative patterns

Parsing (7 of 7) - Notes

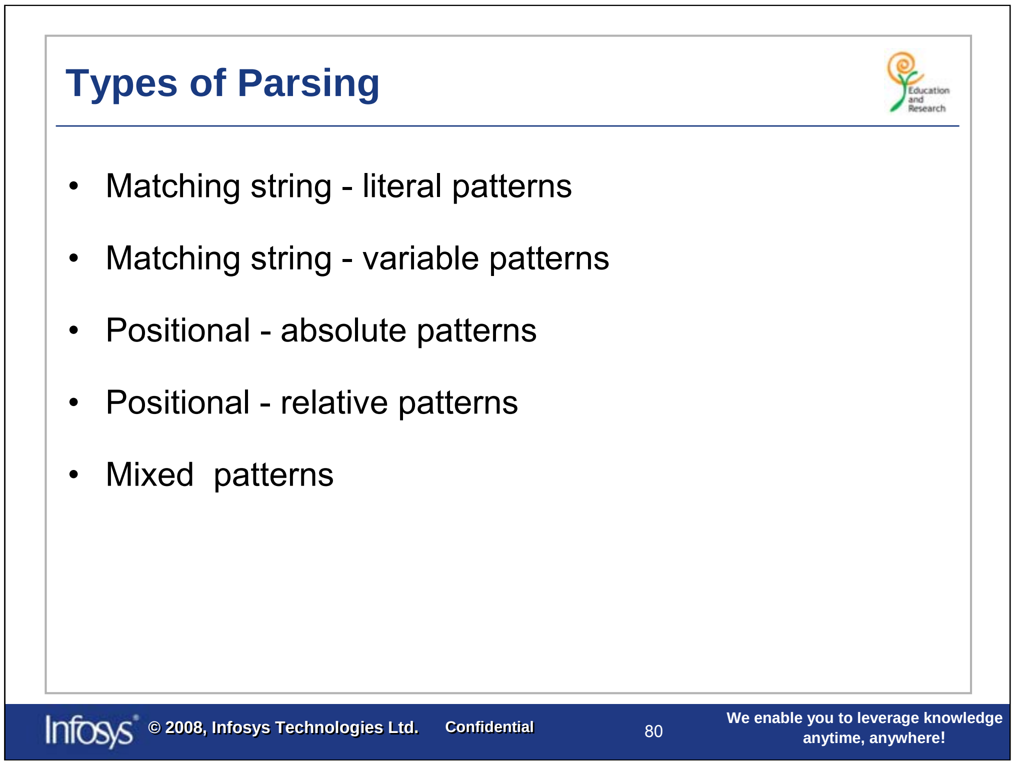
We can specify complicated parsing patterns in the PARSE statement . These parsing patterns in turn specify different flexible ways in which to break-up or split the input string and store them in the variables named in the template . We can generally talk of two types of parsing patterns :

a) Patterns that search for a matching string (literal string patterns and variable string patterns)

b) Patterns that specify position (absolute patterns and relative patterns)

We shall look at these different types of parsing patterns that can be specified in the PARSE template ,they are

Literal string patterns, Variable string patterns, Absolute patterns and Literal patterns.



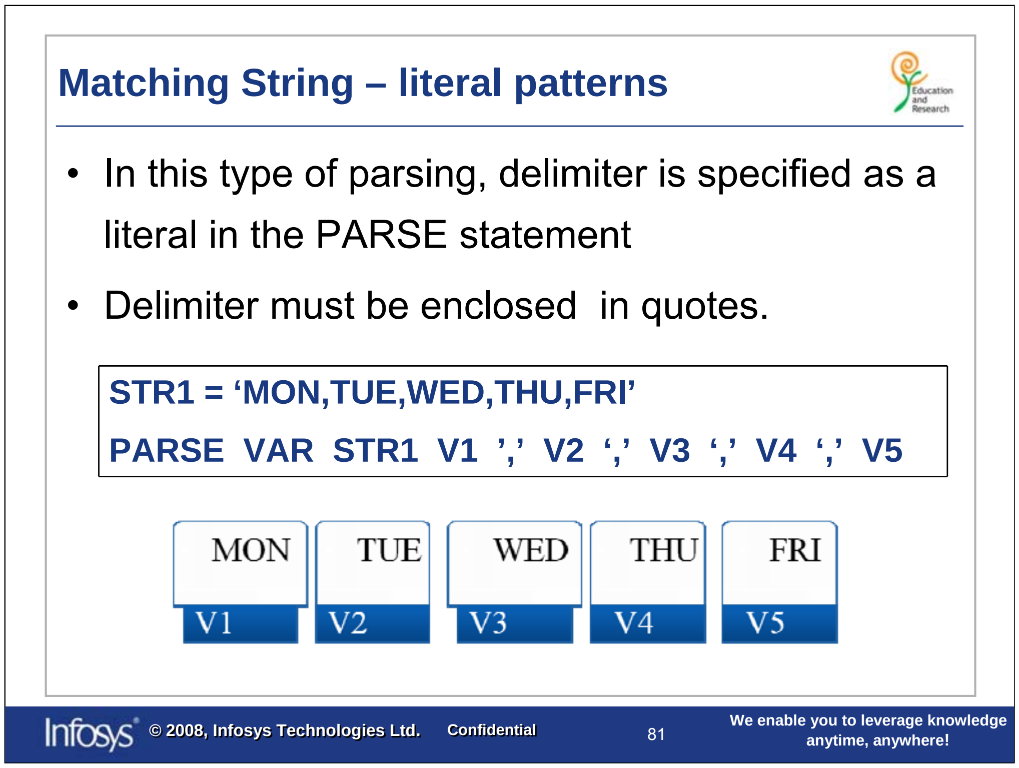
Types of Parsing

- Matching string - literal patterns
- Matching string - variable patterns
- Positional - absolute patterns
- Positional - relative patterns
- Mixed patterns

80

Types of parsing - Notes

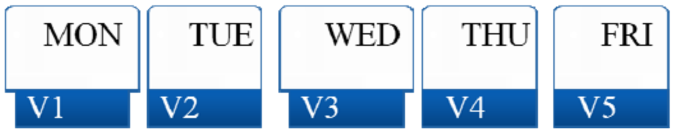
The various type of parsing are listed here. We shall discuss each of these now in detail.



Matching String – literal patterns

- In this type of parsing, delimiter is specified as a literal in the PARSE statement
- Delimiter must be enclosed in quotes.

```
STR1 = ‘MON,TUE,WED,THU,FRI’
PARSE  VAR  STR1  V1  ’,’  V2  ‘,’  V3  ‘,’  V4  ‘,’  V5
```



Infosys © 2008, Infosys Technologies Ltd. Confidential 81 We enable you to leverage knowledge anytime, anywhere!

Matching String - literal patterns - Notes

In this type of parsing, word delimiters can be specified for the parse template (space is default delimiter). The most wide spread use of this delimiter pattern is in separating day, month and year from a given date variable.

Example:

STR2 = ‘15/12/97’

PARSE VAR STR2 DD ’/’ MM ‘/’ YY

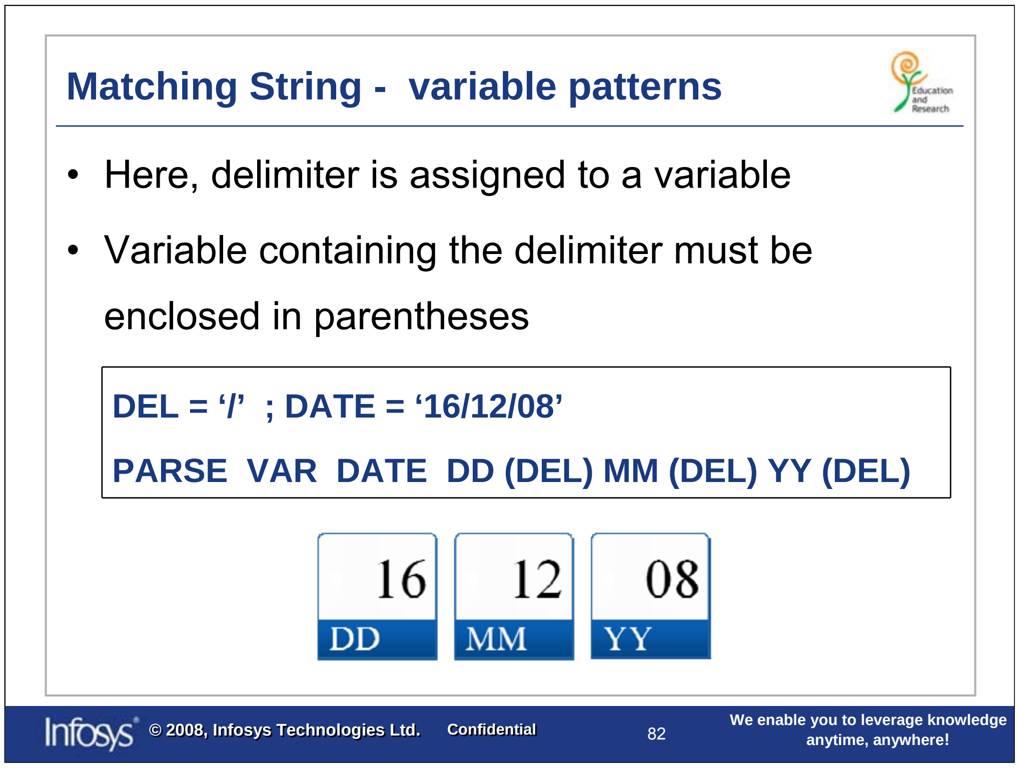
‘/’ will be used as delimiter here.

The result after parsing will be as follows: DD= 15, MM = 12 and YY = 97

Ex: **STR1 = ‘MON,TUE,WED,THU,FRI’**

PARSE VAR STR1 V1 ’,’ V2 ‘,’ V3 ‘,’ V4 ‘,’ V5

In this example, comma will be used as the delimiter while parsing & appropriate values will be moved to respective variables.



Matching String - variable patterns

- Here, delimiter is assigned to a variable
- Variable containing the delimiter must be enclosed in parentheses

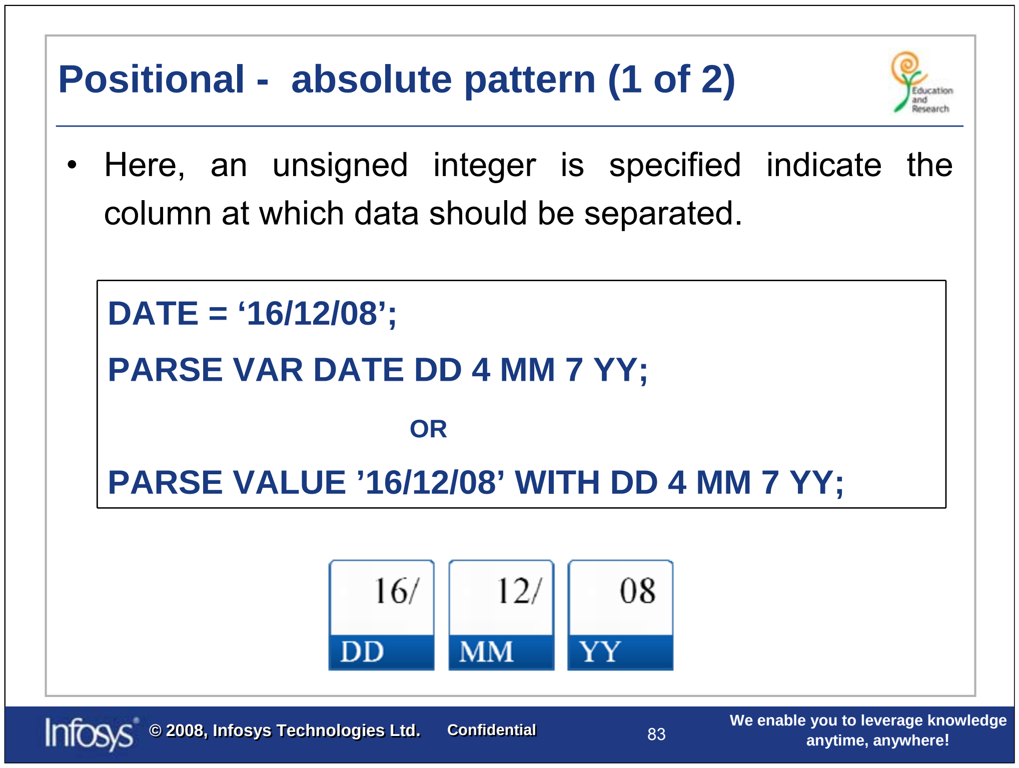
```
DEL = ‘/’  ; DATE = ‘16/12/08’
PARSE  VAR  DATE  DD (DEL) MM (DEL) YY (DEL)
```

16
DD
12
MM
08
YY

Matching String - variable patterns - Notes

When you do not know the delimiter in advance, the previous parsing type would not be of any use. With the matching string variable pattern type, you can use a variable enclosed in parentheses to be replaced as a delimiter.

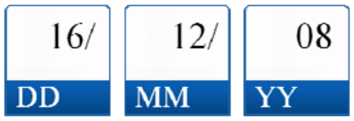
In this type of parsing, delimiter is assigned to a variable & this variable is specified in the PARSE statement.



Positional - absolute pattern (1 of 2)

- Here, an unsigned integer is specified indicate the column at which data should be separated.

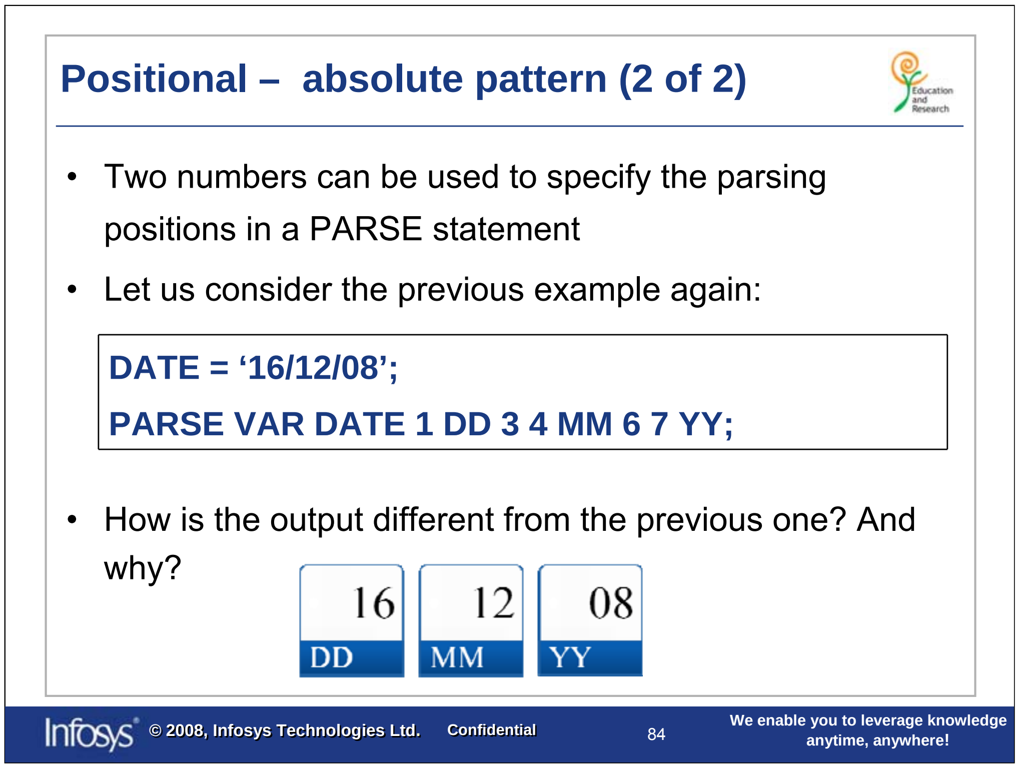
```
DATE = '16/12/08';
PARSE VAR DATE DD 4 MM 7 YY;
                    OR
PARSE VALUE '16/12/08' WITH DD 4 MM 7 YY;
```



Positional – absolute pattern (1 of 2) - Notes

Numbers can be used in a template to indicate the column at which the data should be separated. An absolute column position is always indicated by an unsigned integer. You can use this parsing technique only when you are sure about the string pattern and that it remains the same always.

In the example shown in the slide, the part of the string starting from the 1st character to the 3rd character (including 3rd character) is stored in DD, the part of the string from the 4th character to the 6th character (including the 6th character) is stored in MM and the rest of the string from the 7th character onwards is stored in YY.

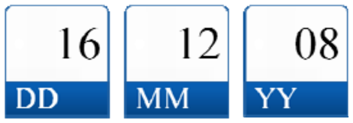


Positional – absolute pattern (2 of 2)

- Two numbers can be used to specify the parsing positions in a PARSE statement
- Let us consider the previous example again:

```
DATE = ‘16/12/08’;
PARSE VAR DATE 1 DD 3 4 MM 6 7 YY;
```

- How is the output different from the previous one? And why?



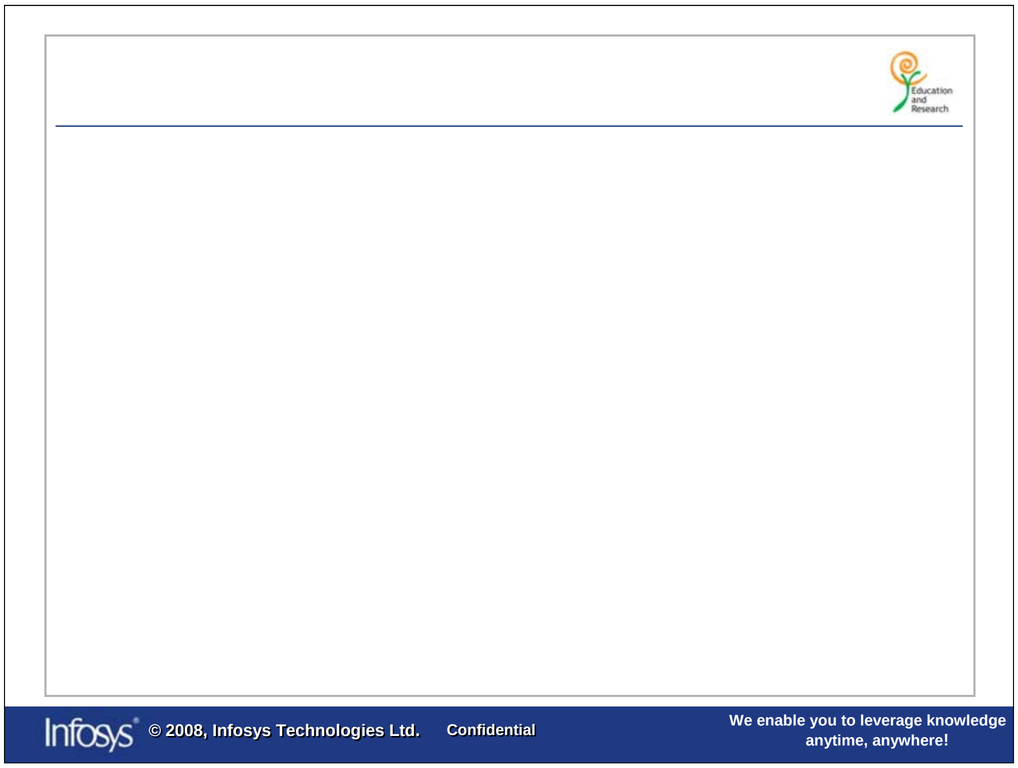
Positional – absolute pattern (2 of 2) - Notes

Two numbers can be used to specify the parsing positions in a template .

Let us consider the previous example again:

```
DATE = ‘16/12/08’
PARSE VAR DATE WITH 1 DD 3 4 MM 6 7 YY
```

Here, unlike in the previous example, two numbers are specified. One number indicates the ending position for the previous variable & the other indicates, starting position for the next variable. In the current example, for DD 1 is the starting position & 3 is the ending position. Thus DD will contain 16. For MM, 4 is the starting position & 6 is the ending position. Thus MM=12 & for YY, 7 is the starting position. Thus YY=08.



Education and Research

Example: STRING = ‘ABCDEFGHIJKLM’

```
PARSE VAR STRING WITH 1 V1 3 2 V2 6 5 V3 11
```

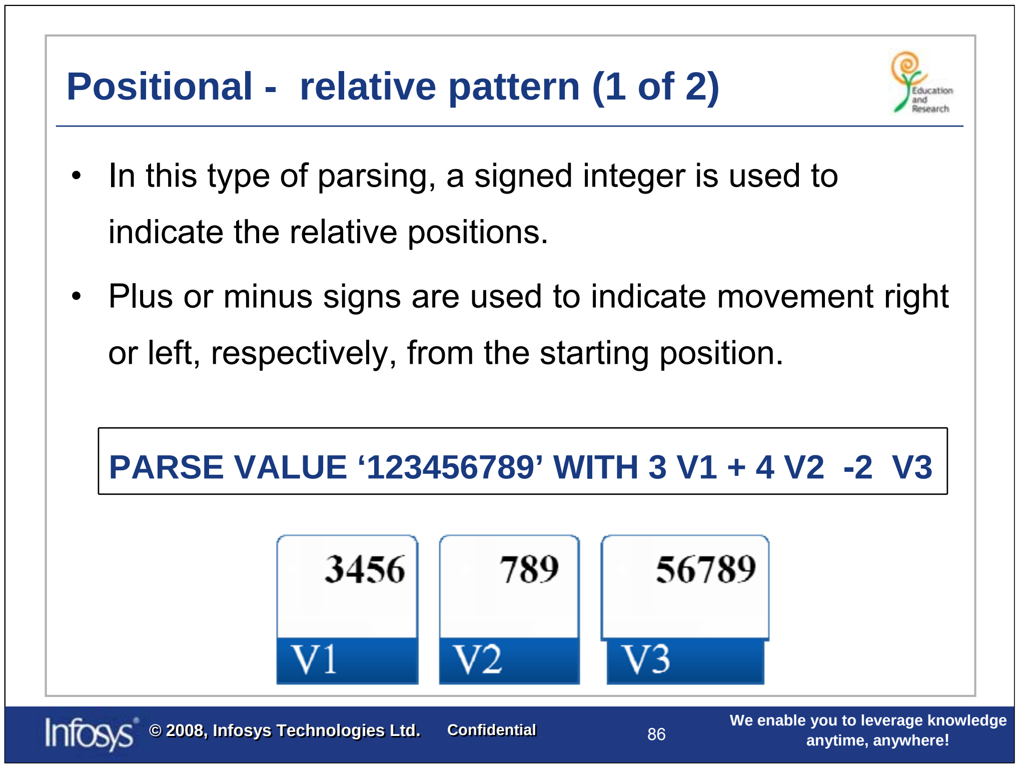
```
Output : V1 = AB  (1st to 2nd)
         V2 = BCDE  (2nd to 5th)
         V3 = EFGHIJ (5th to 10th)
```

In the above example, this is how the parsing template is interpreted :

Start with 1st position and put the string up to 2nd position in V1. i.e. Stop before the 3rd position.

Then start with 2nd position and put string up to 5th position in V2 i.e. Stop before the 6th position.

and then start with 5th position and put string up to 10th position in V3. i.e. Stop before the 11th position.



Positional - relative pattern (1 of 2)

- In this type of parsing, a signed integer is used to indicate the relative positions.
- Plus or minus signs are used to indicate movement right or left, respectively, from the starting position.

PARSE VALUE ‘123456789’ WITH 3 V1 + 4 V2 -2 V3

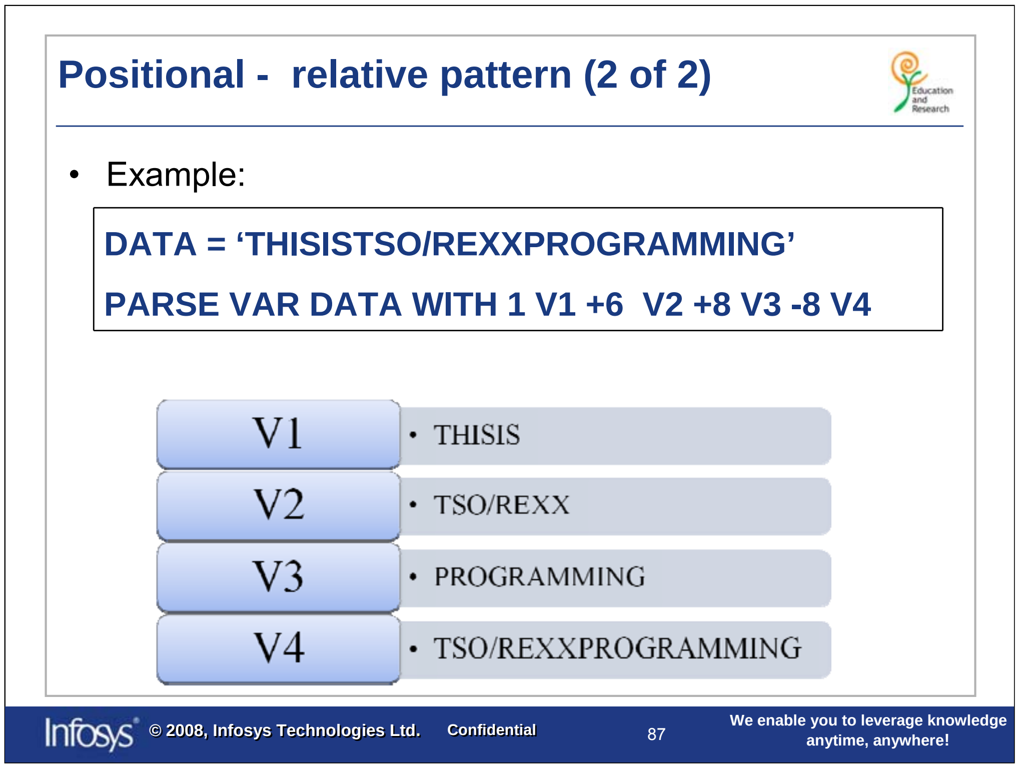
3456
789
56789
V1
V2
V3

Positional – relative pattern (1 of 2) - Notes

You can also indicate the position as a signed integer, which indicates a relative column position. If a signed integer is specified in the template, then it indicates relative position. In this case, data is separated according to the relative column position. The plus or minus sign indicates movement right or left, respectively, from the starting position.

Example: PARSE VALUE ‘123456789’ WITH 3 V1 + 4 V2 -2 V3

This says that at first, 3 is the starting position. +4 indicates the length of 4. So start with 3rd position and put a substring of length 4 into V1. That makes the position of next selection 3 + 4 = 7. -2 (in fact, any negative number) indicates that take the substring till end of the input string. So start with 7th position and take the rest of the whole string and put into V2. Now, position for next selection is 7 - 2 = 5. Start with 5th position and take rest of the string (as nothing is specified for the last variable V3) and put into V3. Finally the variables values after parsing would be: V1 = 3456, V2 = 789 and V3 = 56789.



Positional - relative pattern (2 of 2)

- Example:

```
DATA = 'THISISTSO/REXXPROGRAMMING'
PARSE VAR DATA WITH 1 V1 +6  V2 +8 V3 -8 V4
```

V1	• THISIS
V2	• TSO/REXX
V3	• PROGRAMMING
V4	• TSO/REXXPROGRAMMING

Infosys® © 2008, Infosys Technologies Ltd. Confidential 87 We enable you to leverage knowledge anytime, anywhere!

Positional – relative pattern (2 of 2) – Notes

Example: **DATA = 'THISISTSO/REXXPROGRAMMING'**

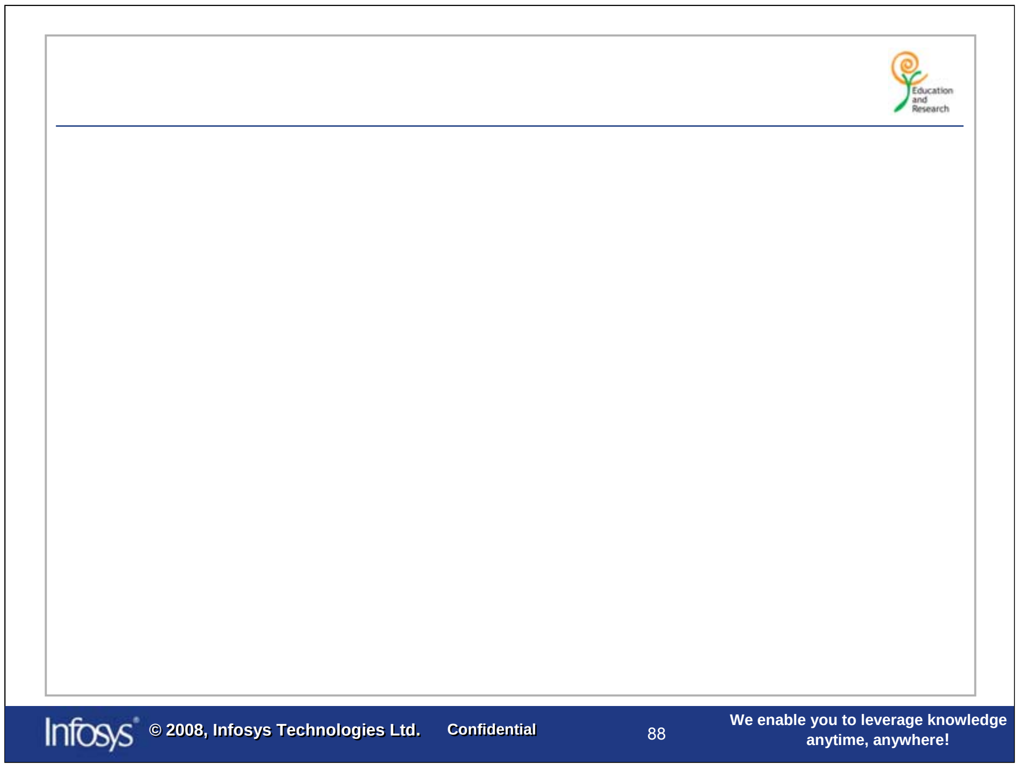
PARSE VAR DATA WITH 1 V1 +6 V2 +8 V3 -8 V4

In the above example, parsing starts at position 1. For V1 the length is mentioned as +6. Thus V1 will contain THISIS.

For V2, starting position is now 7 & length is given as +8. Thus V2=TSO/REXX

For V3, starting position is now 15 & length is specified as -8. Thus V3 will contain entire string starting from column 15 till end. V3 = PROGRAMMING.

For V4, the starting position is 15-8, which is 7. End position is not mentioned for the same. Hence V4 = TSO/REXXPROGRAMMING



Example: **DATA = ‘THIS IS FOR THE EXAMPLE’**

PARSE VAR DATA 6 V1 -3 V2 +6 V3 +10 V4

Output :

V1 = ‘IS FOR THE EXAMPLE’

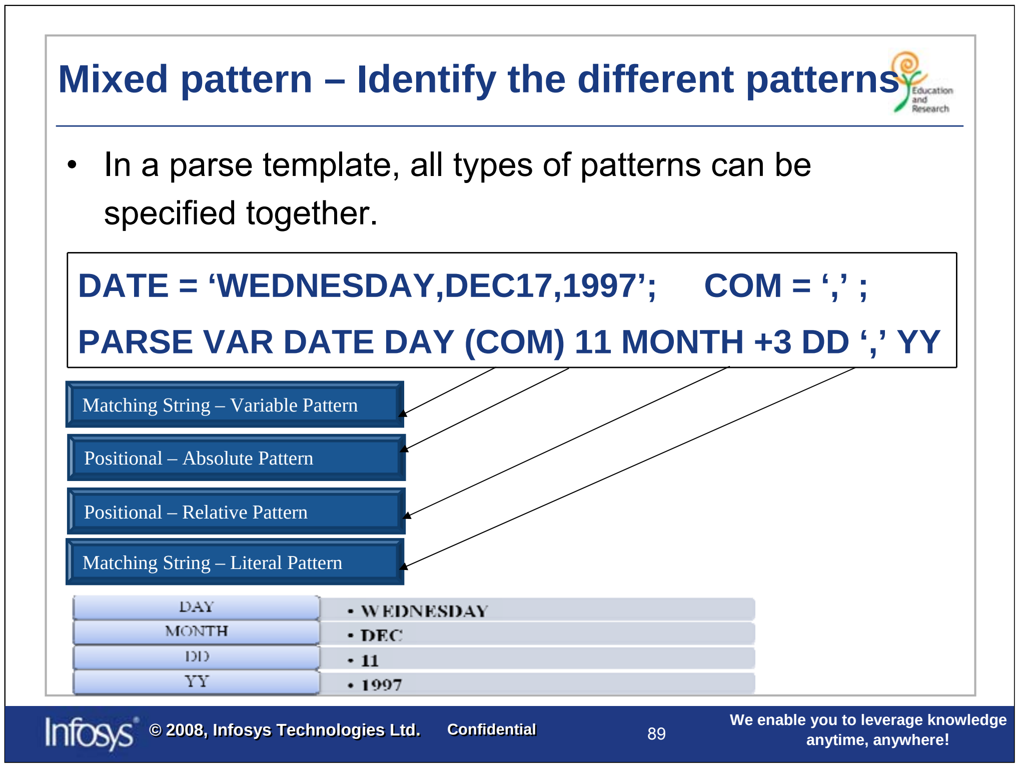
V2 = ‘IS IS’

V3 = ‘FOR THE EX’

V4 = ‘AMPLE’

Given above is one more example to make this concept more clear.

Start with position 6, since -3 is given as length so take rest of the string and put into V1. i.e., V1 = ‘IS FOR THE EXAMPLE’. Now position is 6 - 3 = 3rd. Start with 3rd position, take a substring of length 6(as indicated by +6) and put into V2. i.e., V2= ‘IS IS ’. Now position is 3 + 6 = 9. Start with 9th position, take the substring of length 10 (indicated by +10) and put into V3. i.e., V3 = ‘FOR THE EX’. Position is now 9 + 10 = 19. Start with 19th position and take rest of the string and put into V4. i.e., V4 = AMPLE



Mixed pattern – Identify the different patterns

- In a parse template, all types of patterns can be specified together.

```
DATE = ‘WEDNESDAY,DEC17,1997’;     COM = ‘,’ ;
PARSE VAR DATE DAY (COM) 11 MONTH +3 DD ‘,’ YY
```

Matching String – Variable Pattern
Positional – Absolute Pattern
Positional – Relative Pattern
Matching String – Literal Pattern

DAY	• WEDNESDAY
MONTH	• DEC
DD	• 11
YY	• 1997

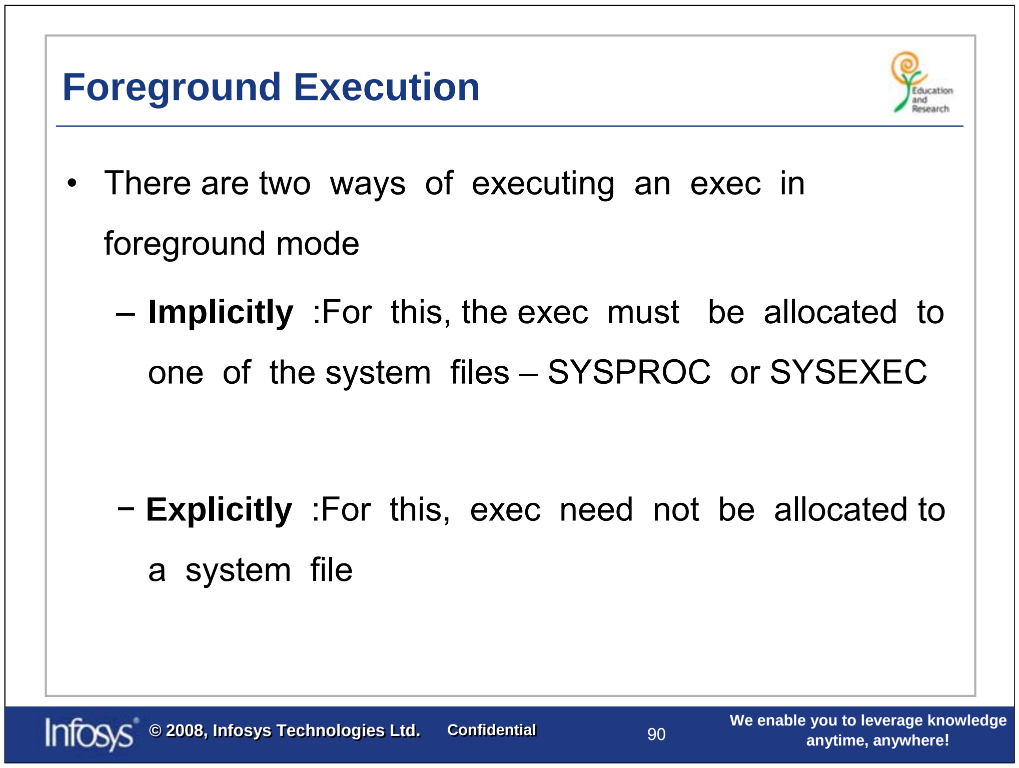
Mixed pattern - Identify the different patterns - Notes

Our parsing requirements most of the time might require applying a combination of all the different types of parsing studied till now. This is called mixed pattern.

Examples:

```
        DATE = ‘WEDNESDAY,DEC 17,1997’
        COM = ‘,’
  PARSE VAR DATE DAY (COM) 11 MONTH +3 DD ‘,’ YY
```

Now start with the first position and take a substring delimited by COM (that is comma ‘,’) and put into DAY. So it will put the value WEDNESDAY into variable DAY. Start with position 11 and put a substring of length 3 into variable MONTH which will put ‘DEC‘ into MONTH. So position for the next selection is 11 + 3 = 14. Start at position 14 and put the substring delimited by ‘,’ into DD which will put ‘17’ into DD & the remaining string (year) will be placed in YY. Thus YY will contain 1997

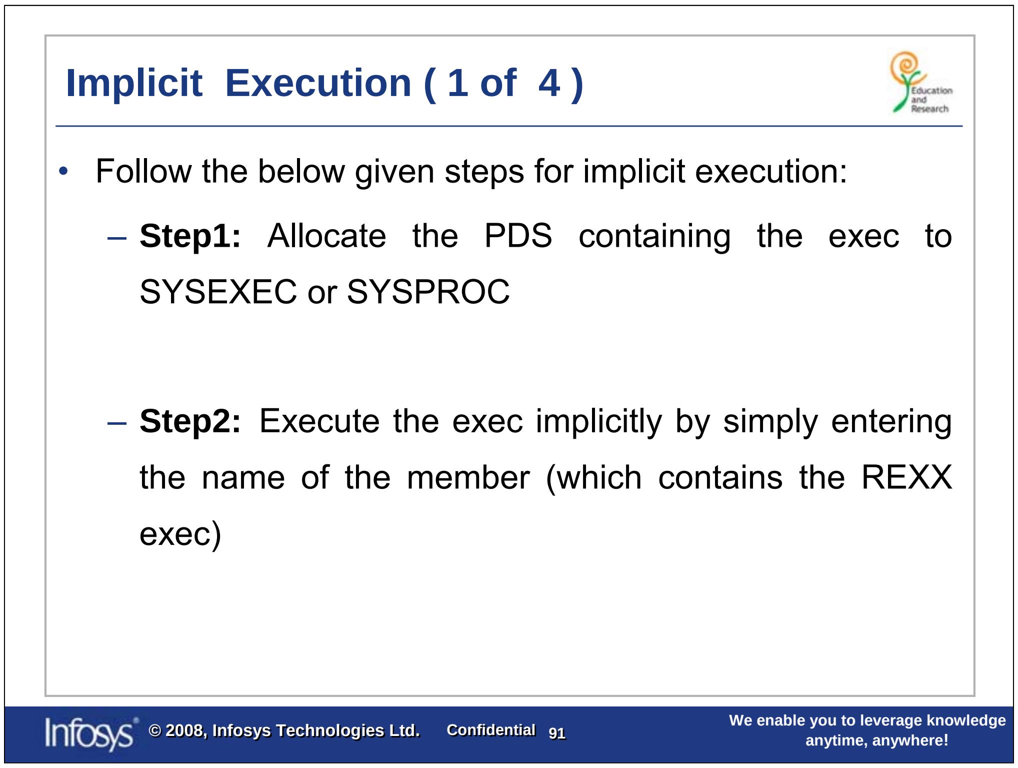


Foreground Execution

- There are two ways of executing an exec in foreground mode
  - **Implicitly** :For this, the exec must be allocated to one of the system files – SYSPROC or SYSEXEC
  - **Explicitly** :For this, exec need not be allocated to a system file

Foreground Execution- Notes

Once we have created a REXX exec, how we execute the exec is dependent on whether or not the PDS containing our exec has been allocated to one of the two system files : SYSEXEC or SYSPROC . If the PDS containing our exec has been allocated to one of the system files – SYSPROC or SYSEXEC , then we can execute the exec implicitly . If this is not so , then we have to execute our exec explicitly . We will learn more about implicit and explicit execution as we proceed through the subsequent slides .

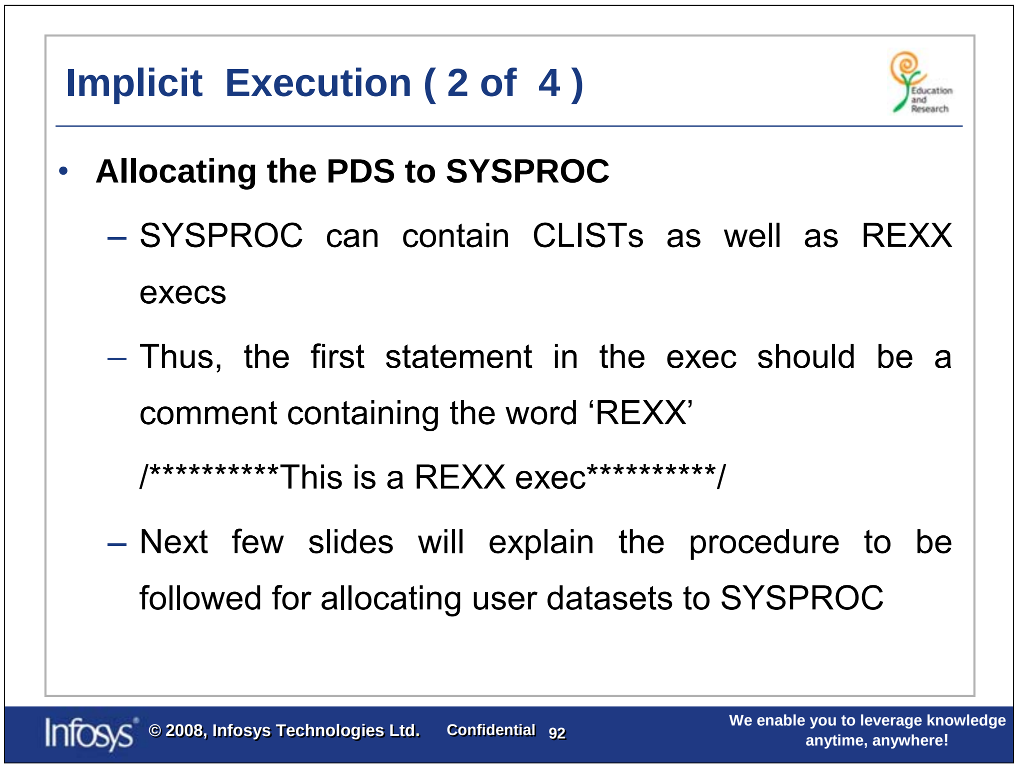


Implicit Execution (1 of 4)

- Follow the below given steps for implicit execution:
  - **Step1:** Allocate the PDS containing the exec to SYSEXEC or SYSPROC
  - **Step2:** Execute the exec implicitly by simply entering the name of the member (which contains the REXX exec)

Implicit Execution (1 of 4) - Notes

We shall now discuss about implicit execution . To implicitly execute an exec, we need to simply enter the member name of the data set that contains the exec . However before we do this , the PDS containing our exec must be allocated to one of the system files : SYSEXEC or SYSPROC



Implicit Execution (2 of 4)

- **Allocating the PDS to SYSPROC**
  - SYSPROC can contain CLISTs as well as REXX execs
  - Thus, the first statement in the exec should be a comment containing the word ‘REXX’

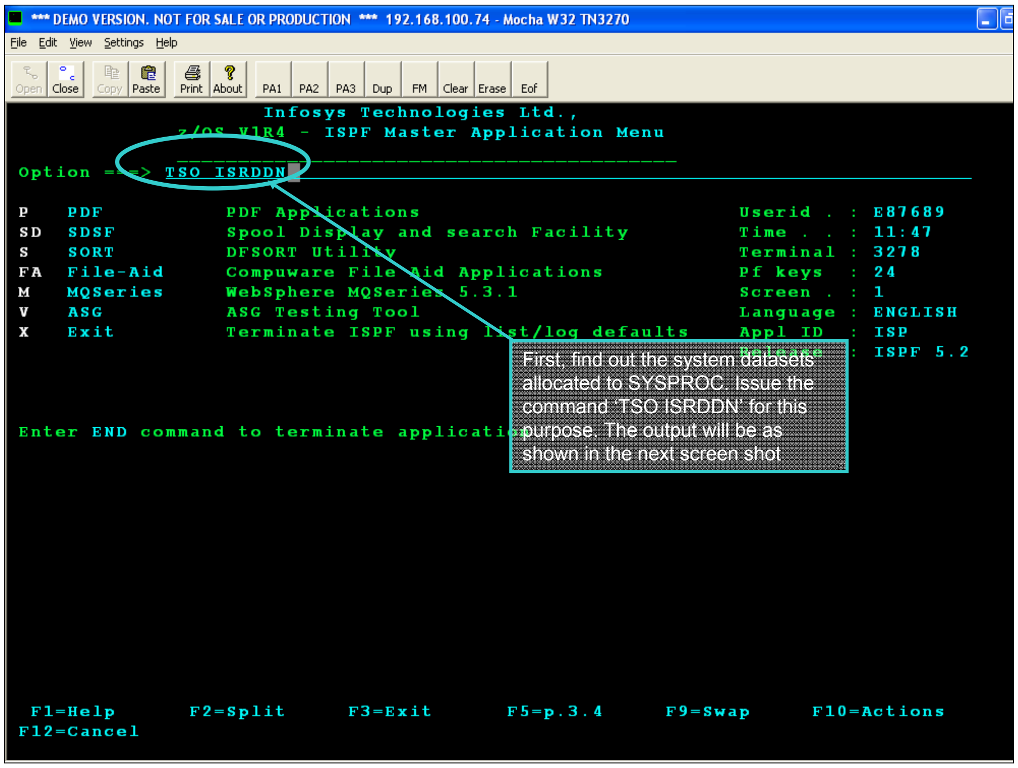
    /**********This is a REXX exec**********/
  - Next few slides will explain the procedure to be followed for allocating user datasets to SYSPROC

Infosys © 2008, Infosys Technologies Ltd. Confidential 92 We enable you to leverage knowledge anytime, anywhere!

Implicit Execution (2 of 4) - Notes

SYSPROC can contain both CLISTs and execs . An exec is distinguished from a CLIST by a comment statement containing the word REXX at the beginning of the exec .

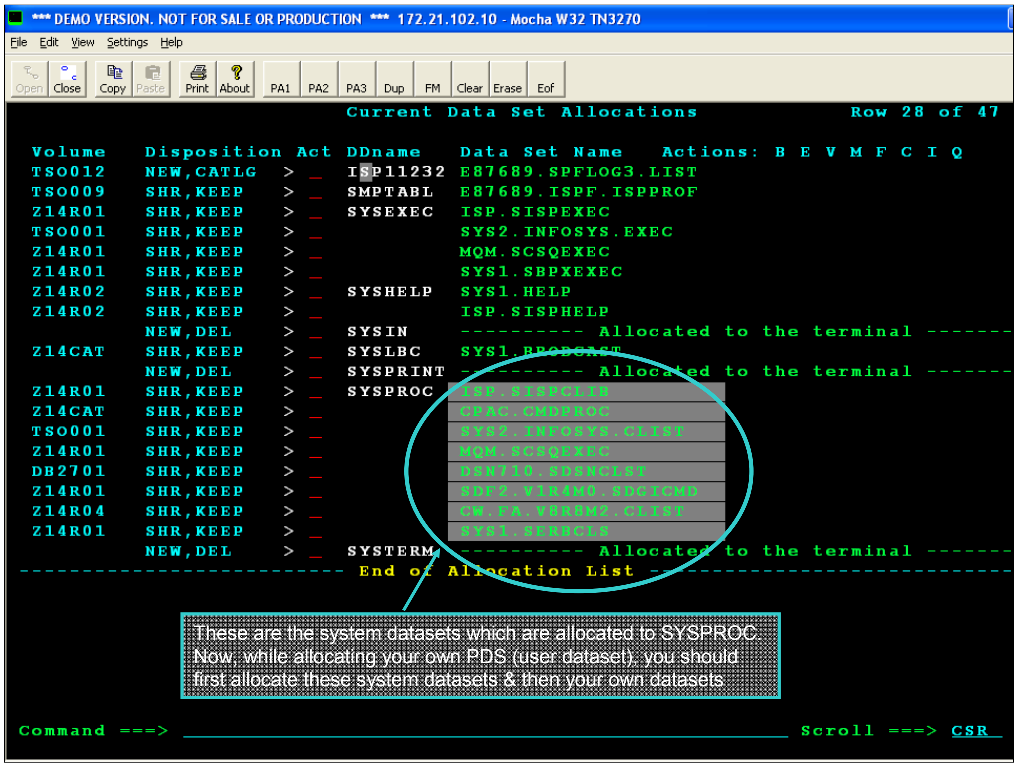
/******REXX*******/



*** DEMO VERSION. NOT FOR SALE OR PRODUCTION *** 192.168.100.74 - Mocha W32 TN3270
File Edit View Settings Help
Open Close Copy Paste Print About PA1 PA2 PA3 Dup FM Clear Erase Eof

```
                       Infosys Technologies Ltd.,
              z/OS V1R4 - ISPF Master Application Menu

Option ===> TSO ISRDDN

P    PDF            PDF Applications                        Userid . : E87689
SD   SDSF           Spool Display and search Facility       Time . . : 11:47
S    SORT           DFSORT Utility                          Terminal : 3278
FA   File-Aid       Compuware File Aid Applications         Pf keys  : 24
M    MQSeries       WebSphere MQSeries 5.3.1                Screen . : 1
V    ASG            ASG Testing Tool                        Language : ENGLISH
X    Exit           Terminate ISPF using list/log defaults  Appl ID  : ISP
                                                            Release  : ISPF 5.2


Enter END command to terminate application.

 F1=Help      F2=Split     F3=Exit      F5=p.3.4     F9=Swap     F10=Actions
F12=Cancel
```

First, find out the system datasets allocated to SYSPROC. Issue the command 'TSO ISRDDN' for this purpose. The output will be as shown in the next screen shot


*** DEMO VERSION. NOT FOR SALE OR PRODUCTION *** 172.21.102.10 - Mocha W32 TN3270
File Edit View Settings Help
Open Close Copy Paste Print About PA1 PA2 PA3 Dup FM Clear Erase Eof

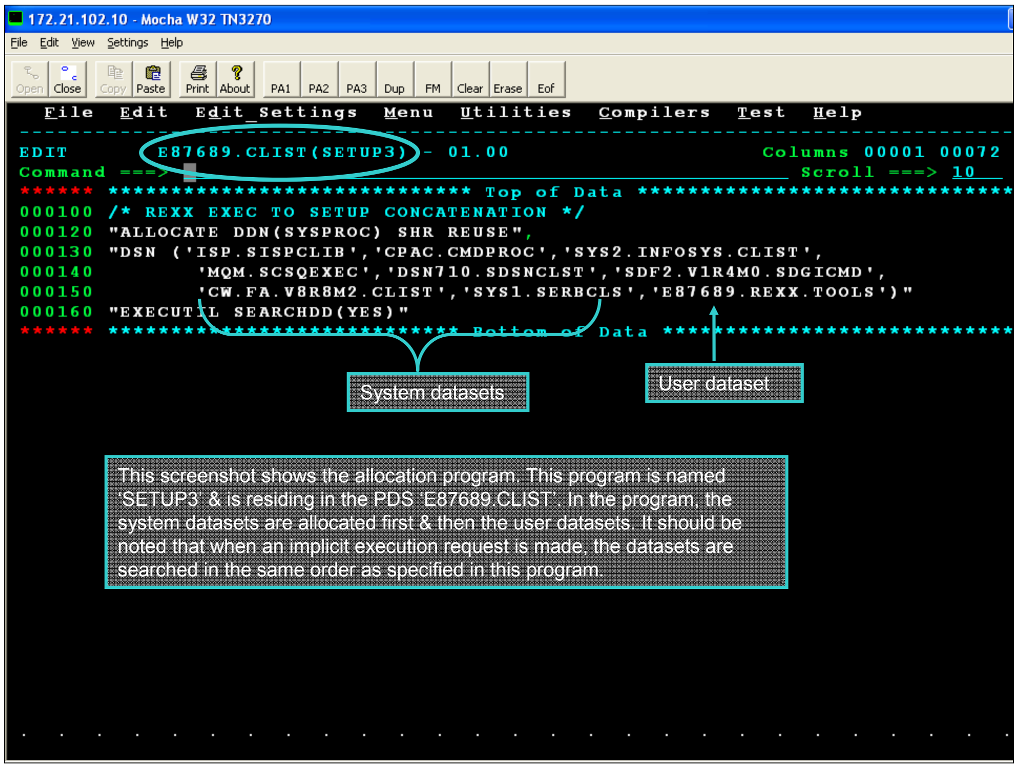
```
                                   Current Data Set Allocations                 Row 28 of 47

  Volume    Disposition  Act DDname    Data Set Name     Actions: B E V M F C I Q
  TSO012    NEW,CATLG   > _  ISP11232 E87689.SPFLOG3.LIST
  TSO009    SHR,KEEP    > _  SMPTABL  E87689.ISPF.ISPPROF
  Z14R01    SHR,KEEP    > _  SYSEXEC  ISP.SISPEXEC
  TSO001    SHR,KEEP    > _           SYS2.INFOSYS.EXEC
  Z14R01    SHR,KEEP    > _           MQM.SCSQEXEC
  Z14R01    SHR,KEEP    > _           SYS1.SBPXEXEC
  Z14R02    SHR,KEEP    > _  SYSHELP  SYS1.HELP
  Z14R02    SHR,KEEP    > _           ISP.SISPHELP
            NEW,DEL     > _  SYSIN    ---------- Allocated to the terminal -------
  Z14CAT    SHR,KEEP    > _  SYSLBC   SYS1.BRODCAST
            NEW,DEL     > _  SYSPRINT ---------- Allocated to the terminal -------
  Z14R01    SHR,KEEP    > _  SYSPROC  ISP.SISPCLIB
  Z14CAT    SHR,KEEP    > _           CPAC.CMDPROC
  TSO001    SHR,KEEP    > _           SYS2.INFOSYS.CLIST
  Z14R01    SHR,KEEP    > _           MQM.SCSQEXEC
  DB2701    SHR,KEEP    > _           DSN710.SDSNCLST
  Z14R01    SHR,KEEP    > _           SDF2.V1R4M0.SDGICMD
  Z14R04    SHR,KEEP    > _           CW.FA.V8R8M2.CLIST
  Z14R01    SHR,KEEP    > _           SYS1.SERBCLS
            NEW,DEL     > _  SYSTERM  ---------- Allocated to the terminal -------
 -------------------------- End of Allocation List --------------------------

 Command ===> ________________________________________________  Scroll ===> CSR
```

These are the system datasets which are allocated to SYSPROC. Now, while allocating your own PDS (user dataset), you should first allocate these system datasets & then your own datasets

The above screenshot shows the system datasets which are allocated to SYSPROC. Now, while allocating your own PDS, you should first allocate these system datasets & then your own datasets. If you allocate your datasets without allocating the system datasets, then you will not be able to use any CLIST or REXX EXEC which is in one of these system datasets. It is always a good practice to first allocate the system datasets & then the user datasets. This is shown in the next screenshot.

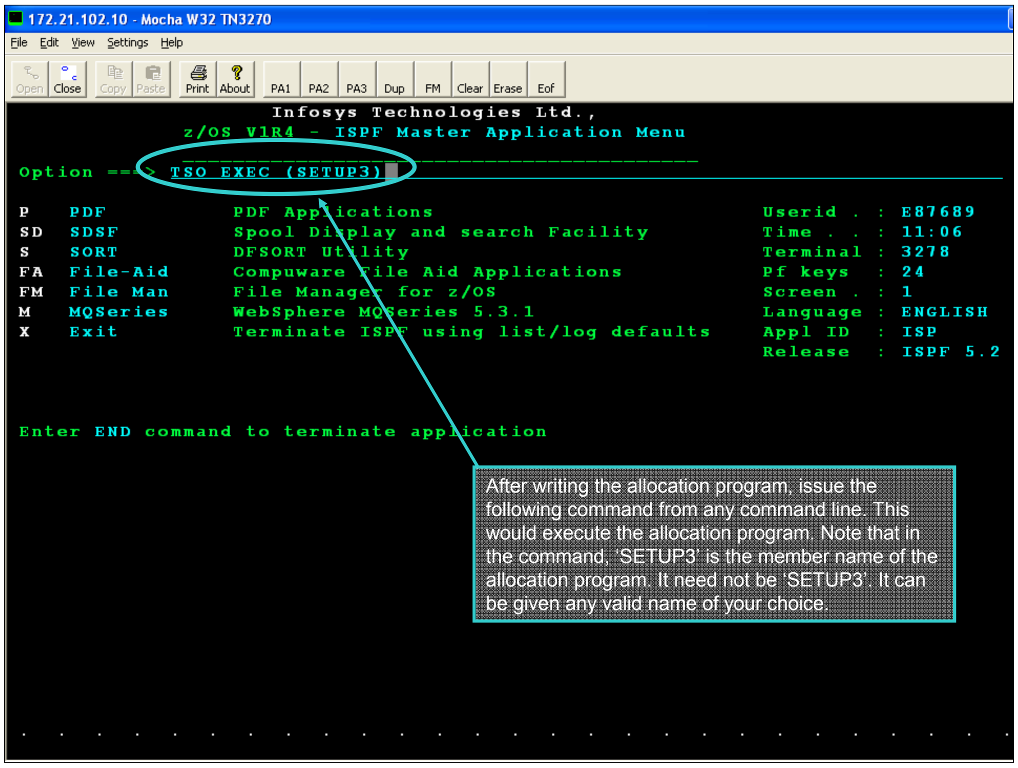
However, one should note that the allocation which we do will be active only for the current session. This means that if you logoff & then login again, your datasets will not be a part of SYSPROC (Only the system datasets will be a part of SYSPROC). Thus, you will have to run the allocation program each time you login in order to execute your rexx execs implicitly.

However, if this allocation program is made as a part of your logon procedure, then each time you login, automatically your datasets get allocated to SYSPROC along with the system datasets. The procedure to include the allocation program in the logon procedure is out of scope.


172.21.102.10 - Mocha W32 TN3270
File Edit View Settings Help
Open Close Copy Paste Print About PA1 PA2 PA3 Dup FM Clear Erase Eof
File Edit Edit_Settings Menu Utilities Compilers Test Help
System datasets
User dataset

```
EDIT       E87689.CLIST(SETUP3) - 01.00                 Columns 00001 00072
Command ===>                                                 Scroll ===> 10
****** ***************************** Top of Data ******************************
000100 /* REXX EXEC TO SETUP CONCATENATION */
000120 "ALLOCATE DDN(SYSPROC) SHR REUSE",
000130 "DSN ('ISP.SISPCLIB','CPAC.CMDPROC','SYS2.INFOSYS.CLIST',
000140        'MQM.SCSQEXEC','DSN710.SDSNCLST','SDF2.V1R4M0.SDGICMD',
000150        'CW.FA.V8R8M2.CLIST','SYS1.SERBCLS','E87689.REXX.TOOLS')"
000160 "EXECUTIL SEARCHDD(YES)"
****** **************************** Bottom of Data ****************************
```

This screenshot shows the allocation program. This program is named 'SETUP3' & is residing in the PDS 'E87689.CLIST'. In the program, the system datasets are allocated first & then the user datasets. It should be noted that when an implicit execution request is made, the datasets are searched in the same order as specified in this program.



```
                    Infosys Technologies Ltd.,
             z/OS V1R4 - ISPF Master Application Menu

Option ===> TSO EXEC (SETUP3)

P   PDF          PDF Applications                   Userid  . : E87689
SD  SDSF         Spool Display and search Facility  Time  . . : 11:06
S   SORT         DFSORT Utility                     Terminal  : 3278
FA  File-Aid     Compuware File Aid Applications    Pf keys   : 24
FM  File Man     File Manager for z/OS              Screen  . : 1
M   MQSeries     WebSphere MQSeries 5.3.1           Language  : ENGLISH
X   Exit         Terminate ISPF using list/log defaults  Appl ID  : ISP
                                                    Release   : ISPF 5.2

Enter END command to terminate application
```

After writing the allocation program, issue the following command from any command line. This would execute the allocation program. Note that in the command, 'SETUP3' is the member name of the allocation program. It need not be 'SETUP3'. It can be given any valid name of your choice.

After writing the allocation program, issue the following command from any command line. Note that in the command, 'SETUP3' is the member name of the allocation program. It need not be 'SETUP3'. It can be given any valid name of your choice.

Now, there are two ways of executing this allocation program.

CASE1:

Using the short form command **'TSO EXEC (<allocation-program-member-name>)'**.

In this case, it is **'TSO EXEC (SETUP3)'**.

This short form command can be issued only when the allocation program is in a member of a PDS whose name is **'USERID.CLIST'.**

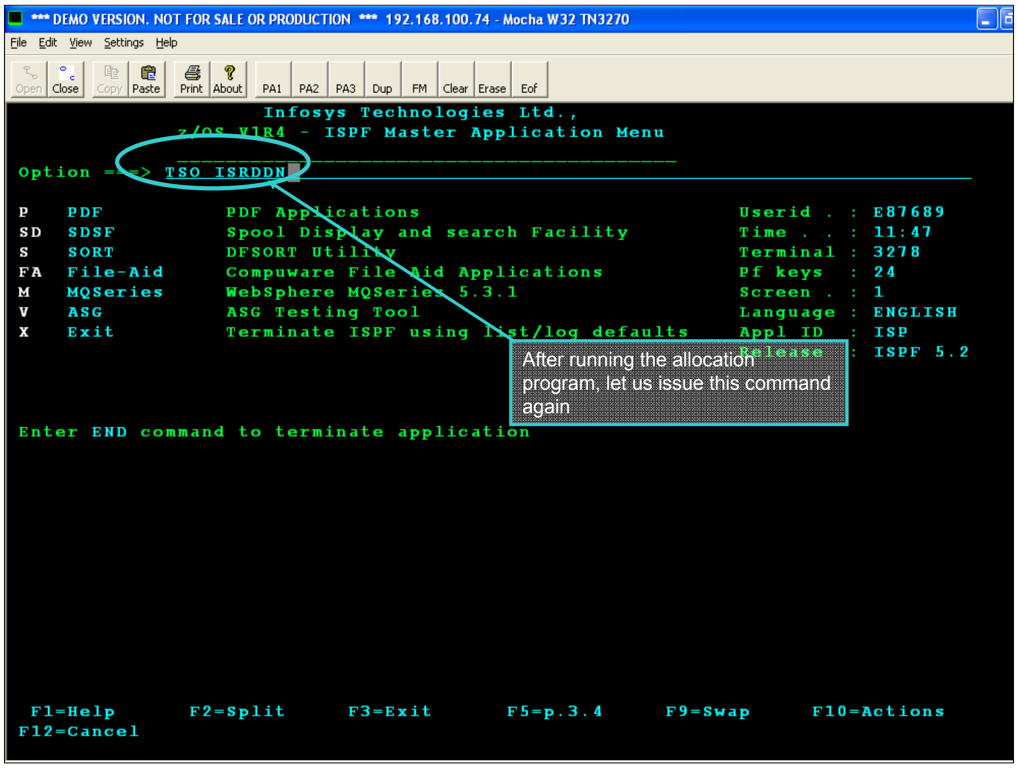
CASE2:

The complete command **'TSO EXEC <pdsname(member-name)>'**.

This command has to be used when the allocation program is not a part of the PDS USERID.CLIST.

Assume that the member 'SETUP3' is in a PDS by name **'USERID.REXX.PROGS'**, then to run the allocation program, the following command needs to be given:

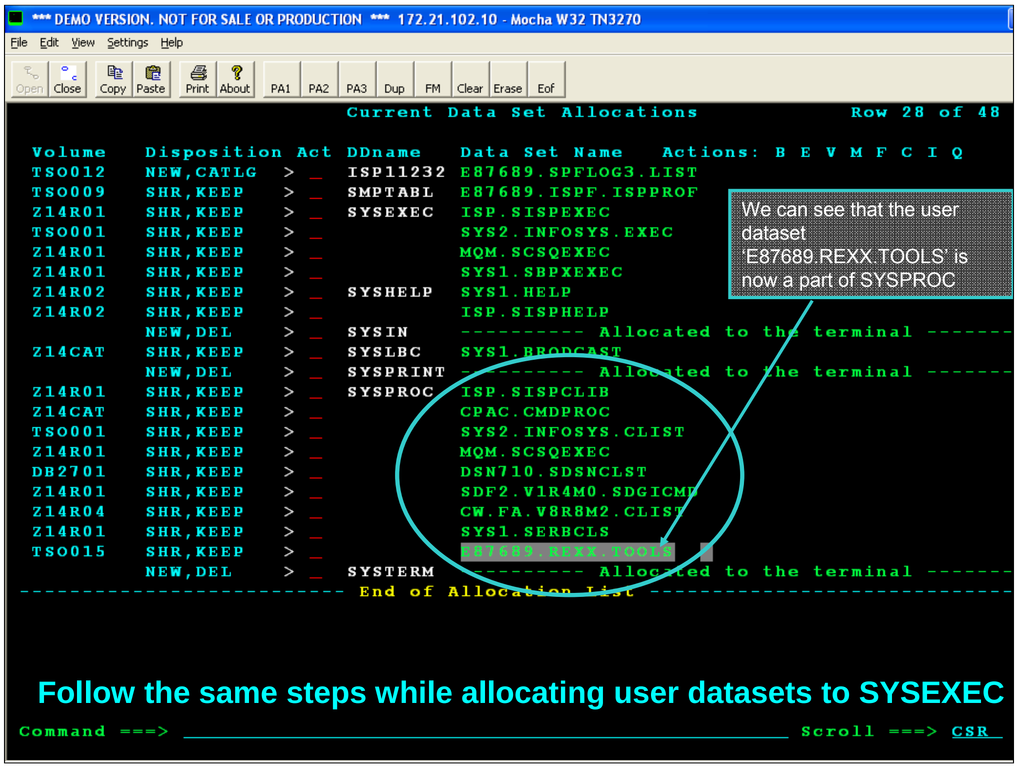
"TSO EXEC 'USERID.REXX.PROGS(SETUP3)' " **-->** Here, the fully qualified name of the PDS is mentioned. Hence, it is enclosed in quotes.



```
                      Infosys Technologies Ltd.,
              z/OS V1R4 - ISPF Master Application Menu

Option ===> TSO ISRDDN

P   PDF          PDF Applications                    Userid . : E87689
SD  SDSF         Spool Display and search Facility   Time . . : 11:47
S   SORT         DFSORT Utility                      Terminal : 3278
FA  File-Aid     Compuware File Aid Applications     Pf keys  : 24
M   MQSeries     WebSphere MQSeries 5.3.1            Screen . : 1
V   ASG          ASG Testing Tool                    Language : ENGLISH
X   Exit         Terminate ISPF using list/log defaults  Appl ID  : ISP
                                                     Release  : ISPF 5.2

Enter END command to terminate application

 F1=Help      F2=Split     F3=Exit      F5=p.3.4     F9=Swap     F10=Actions
F12=Cancel
```

*** DEMO VERSION. NOT FOR SALE OR PRODUCTION *** 192.168.100.74 - Mocha W32 TN3270
File Edit View Settings Help
Open Close Copy Paste Print About PA1 PA2 PA3 Dup FM Clear Erase Eof

After running the allocation program, let us issue this command again

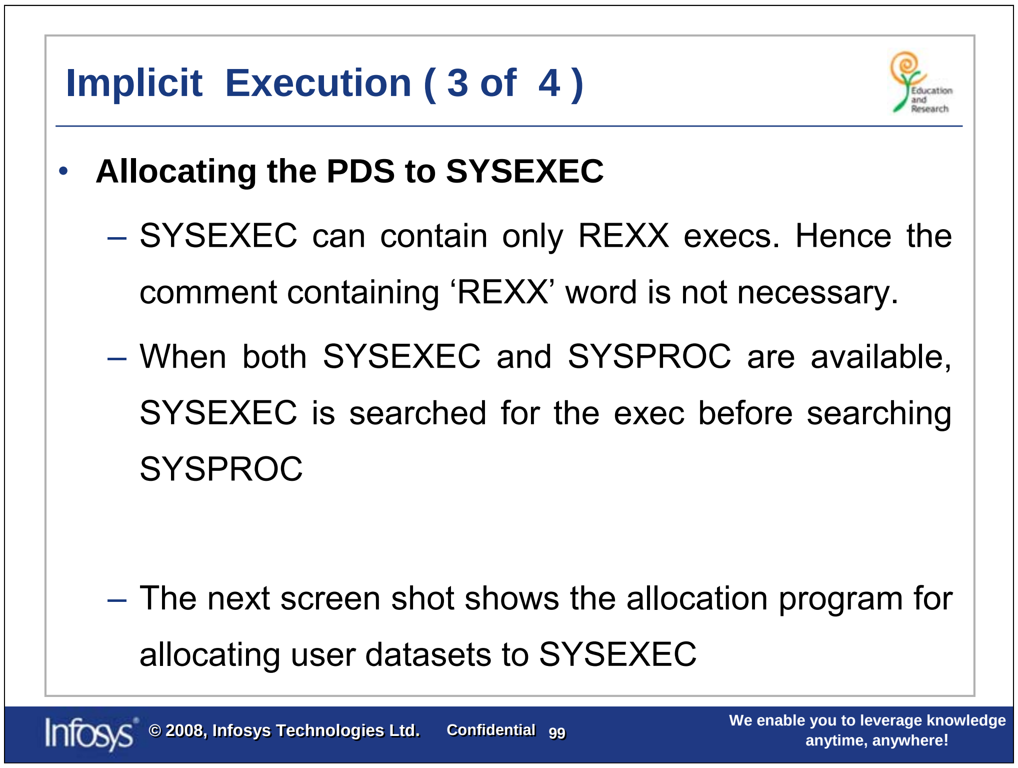


*** DEMO VERSION. NOT FOR SALE OR PRODUCTION *** 172.21.102.10 - Mocha W32 TN3270
File Edit View Settings Help
Open Close Copy Paste Print About PA1 PA2 PA3 Dup FM Clear Erase Eof

```
                           Current Data Set Allocations                  Row 28 of 48

 Volume    Disposition Act DDname   Data Set Name   Actions: B E V M F C I Q
 TSO012    NEW,CATLG   > _ ISP11232 E87689.SPFLOG3.LIST
 TSO009    SHR,KEEP    > _ SMPTABL  E87689.ISPF.ISPPROF
 Z14R01    SHR,KEEP    > _ SYSEXEC  ISP.SISPEXEC
 TSO001    SHR,KEEP    > _          SYS2.INFOSYS.EXEC
 Z14R01    SHR,KEEP    > _          MQM.SCSQEXEC
 Z14R01    SHR,KEEP    > _          SYS1.SBPXEXEC
 Z14R02    SHR,KEEP    > _ SYSHELP  SYS1.HELP
 Z14R02    SHR,KEEP    > _          ISP.SISPHELP
           NEW,DEL     > _ SYSIN    ---------- Allocated to the terminal -------
 Z14CAT    SHR,KEEP    > _ SYSLBC   SYS1.BRODCAST
           NEW,DEL     > _ SYSPRINT ---------- Allocated to the terminal -------
 Z14R01    SHR,KEEP    > _ SYSPROC  ISP.SISPCLIB
 Z14CAT    SHR,KEEP    > _          CPAC.CMDPROC
 TSO001    SHR,KEEP    > _          SYS2.INFOSYS.CLIST
 Z14R01    SHR,KEEP    > _          MQM.SCSQEXEC
 DB2701    SHR,KEEP    > _          DSN710.SDSNCLST
 Z14R01    SHR,KEEP    > _          SDF2.V1R4M0.SDGICMD
 Z14R04    SHR,KEEP    > _          CW.FA.V8R8M2.CLIST
 Z14R01    SHR,KEEP    > _          SYS1.SERBCLS
 TSO015    SHR,KEEP    > _          E87689.REXX.TOOLS
           NEW,DEL     > _ SYSTERM  ---------- Allocated to the terminal -------
-------------------------- End of Allocation List -----------------------------

Command ===> ____________________________________________________ Scroll ===> CSR
```

We can see that the user dataset 'E87689.REXX.TOOLS' is now a part of SYSPROC

Follow the same steps while allocating user datasets to SYSEXEC

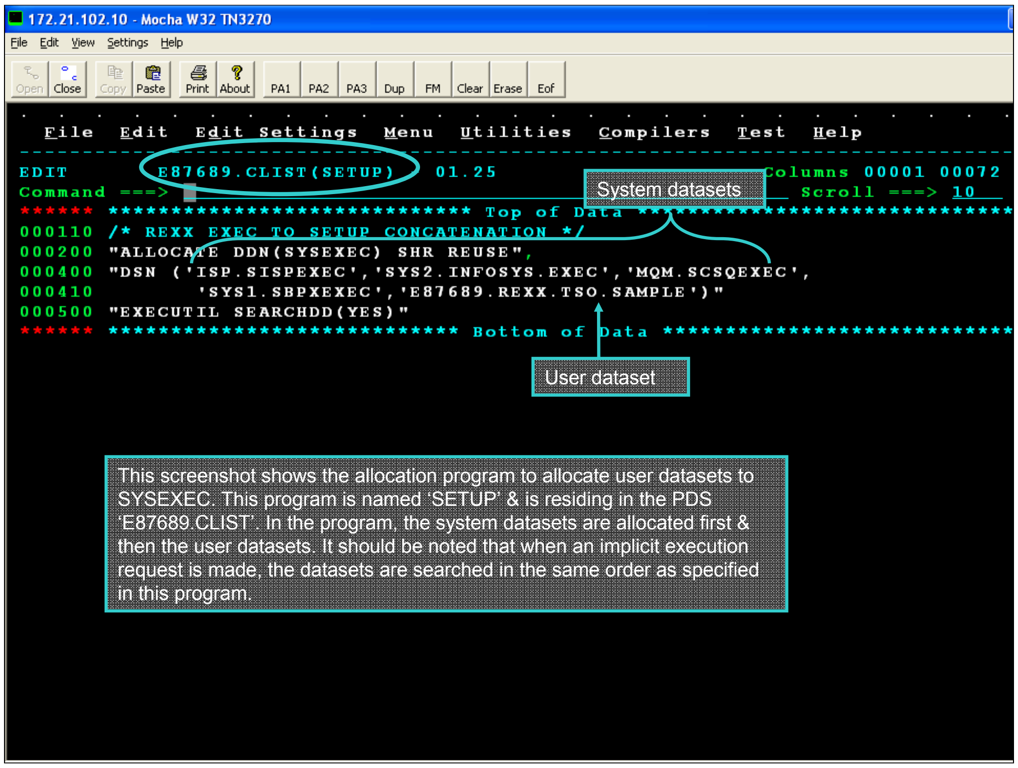


Implicit Execution (3 of 4)

- **Allocating the PDS to SYSEXEC**
  - SYSEXEC can contain only REXX execs. Hence the comment containing ‘REXX’ word is not necessary.
  - When both SYSEXEC and SYSPROC are available, SYSEXEC is searched for the exec before searching SYSPROC
  - The next screen shot shows the allocation program for allocating user datasets to SYSEXEC

Implicit Execution (3 of 4) - Notes

SYSEXEC contains only execs and it precedes SYSPROC in the search order when both the files are available . In other words , if we want our execs to be searched before CLISTS , then we may allocate the PDS (containing our execs) to SYSEXEC instead of SYSPROC .



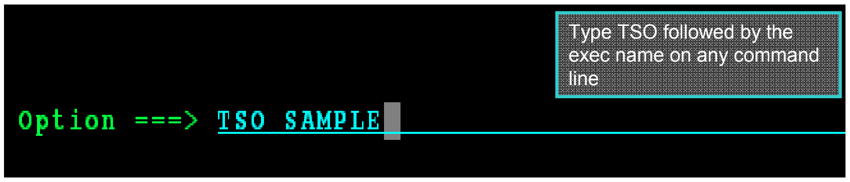
172.21.102.10 - Mocha W32 TN3270
File Edit View Settings Help
Open Close Copy Paste Print About PA1 PA2 PA3 Dup FM Clear Erase Eof
File Edit Edit_Settings Menu Utilities Compilers Test Help
System datasets
User dataset

```
EDIT       E87689.CLIST(SETUP)   01.25                    Columns 00001 00072
Command ===>                                                 Scroll ===> 10
****** ***************************** Top of Data ******************************
000110 /* REXX EXEC TO SETUP CONCATENATION */
000200 "ALLOCATE DDN(SYSEXEC) SHR REUSE",
000400 "DSN ('ISP.SISPEXEC','SYS2.INFOSYS.EXEC','MQM.SCSQEXEC',
000410        'SYS1.SBPXEXEC','E87689.REXX.TSO.SAMPLE')"
000500 "EXECUTIL SEARCHDD(YES)"
****** **************************** Bottom of Data ****************************
```

This screenshot shows the allocation program to allocate user datasets to SYSEXEC. This program is named 'SETUP' & is residing in the PDS 'E87689.CLIST'. In the program, the system datasets are allocated first & then the user datasets. It should be noted that when an implicit execution request is made, the datasets are searched in the same order as specified in this program.

Implicit Execution (4 of 4)

- **Executing the exec implicitly**
  - After allocating the PDS containing the exec to SYSEXEC or SYSPROC, the exec can be invoked from any command line just by it's name
  - Ex: Running an exec by name '**SAMPLE**' implicitly



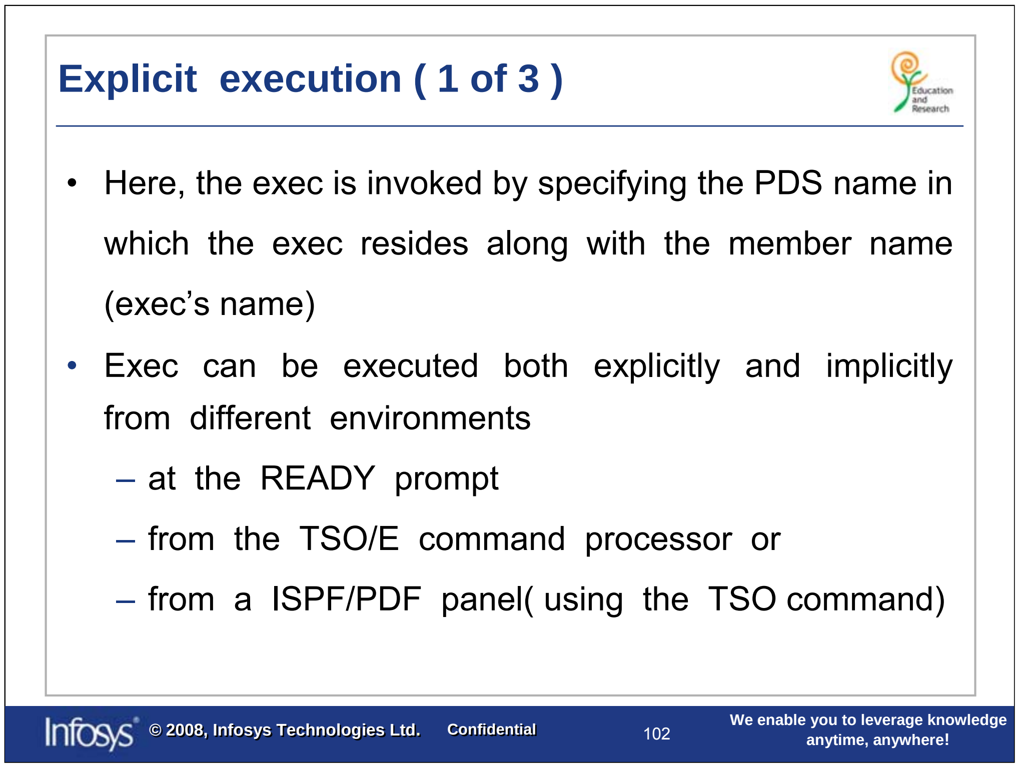
Type TSO followed by the exec name on any command line

```
Option ===> TSO SAMPLE
```

Implicit Execution (4 of 4) – Notes

Executing the exec implicitly

After allocating the PDS containing the exec to SYSEXEC or SYSPROC, the exec can be invoked from any command line just by it's name. The above slide shows an example where an EXEC named 'SAMPLE' is being invoked implicitly.

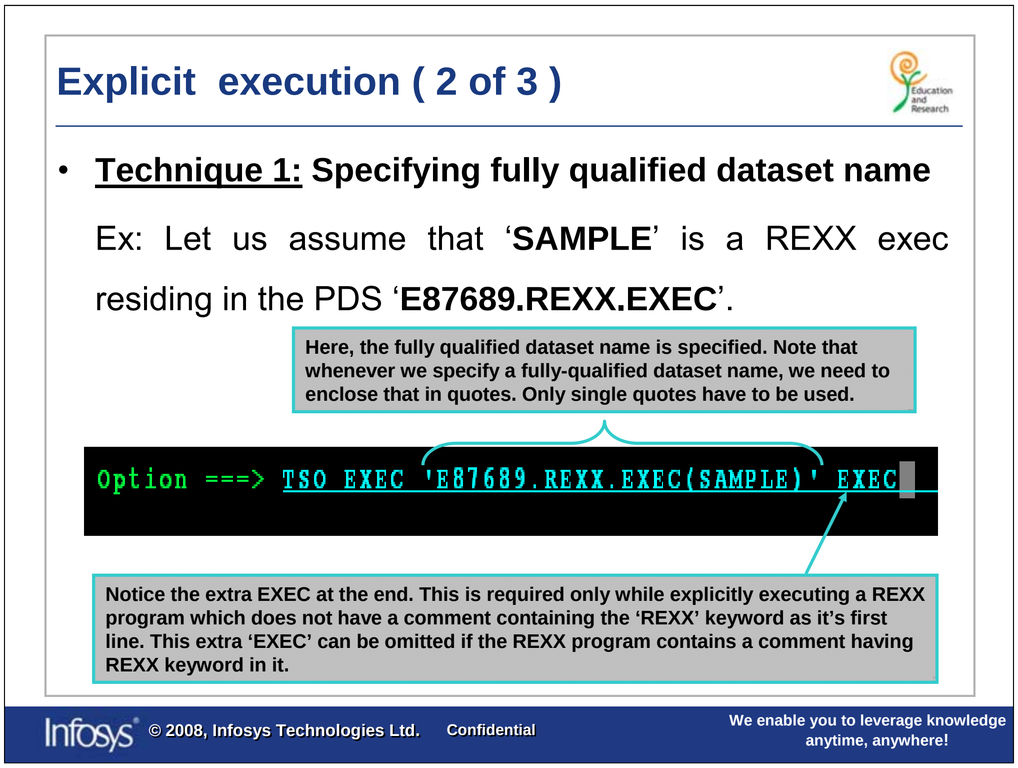


Explicit execution (1 of 3)

- Here, the exec is invoked by specifying the PDS name in which the exec resides along with the member name (exec's name)
- Exec can be executed both explicitly and implicitly from different environments
  - at the READY prompt
  - from the TSO/E command processor or
  - from a ISPF/PDF panel(using the TSO command)

Explicit execution (1 of 3) - Notes

Here, the exec is invoked by specifying the PDS name in which the exec resides along with the member name (exec's name). Please note that an exec can be executed both explicitly and implicitly from different environments – at the READY prompt , from the TSO/E command processor or from a ISPF/PDF panel (using the TSO command)



Explicit execution (2 of 3)

- **Technique 1:** **Specifying fully qualified dataset name**

  Ex: Let us assume that '**SAMPLE**' is a REXX exec residing in the PDS '**E87689.REXX.EXEC**'.

Here, the fully qualified dataset name is specified. Note that whenever we specify a fully-qualified dataset name, we need to enclose that in quotes. Only single quotes have to be used.

```
Option ===> TSO EXEC 'E87689.REXX.EXEC(SAMPLE)' EXEC
```

Notice the extra EXEC at the end. This is required only while explicitly executing a REXX program which does not have a comment containing the 'REXX' keyword as it's first line. This extra 'EXEC' can be omitted if the REXX program contains a comment having REXX keyword in it.

Explicit execution (2 of 3) – Notes

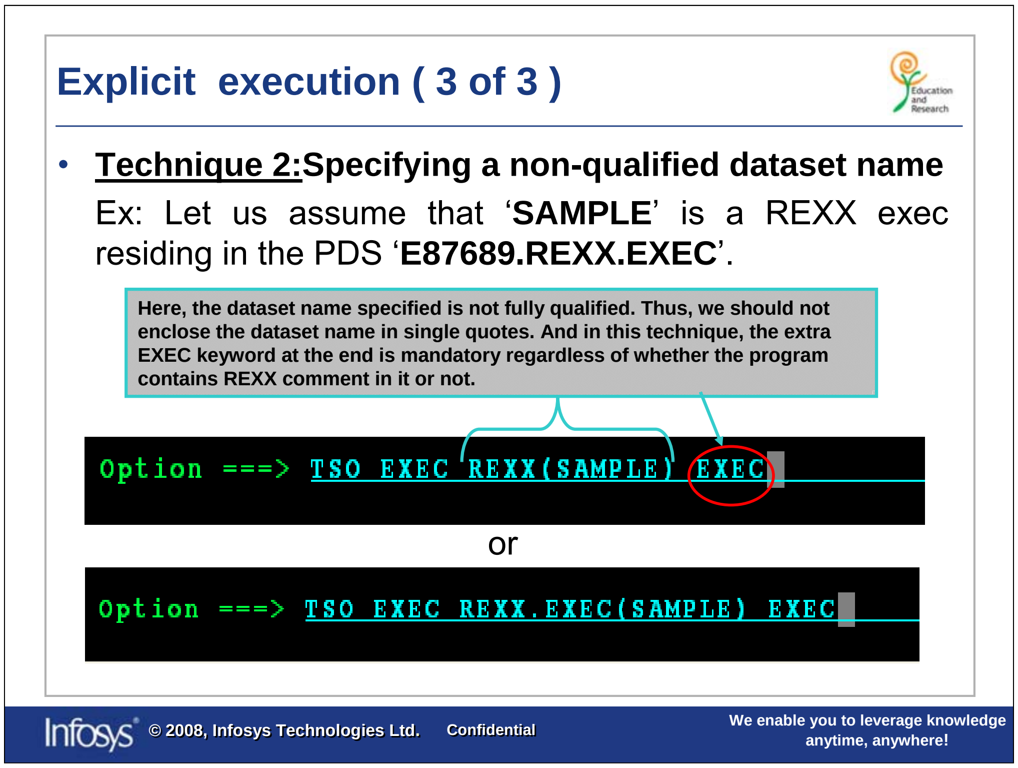
Technique 1: Specifying fully qualified dataset name

Ex: Let us assume that '**SAMPLE**' is a REXX exec residing in the PDS '**E87689.REXX.EXEC**'. The exec is invoked explicitly as below

==> TSO EXEC 'E87689.REXX.EXEC(SAMPLE)' EXEC

In the above example, fully qualified dataset name is specified. Note that whenever we specify a fully-qualified dataset name, we need to enclose that in quotes. Only single quotes have to be used. We cannot use double quotes. Try using double quotes & see what happens.

And also, towards the end of the command, there is an extra 'EXEC' keyword specified. In this case it is required only when executing a REXX program which does not have a comment containing the 'REXX' keyword as it's first line. This extra 'EXEC' can be omitted if the REXX program contains a comment having REXX keyword in it. However, this extra EXEC is mandatory for technique shown in the next slide.



Explicit execution (3 of 3)

- **<u>Technique 2:</u>Specifying a non-qualified dataset name**

  Ex: Let us assume that '**SAMPLE**' is a REXX exec residing in the PDS '**E87689.REXX.EXEC**'.

Here, the dataset name specified is not fully qualified. Thus, we should not enclose the dataset name in single quotes. And in this technique, the extra EXEC keyword at the end is mandatory regardless of whether the program contains REXX comment in it or not.

```
Option ===> TSO EXEC REXX(SAMPLE) EXEC
```

or

```
Option ===> TSO EXEC REXX.EXEC(SAMPLE) EXEC
```

Explicit execution (3 of 3) – Notes

<u>Technique 2:</u> Specifying a non-qualified dataset name

Ex: Let us assume that '**SAMPLE**' is a REXX exec residing in the PDS '**E87689.REXX.EXEC**'.

Then, the exec is invoked either as

==><u>TSO EXEC REXX(SAMPLE) EXEC</u> OR as

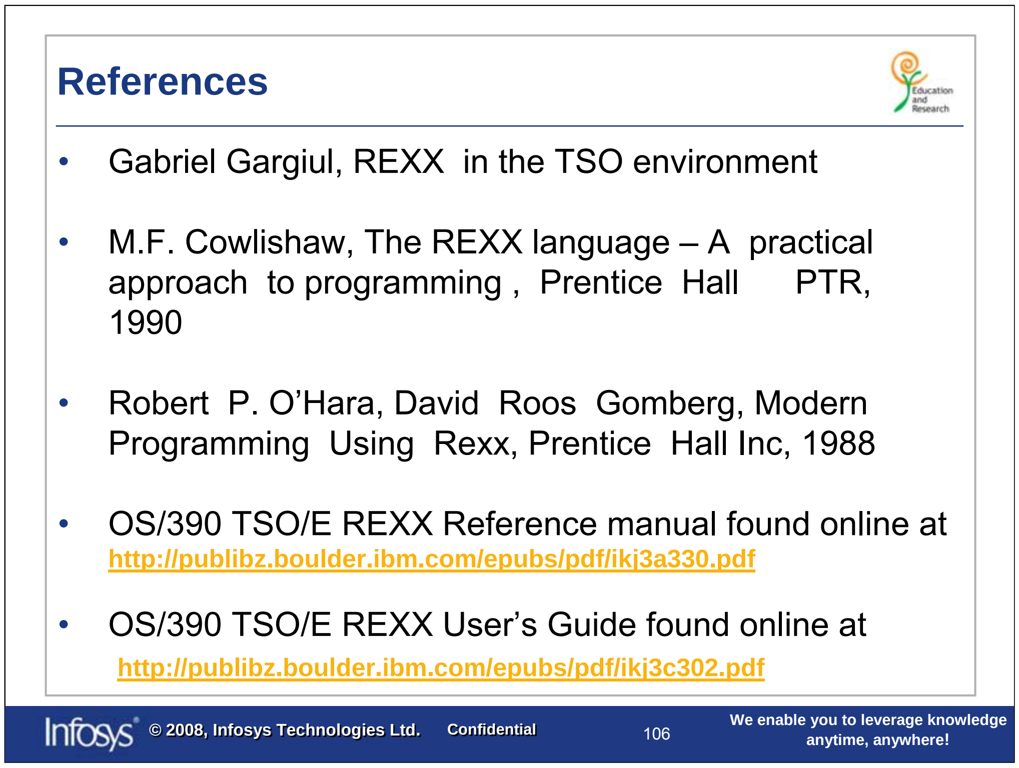
==><u>TSO EXEC REXX.EXEC(SAMPLE) EXEC</u>

Here, the dataset name specified is not fully qualified. Thus, we should not enclose the dataset name in single quotes. Moreover, it is possible to execute a program this way only when the PDS name ends with the keyword 'EXEC', as in this case **(E87689.REXX.EXEC)**. For ex, if the dataset name is 'E87689.REXX.PROGS' instead of 'E87689.REXX.EXEC', then it is not possible to execute the program 'SAMPLE' using the technique shown in this slide. We will have to execute it using technique 1 shown in previous slide (by specifying the fully qualified dataset name

→ TSO EXEC 'E87689.REXX.PROGS(SAMPLE)')

Also, in this technique the extra 'EXEC' keyword at the end is very much mandatory regardless of whether the rexx program contains a comment in the first line having REXX keyword in it. This is basically to differentiate REXX execs from CLISTs.

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



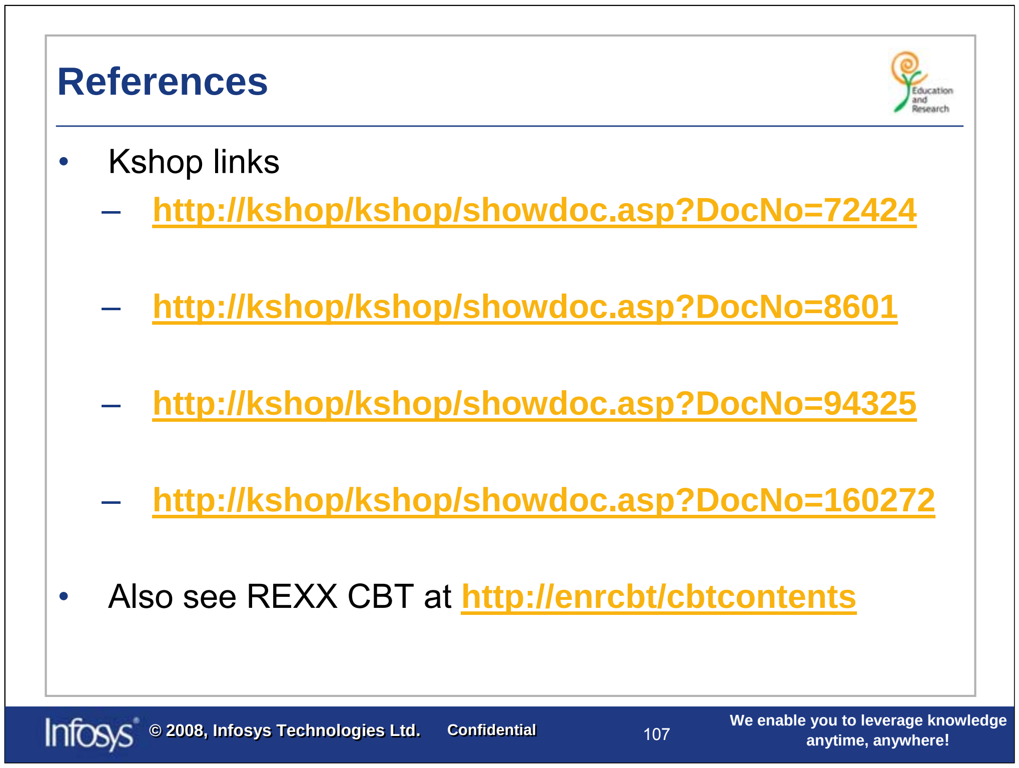
References

- 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

Infosys © 2008, Infosys Technologies Ltd. Confidential 106 We enable you to leverage knowledge anytime, anywhere!

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.